DRAFT INTERNATIONAL STANDARD ISO/IEC 19500-1

Attributed to ISO/IEC JTC 1 by the Central Secretariat (see page iii)

Voting begins on	Voting terminates on
2008-12-22	**2009-06-22**

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION • МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ • ORGANISATION INTERNATIONALE DE NORMALISATION
INTERNATIONAL ELECTROTECHNICAL COMMISSION • МЕЖДУНАРОДНАЯ ЭЛЕКТРОТЕХНИЧЕСКАЯ КОММИСИЯ • COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

PUBLICLY AVAILABLE SPECIFICATION PROCEDURE

Information technology — Open distributed processing — Common Object Request Broker Architecture (CORBA) specification —

Part 1: CORBA interfaces

Technologies de l'information — Traitement réparti ouvert — Spécification CORBA ("Common Object Request Broker Architecture") —

Partie 1: Interfaces CORBA

ICS 35.080

In accordance with the provisions of Council Resolution 21/1986 this DIS is circulated in the English language only.

Conformément aux dispositions de la Résolution du Conseil 21/1986, ce DIS est distribué en version anglaise seulement.

This Publicly Available Specification (PAS) is being submitted for Fast-track processing in accordance with the provisions of ISO/IEC JTC 1 Directives.

THIS DOCUMENT IS A DRAFT CIRCULATED FOR COMMENT AND APPROVAL. IT IS THEREFORE SUBJECT TO CHANGE AND MAY NOT BE REFERRED TO AS AN INTERNATIONAL STANDARD UNTIL PUBLISHED AS SUCH.

IN ADDITION TO THEIR EVALUATION AS BEING ACCEPTABLE FOR INDUSTRIAL, TECHNOLOGICAL, COMMERCIAL AND USER PURPOSES, DRAFT INTERNATIONAL STANDARDS MAY ON OCCASION HAVE TO BE CONSIDERED IN THE LIGHT OF THEIR POTENTIAL TO BECOME STANDARDS TO WHICH REFERENCE MAY BE MADE IN NATIONAL REGULATIONS.

RECIPIENTS OF THIS DRAFT ARE INVITED TO SUBMIT, WITH THEIR COMMENTS, NOTIFICATION OF ANY RELEVANT PATENT RIGHTS OF WHICH THEY ARE AWARE AND TO PROVIDE SUPPORTING DOCUMENTATION.

© International Organization for Standardization, 2008
© International Electrotechnical Commission, 2008

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

Copyright notice

This ISO document is a Draft International Standard and is copyright-protected by ISO. Except as permitted under the applicable laws of the user's country, neither this ISO draft nor any extract from it may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, photocopying, recording or otherwise, without prior written permission being secured.

Requests for permission to reproduce should be addressed to either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Reproduction may be subject to royalty payments or a licensing agreement.

Violators may be prosecuted.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

© ISO/IEC 2008 — All rights reserved

Not for Resale

NOTE FROM ITTF

The ballot on the transposition of a PAS into an international standard follows the JTC 1 PAS procedures contained in the JTC 1 Directives Section 14.

In the PAS process, the JTC 1 secretariat will inform JTC 1 National Bodies and Liaison Organisations, and those organisations authorised to be PAS submitters, of the initiation of the PAS ballot, the result of the ballot, and the identity of the JTC 1 committee which will be responsible for any future work.

For ballot, JTC 1 National Bodies and the PAS submitter will receive both the PAS to be transposed and the accompanying Explanatory Report. During the ballot JTC 1 members may propose changes to the PAS. These can be resolved with the PAS Submitter after completion of the ballot.

The period for combined DIS voting shall be six months. In order to be accepted the DIS must be supported by 75 % of the votes cast (abstention is not counted as a vote) and by two-thirds of the P-members voting of JTC 1.

In the case of a failure of the ballot, JTC 1 will make known to the Submitter the reasons which have led to the negative result. Based on this information, the Submitter may choose to re-submit the specification after modification.

Once the Draft International Standard has been approved by JTC 1 as an IS, it will be published.

© ISO/IEC 2008 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table of Contents

Foreword xiii

Introduction xv

1 Scope 1

2 Conformance and Compliance 1

3 Normative References 1

4 Part 1 Document 1

5 The Object Model 3

5.1 Overview 3

5.2 Object Semantics 4

5.2.1 Objects 4
5.2.2 Requests 4
5.2.3 Object Creation and Destruction 5
5.2.4 Types 5
5.2.5 Interfaces 6
5.2.6 Value Types 7
5.2.7 Abstract Interfaces 7
5.2.8 Operations 7
5.2.9 Attributes 9

5.3 Object Implementation 9

5.3.1 The Execution Model: Performing Services 9
5.3.2 The Construction Model 10

6 CORBA Overview 11

6.1 Structure of an Object Request Broker 11

6.1.1 Object Request Broker 15
6.1.2 Clients 16
6.1.3 Object Implementations 16
6.1.4 Object References 16
6.1.5 OMG Interface Definition Language 17
6.1.6 Mapping of OMG IDL to Programming Languages 17
6.1.7 Client Stubs 17
6.1.8 Dynamic Invocation Interface 17
6.1.9 Implementation Skeleton 18
6.1.10 Dynamic Skeleton Interface 18
6.1.11 Object Adapters 18

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.12 ORB Interface 18
6.1.13 Interface Repository 19
6.1.14 Implementation Repository 19
6.2 Example ORBs 19
6.2.1 Client- and Implementation-resident ORB 19
6.2.2 Server-based ORB 19
6.2.3 System-based ORB 20
6.2.4 Library-based ORB 20
6.3 Structure of a Client 20
6.4 Structure of an Object Implementation 21
6.5 Structure of an Object Adapter 23
6.6 CORBA Required Object Adapter 24
6.6.1 Portable Object Adapter 24
6.7 The Integration of Foreign Object Systems 24
7 OMG IDL Syntax and Semantics 27
7.1 Overview 27
7.2 Lexical Conventions 28
7.2.1 Tokens 31
7.2.2 Comments 31
7.2.3 Identifiers 31
7.2.4 Keywords 33
7.2.5 Literals 34
7.3 Preprocessing 36
7.4 OMG IDL Grammar 37
7.5 OMG IDL Specification 43
7.6 Import Declaration 44
7.7 Module Declaration 45
7.8 Interface Declaration 45
7.8.1 Interface Header 45
7.8.2 Interface Inheritance Specification 46
7.8.3 Interface Body 46
7.8.4 Forward Declaration 46
7.8.5 Interface Inheritance 47
7.8.6 Abstract Interface 49
7.8.7 Local Interface 50
7.9 Value Declaration 50
7.9.1 Regular Value Type 50
7.9.2 Boxed Value Type 52
7.9.3 Abstract Value Type 53
7.9.4 Value Forward Declaration 53
7.9.5 Valuetype Inheritance 54

© ISO/IEC 2008 - All rights reserved
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.10 Constant Declaration 55
7.10.1 Syntax 55
7.10.2 Semantics 56
7.11 Type Declaration 59
7.11.1 Basic Types 60
7.11.2 Constructed Types 62
7.11.3 Template Types 66
7.11.4 Complex Declarator 68
7.11.5 Native Types 68
7.11.6 Deprecated Anonymous Types 69
7.12 Exception Declaration 71
7.13 Operation Declaration 71
7.13.1 Operation Attribute 72
7.13.2 Parameter Declarations 72
7.13.3 Raises Expressions 73
7.13.4 Context Expressions 74
7.14 Attribute Declaration 74
7.15 Repository Identity Related Declarations 75
7.15.1 Repository Identity Declaration 75
7.15.2 Repository Identifier Prefix Declaration 76
7.15.3 Repository Id Conflict 77
7.16 Event Declaration 77
7.16.1 Regular Event Type 78
7.16.2 Abstract Event Type 78
7.16.3 Event Forward Declaration 78
7.16.4 Eventtype Inheritance 78
7.17 Component Declaration 79
7.17.1 Component 79
7.17.2 Component Header 79
7.17.3 Component Body 81
7.17.4 Event Sources—publishers and emitters 82
7.17.5 Event Sinks 83
7.17.6 Basic and Extended Components 83
7.18 Home Declaration 84
7.18.1 Home 84
7.18.2 Home Header 84
7.18.3 Home Body 85
7.19 CORBA Module 86
7.20 Names and Scoping 87
7.20.1 Qualified Names 87
7.20.2 Scoping Rules and Name Resolution 89
7.20.3 Special Scoping Rules for Type Names 91
8 ORB Interface 95
8.1 Overview 95

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.2 The ORB Operations 95
8.2.1 ORB Identity 101
8.2.2 Converting Object References to Strings 101
8.2.3 Getting Service Information 102
8.2.4 Creating a New Context 102
8.2.5 Thread-Related Operations 102
8.3 Object Reference Operations 105
8.3.1 Determining the Object Interface 107
8.3.2 Duplicating and Releasing Copies of Object References 107
8.3.3 Nil Object References 107
8.3.4 Equivalence Checking Operation 108
8.3.5 Probing for Object Non-Existence 108
8.3.6 Object Reference Identity 108
8.3.7 Type Coercion Considerations 110
8.3.8 Getting Policy Associated with the Object 110
8.3.9 Overriding Associated Policies on an Object Reference 111
8.3.10 Validating Connection 112
8.3.11 Getting the Domain Managers Associated with the Object 112
8.3.12 Getting Component Associated with the Object 113
8.3.13 Getting the ORB 113
8.3.14 LocalObject Operations 113
8.4 ValueBase Operations 114
8.5 ORB and OA Initialization and Initial References 115
8.5.1 ORB Initialization 115
8.5.2 Obtaining Initial Object References 117
8.5.3 Configuring Initial Service References 120
8.6 Context Object 122
8.6.1 Introduction 122
8.6.2 Context Object Operations 122
8.7 Current Object 125
8.8 Policy Object 126
8.8.1 Definition of Policy Object 126
8.8.2 Creation of Policy Objects 127
8.8.3 Usages of Policy Objects 129
8.8.4 Policy Associated with the Execution Environment 129
8.8.5 Specification of New Policy Objects 130
8.8.6 Standard Policies 131
8.9 Management of Policies 131
8.9.1 Client Side Policy Management 131
8.9.2 Server Side Policy Management 132
8.9.3 Policy Management Interfaces 132
8.10 Management of Policy Domains 134
8.10.1 Basic Concepts 134
8.10.2 Domain Management Operations 136
8.11 TypeCodes 138
8.11.1 The TypeCode Interface 138

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.11.2 TypeCode Constants 142
8.11.3 Creating TypeCodes 143
8.12 Exceptions 148
8.12.1 Definition of Terms 148
8.12.2 System Exceptions 148
8.12.3 Standard System Exception Definitions 150
8.12.4 Standard Minor Exception Codes 156
9 Value Type Semantics 157
9.1 Overview 157
9.2 Architecture 157
9.2.1 Abstract Values 158
9.2.2 Operations 158
9.2.3 Value Type vs. Interfaces 159
9.2.4 Parameter Passing 159
9.2.5 Substitutability Issues 160
9.2.6 Widening/Narrowing 161
9.2.7 Value Base Type 161
9.2.8 Life Cycle issues 161
9.2.9 Security Considerations 162
9.3 Standard Value Box Definitions 162
9.4 Language Mappings 163
9.4.1 General Requirements 163
9.4.2 Language Specific Marshaling 163
9.4.3 Language Specific Value Factory Requirements 163
9.4.4 Value Method Implementation 164
9.5 Custom Marshaling 164
9.5.1 Implementation of Custom Marshaling 164
9.5.2 Marshaling Streams 165
9.6 Access to the Sending Context Run Time 171
10 Abstract Interface Semantics 173
10.1 Overview 173
10.2 Semantics of Abstract Interfaces 173
10.3 Usage Guidelines 174
10.4 Example 174
10.5 Security Considerations 175
10.5.1 Passing Values to Trusted Domains 175
11 Dynamic Invocation Interface 177
11.1 Overview 177
11.1.1 Common Data Structures 177
11.1.2 Memory Usage 179

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.1.3 Return Status and Exceptions 179
11.2 Request Operations 179
11.2.1 create_request 180
11.2.2 add_arg 182
11.2.3 invoke 182
11.2.4 delete 183
11.2.5 send 183
11.2.6 poll_response 183
11.2.7 get_response 183
11.2.8 sendp 184
11.2.9 prepare 184
11.2.10sendc 184
11.3 ORB Operations 185
11.3.1 send_multiple_requests 185
11.3.2 get_next_response and poll_next_response 185
11.4 Polling 186
11.4.1 Abstract Valuetype Pollable 187
11.4.2 Abstract Valuetype DIIPollable 188
11.4.3 interface PollableSet 188
11.5 List Operations 189
11.5.1 create_list 190
11.5.2 add_item 190
11.5.3 free 191
11.5.4 free_memory 191
11.5.5 get_count 191
11.5.6 create_operation_list 191
12 Dynamic Skeleton Interface 193
12.1 Introduction 193
12.2 Overview 193
12.3 ServerRequestPseudo-Object 194
12.3.1 ExplicitRequest State: ServerRequestPseudo-Object 194
12.4 DSI: Language Mapping 195
12.4.1 ServerRequest's Handling of Operation Parameters 195
12.4.2 Registering Dynamic Implementation Routines 195
13 Dynamic Management of Any Values 197
13.1 Overview 197
13.2 DynAny API 198
13.2.1 Creating a DynAny Object 204
13.2.2 The DynAny Interface 206
13.2.3 The DynFixed Interface 210
13.2.4 The DynEnum Interface 210
13.2.5 The DynStruct Interface 211
13.2.6 The DynUnion Interface 212

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

13.2.7 The DynSequence Interface 214
13.2.8 The DynArray Interface 215
13.2.9 The DynValueCommon Interface 216
13.2.10 The DynValue Interface 216
13.2.11 The DynValueBox Interface 217
13.3 Usage in C++ Language 218
13.3.1 Dynamic Creation of CORBA::Any values 218
13.3.2 Dynamic Interpretation of CORBA::Any values 219
14 The Interface Repository 221
14.1 Overview 221
14.2 Scope of an Interface Repository 221
14.3 Implementation Dependencies 223
14.3.1 Managing Interface Repositories 223
14.4 Basics 224
14.4.1 Names and Identifiers 224
14.4.2 Types and TypeCodes 225
14.4.3 Interface Repository Objects 225
14.4.4 Structure and Navigation of the Interface Repository 226
14.5 Interface Repository Interfaces 228
14.5.1 Supporting Type Definitions 229
14.5.2 IRObject 230
14.5.3 Contained 231
14.5.4 Container 233
14.5.5 IDLType 238
14.5.6 Repository 238
14.5.7 ModuleDef 240
14.5.8 ConstantDef 240
14.5.9 TypedefDef 241
14.5.10 StructDef 241
14.5.11 UnionDef 242
14.5.12 EnumDef 243
14.5.13 AliasDef 243
14.5.14 PrimitiveDef 244
14.5.15 StringDef 244
14.5.16 WstringDef 244
14.5.17 FixedDef 245
14.5.18 SequenceDef 245
14.5.19 ArrayDef 245
14.5.20 ExceptionDef 246
14.5.21 AttributeDef 247
14.5.22 ExtAttributeDef 247
14.5.23 OperationDef 248
14.5.24 InterfaceDef 250
14.5.25 ExtInterfaceDef 252
14.5.26 AbstractInterfaceDef 253
14.5.27 ExtAbstractInterfaceDef 254

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.5.28LocalInterfaceDef 255
14.5.29ExtLocalInterfaceDef 256
14.5.30ValueMemberDef 256
14.5.31ValueDef 257
14.5.32ExtValueDef 260
14.5.33ValueBoxDef 262
14.5.34NativeDef 262
14.6 Component Interface Repository Interfaces 262
14.6.1 ComponentIR::Container 262
14.6.2 ComponentIR::Repository 264
14.6.3 ComponentIR::ProvidesDef 265
14.6.4 ComponentIR::UsesDef 265
14.6.5 ComponentIR::EventDef 266
14.6.6 ComponentIR::EventPortDef 266
14.6.7 ComponentIR::EmitsDef 267
14.6.8 ComponentIR::PublishesDef 268
14.6.9 ComponentIR::ConsumesDef 268
14.6.10ComponentIR::ComponentDef 268
14.6.11ComponentIR::FactoryDef 271
14.6.12ComponentIR::FinderDef 272
14.6.13ComponentIR::HomeDef 272
14.7 RepositoryIds 274
14.7.1 OMG IDL Format 275
14.7.2 RMI Hashed Format 275
14.7.3 DCE UUID Format 277
14.7.4 LOCAL Format 277
14.7.5 Pragma Directives for RepositoryId 277
14.7.6 For More Information 282
14.7.7 RepositoryIDs for OMG-Specified Types 282
14.7.8 Uniqueness Constraints on Repository IDs 283
14.8 OMG IDL for Interface Repository 284

15 The Portable Object Adapter 303
15.1 Overview 303
15.2 Abstract Model Description 303
15.2.1 Model Components 303
15.2.2 Model Architecture 305
15.2.3 POA Creation 306
15.2.4 Reference Creation 307
15.2.5 Object Activation States 308
15.2.6 Request Processing 309
15.2.7 Implicit Activation 309
15.2.8 Multi-threading 310
15.2.9 Dynamic Skeleton Interface 311
15.2.10Location Transparency 312
15.3 Interfaces 312
15.3.1 The Servant IDL Type 313

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

15.3.2 POAManager Interface 314
15.3.3 POAManagerFactory Interface 318
15.3.4 AdapterActivator Interface 319
15.3.5 ServantManager Interface 320
15.3.6 ServantActivator Interface 321
15.3.7 ServantLocator Interface 323
15.3.8 POA Policy Objects 325
15.3.9 POA Interface 328
15.3.10 Current Operations 337
15.4 IDL for PortableServer Module 338
15.5 UML Description of PortableServer 344
15.6 Usage Scenarios 346
15.6.1 Getting the Root POA 346
15.6.2 Creating a POA 347
15.6.3 Explicit Activation with POA-assigned Object Ids 347
15.6.4 Explicit Activation with User-assigned Object Ids 348
15.6.5 Creating References before Activation 349
15.6.6 Servant Manager Definition and Creation 349
15.6.7 Object Activation on Demand 351
15.6.8 Persistent Objects with POA-assigned Ids 352
15.6.9 Multiple Object Ids Mapping to a Single Servant 352
15.6.10 One Servant for All Objects 352
15.6.11 Single Servant, Many Objects and Types, Using DSI 355

16 Portable Interceptors 359

16.1 Introduction 359
16.1.1 Object Creation 359
16.1.2 Client Sends Request 360
16.1.3 Server Receives Request 361
16.1.4 Server Sends Reply 361
16.1.5 Client Receives Reply 362
16.2 General Behavior of Local Objects 362
16.3 Interceptor Interface 362
16.4 Request Interceptors 363
16.4.1 Design Principles 363
16.4.2 General Flow Rules 364
16.4.3 The Flow Stack Visual Model 364
16.4.4 The Request Interceptor Points 365
16.4.5 Client-Side Interceptor 365
16.4.6 Client-Side Interception Points 365
16.4.7 Client-Side Interception Point Flow 367
16.4.8 Server-Side Interceptor 370
16.4.9 Server-Side Interception Points 370
16.4.10 Server-Side Interception Point Flow 372
16.4.11 Request Information 375
16.4.12 RequestInfo Interface 375
16.4.13 ClientRequestInfo Interface 379

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.4.14ServerRequestInfo Interface 382
16.4.15ForwardRequest Exception 386
16.5 Portable Interceptor Current 386
16.5.1 Overview 386
16.5.2 Obtaining the Portable Interceptor Current 386
16.5.3 Portable Interceptor Current Interface 387
16.5.4 Use of Portable Interceptor Current 388
16.6 IOR Interceptor 392
16.6.1 Overview 392
16.6.2 An Abstract Model for Object Adapters 392
16.6.3 Object Reference Template 394
16.6.4 IORInterceptor Interface 396
16.6.5 IORInfo Interface 397
16.7 Interceptor Policy Objects 400
16.7.1 ProcessingMode Policy 400
16.8 PolicyFactory 401
16.8.1 PolicyFactory Interface 401
16.9 Registering Interceptors 401
16.9.1 ORBInitializer Interface 401
16.9.2 ORBInitInfo Interface 402
16.9.3 register_orb_initializer Operation 406
16.9.4 Notes about Registering Interceptors 408
16.10 Dynamic Initial References 408
16.10.1register_initial_reference 409
16.11 Module Dynamic 409
16.11.1NVList PIDL Represented by ParameterList IDL 410
16.11.2ContextList PIDL Represented by ContextList IDL 410
16.11.3ExceptionList PIDL Represented by ExceptionList IDL 410
16.11.4Context PIDL Represented by RequestContext IDL 410
16.12 Consolidated IDL 410
16.12.1Dynamic 410
16.12.2Portions of IOP Relevant to Portable Interceptor 411
16.12.3PortableInterceptor 411
17 CORBA Messaging 417
17.1 Section I - Introduction 417
17.2 Messaging Quality of Service 417
17.2.1 Rebind Support 419
17.2.2 Synchronization Scope 420
17.2.3 Request and Reply Priority 421
17.2.4 Request and Reply Timeout 422
17.2.5 Routing 424
17.2.6 Queue Ordering 425
17.3 Propagation of Messaging QoS 425
17.3.1 Structures 426

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.3.2 Messaging QoS Profile Component 426
17.3.3 Messaging QoS Service Context 426
17.4 Section II - Introduction 426
17.5 Running Example 428
17.6 Async Operation Mapping 428
17.6.1 Callback Model Signatures (sendc) 429
17.6.2 Polling Model Signatures (sendp) 431
17.7 Exception Delivery in the Callback Model 432
17.7.1 Messaging::ExceptionHolder valuetype 433
17.8 Type-Specific ReplyHandler Mapping 433
17.8.1 ReplyHandler Operations for NO_EXCEPTION Replies 434
17.8.2 ReplyHandler Operations for Exceptional Replies 435
17.8.3 Example 435
17.9 Generic Poller Value 436
17.9.1 operation_target 436
17.9.2 operation_name 437
17.9.3 associated_handler 437
17.9.4 is_from_poller 437
17.10 Type-Specific Poller Mapping 437
17.10.1Basic Type-Specific Poller 437
17.10.2Persistent Type-Specific Poller 439
17.10.3Example 440
17.11 Example Programmer Usage 441
17.11.1Example Programmer Usage (Examples Mapped to C++) 441
17.11.2Client-Side C++ Example for the Asynchronous Method Signatures 441
17.11.3Client-Side C++ Example of the Callback Model 442
17.11.4Client-Side C++ Example of the Polling Model 449
17.11.5Server Side 454
17.12 Section III - Introduction 455
17.13 Routing Object References 456
17.14 Message Routing 457
17.14.1Structures 459
17.14.2Interfaces 460
17.14.3Routing Protocol 462
17.15 Router Administration 467
17.15.1Constants 470
17.15.2Exceptions 470
17.15.3Valuetypes 470
17.15.4Interfaces 471
17.16 CORBA Messaging IDL 472
17.16.1Messaging Module 472
17.16.2MessageRouting Module 475

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex A - IDL Tags and Exceptions 493
Annex B - Legal Information 509
Annex C - OMG's Issue Reporting Procedure 513
Annex D - Acknowledgements 515
Index 517

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 19500-1 was prepared by Technical Committee ISO/IEC/TC JTC1, Information technology, Subcommittee SC 32, Data Management and Interchange in collaboration with the Object Management Group (OMG), following the submission and processing as a Publicly Available Specification (PAS) of the OMG Common Object Request Broker Architecture (CORBA) specification Part 1 Version 3.1 CORBA Interfaces.

ISO/IEC 19500-1 is related to:

- ITU-T Recommendation X.902 (1995) | ISO/IEC 10746-2:1995, Information Technology - Open Distributed Processing - Reference Model: Foundations
- ITU-T Recommendation X.903 (1995) | ISO/IEC 10746-3:1995, Information Technology - Open Distributed Processing - Reference Model: Architecture
- -ITU-T Recommendation X.920 (1997) | ISO/IEC 14750:1997, Information Technology - Open Distributed Processing - Interface Definition Language
- -ISO/IEC 19500-2, Information Technology - Open Distributed Processing - CORBA Specification Part 2: CORBA Interoperability

ISO/IEC 19500 consists of the following parts, under the general title *Information technology - Open distributed processing - CORBA specification*:

- Part 1: CORBA Interfaces
- Part 2: CORBA Interoperability
- Part 3: CORBA Components

It is the common core of the CORBA specification. Optional parts of CORBA, such as mappings to particular programming languages, Real-time CORBA extensions, and the minimum CORBA profile for embedded systems are documented in the other specifications that together comprise the complete CORBA specification. Please visit the CORBA download page at http://www.omg.org/technology/documents/corba_spec_catalog.htm to find the complete CORBA specification set.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This document is Draft version 00.01 of the PAS Submission for ISO/IEC 19500-1: *Information technology - Open distributed processing - CORBA Specification Part 1: CORBA Interfaces.*

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Introduction

The rapid growth of distributed processing has led to a need for a coordinating framework for this standardization and ITU-T Recommendations X.901-904 | ISO/IEC 10746, the Reference Model of Open Distributed Processing (RM-ODP) provides such a framework. It defines an architecture within which support of distribution, interoperability and portability can be integrated.

RM-ODP Part 2 (ISO/IEC 10746-2) defines the foundational concepts and modeling framework for describing distributed systems. The scopes and objectives of the RM-ODP Part 2 and the UML, while related, are not the same and, in a number of cases, the RM-ODP Part 2 and the UML specification use the same term for concepts which are related but not identical (e.g., interface). Nevertheless, a specification using the Part 2 modeling concepts can be expressed using UML with appropriate extensions (using stereotypes, tags, and constraints).

RM-ODP Part 3 (ISO/IEC 10746-3) specifies a generic architecture of open distributed systems, expressed using the foundational concepts and framework defined in Part 2. Given the relation between UML as a modeling language and Part 2 of the RM ODP standard, it is easy to show that UML is suitable as a notation for the individual viewpoint specifications defined by the RM-ODP.

This International Standard for CORBA Interfaces is a standard for the technology specification of an ODP system. It defines a technology to provide the infrastructure required to support functional distribution of an ODP system, specifying functions required to manage physical distribution, communications, processing and storage, and the roles of different technology objects in supporting those functions.

Context of CORBA

The key to understanding the structure of the CORBA architecture is the Reference Model, which consists of the following components:

- **Object Request Broker**, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in this manual.
- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains. For example, the Life Cycle Service defines conventions for creating, deleting, copying, and moving objects; it does not dictate how the objects are implemented in an application. Specifications for Object Services are contained in *CORBAservices*: *Common Object Services Specification*.
- **Common Facilities**, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility. Information about Common Facilities will be contained in *CORBAfacilities*: *Common Facilities Architecture*.
- **Application Objects**, which are products of a single vendor on in-house development group that controls their interfaces. Application Objects correspond to the traditional notion of applications, so they are not standardized by OMG. Instead, Application Objects constitute the uppermost layer of the Reference Model.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The Object Request Broker, then, is the core of the Reference Model. It is like a telephone exchange, providing the basic mechanism for making and receiving calls. Combined with the Object Services, it ensures meaningful communication between CORBA-compliant applications.

The architecture and specifications described in this standard are aimed at software designers and developers who want to produce applications that comply with OMG specifications for the Object Request Broker (ORB). The benefit of compliance is, in general, to be able to produce interoperable applications that are based on distributed, interoperating objects. The ORB provides the mechanisms by which objects transparently make requests and receive responses. Hence, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

Structure of this standard

Chapters 1-5 - Scope, **Conformance and Compliance**, **Normative references**, **Terms and definitions**, and **Symbols**.

Chapter 6 - The Object Model describes the computation model that underlies the CORBA architecture.

Chapter 7 - CORBA Overview contains the overall structure of the ORB architecture and includes information about CORBA interfaces and implementations.

Chapter 8 - OMG IDL Syntax and Semantics details the OMG interface definition language (OMG IDL), which is the language used to describe the interfaces that client objects call and object implementations provide.

Chapter 9 - ORB Interface defines the interface to the ORB functions that do not depend on object adapters: these operations are the same for all ORBs and object implementations.

Chapter 10 - Value Type Semantics describes the semantics of passing an object by value, which is similar to that of standard programming languages.

Chapter 11 - Abstract Interface Semantics explains an IDL abstract interface, which provides the capability to defer the determination of whether an object is passed by reference or by value until runtime.

Chapter 12 - The Dynamic Invocation Interface details the DII, the client’s side of the interface that allows dynamic creation and invocation of request to objects.

Chapter 13 - The Dynamic Skeleton Interface describes the DSI, the server's-side interface that can deliver requests from an ORB to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. DSI is the server's analogue of the client's Dynamic Invocation Interface (DII).

Chapter 14 - Dynamic Management of Any Values details the interface for the Dynamic Any type. This interface allows statically-typed programming languages such as C and Java to create or receive values of type Any without compile-time knowledge that the typer contained in the Any.

Chapter 15 - Interface Repository explains the component of the ORB that manages and provides access to a collection of object definitions.

Chapter 16 - Portable Object Adapter defines a group of IDL interfaces than an implementation uses to access ORB functions.

Chapter 17 - Portable Interceptors defines ORB operations that allow services such as security to be inserted in the invocation path.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chapter 18 - CORBA Messaging includes three general topics: Quality of Service, Asynchronous Method Invocations (to include Time-Independent or "Persistent" Requests), and the specification of interoperable Routing interfaces to support the transport of requests asynchronously from the handling of their replies.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Information technology — Open distributed processing — Common Object Request Broker Architecture (CORBA) specification —

Part 1: CORBA interfaces

1 Scope

This document specifies the CORBA Object Model and uses concepts from that model to define the operation of the Object Request Broker (ORB). The ORB is the basic mechanism by which objects transparently make requests to - and receive responses from - each other on the same machine or across a network. A client need not be aware of the mechanisms used to communicate with or activate an object, how the object is implemented, or where the object is located.

2 Conformance and Compliance

The minimum required for a CORBA-compliant system is adherence to the specifications in this standard and one mapping. Each additional language mapping is a separate, optional compliance point. Optional means users aren't required to implement these points if they are unnecessary at their site, but if implemented, they must adhere to the CORBA specifications to be called CORBA-compliant. For instance, if a vendor supports C++, their ORB must comply with the OMG IDL to C++ binding specified in the C++ Language Mapping Specification.

The CORBA Language Mappings have been separated from this standard and each language mapping is its own separate OMG specification.

3 Normative References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ITU-T Recommendation X.902 (1995) | ISO/IEC 10746-2:1995, Information Technology - Open Distributed Processing - Reference Model: Foundations
- ITU-T Recommendation X.903 (1995) | ISO/IEC 10746-3:1995, Information Technology - Open Distributed Processing - Reference Model: Architecture
- ITU-T Recommendation X.920 (1997) | ISO/IEC 14750:1997, Information Technology - Open Distributed Processing - Interface Definition Language

4 Part 1 Document

Part 1 of the CORBA, v3.1 specification consists of the following:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

1. The syntax and semantics of the interface definition language (IDL), which is used to describe the interfaces that client objects call and object implementations provide.
2. The interface to the ORB functions that do not depend on object adapters: these operations are the same for all ORBs and object implementations.
3. The semantics of passing an object by value.
4. An IDL abstract interface, which provides the capability to defer the determination of whether an object is passed by reference or by value until runtime.
5. The Dynamic Invocation Interface (DII), the client's side of the interface that allows dynamic creation and invocation of request to objects.
6. The Dynamic Skeleton Interface (DSI), the server's-side interface that can deliver requests from an ORB to an object implementation that does not have compile-time knowledge of the type of the object it is implementing.
7. The interface for the Dynamic Any type which allows statically-typed programming languages such as C and Java to create or receive values of type Any without compile-time knowledge that the typer contained in the Any.
8. The Interface Repository that manages and provides access to a collection of object definitions.
9. The Portable Object Adapter which defines a group of IDL interfaces that an implementation uses to access ORB functions.
10. ORB operations that allow services such as security to be inserted in the invocation path.
11. Messaging which covers: Quality of Service, Asynchronous Method Invocations (to include Time-Independent or “Persistent” Requests), and the specification of interoperable Routing interfaces to support the transport of requests asynchronously from the handling of their replies.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5 The Object Model

This clause describes the concrete object model that underlies the CORBA architecture. The model is derived from the abstract Core Object Model defined by the Object Management Group in the *Object Management Architecture Guide*.

5.1 Overview

The object model provides an organized presentation of object concepts and terminology. It defines a partial model for computation that embodies the key characteristics of objects as realized by the submitted technologies. The OMG object model is *abstract* in that it is not directly realized by any particular technology. The model described here is a *concrete* object model. A concrete object model may differ from the abstract object model in several ways:

- It may *elaborate* the abstract object model by making it more specific, for example, by defining the form of request parameters or the language used to specify types.
- It may *populate* the model by introducing specific instances of entities defined by the model, for example, specific objects, specific operations, or specific types.
- It may *restrict* the model by eliminating entities or placing additional restrictions on their use.

An object system is a collection of objects that isolates the requestors of services (clients) from the providers of services by a well-defined encapsulating interface. In particular, clients are isolated from the implementations of services as data representations and executable code.

The object model first describes concepts that are meaningful to clients, including such concepts as object creation and identity, requests and operations, types and signatures. It then describes concepts related to object implementations, including such concepts as methods, execution engines, and activation.

The object model is most specific and prescriptive in defining concepts meaningful to clients. The discussion of object implementation is more suggestive, with the intent of allowing maximal freedom for different object technologies to provide different ways of implementing objects.

There are some other characteristics of object systems that are outside the scope of the object model. Some of these concepts are aspects of application architecture, some are associated with specific domains to which object technology is applied. Such concepts are more properly dealt with in an architectural reference model. Examples of excluded concepts are compound objects, links, copying of objects, change management, and transactions. Also outside the scope of the object model are the details of control structure: the object model does not say whether clients and/or servers are single-threaded or multi-threaded, and does not specify how event loops are programmed nor how threads are created, destroyed, or synchronized.

This object model is an example of a *classical object model*, where a client sends a message to an object. Conceptually, the object interprets the message to decide what service to perform. In the classical model, a message identifies an object and zero or more actual parameters. As in most classical object models, a distinguished first parameter is required, which identifies the operation to be performed; the interpretation of the message by the object involves selecting a method based on the specified operation. Operationally, of course, method selection could be performed either by the object or the ORB.

5.2 Object Semantics

An object system provides services to clients. A *client* of a service is any entity capable of requesting the service. This sub clause defines the concepts associated with object semantics, that is, the concepts relevant to clients.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.2.1 Objects

An object system includes entities known as objects. An *object* is an identifiable, encapsulated entity that provides one or more services that can be requested by a client.

5.2.2 Requests

Clients request services by issuing requests.

The term *request* is broadly used to refer to the entire sequence of causally related events that transpires between a client initiating it and the last event causally associated with that initiation. For example:

- the client receives the final response associated with that *request* from the server,
- the server carries out the associated operation in case of a oneway request, or
- the sequence of events associated with the *request* terminates in a failure of some sort. The initiation of a Request is an event.

The information associated with a request consists of an operation, a target object, zero or more (actual) parameters, and an optional request context.

A *request form* is a description or pattern that can be evaluated or performed multiple times to cause the issuing of requests. As described in the OMG IDL Syntax and Semantics clause, request forms are defined by particular language bindings. An alternative request form consists of calls to the dynamic invocation interface to create an invocation structure, add arguments to the invocation structure, and to issue the invocation (refer to the *Dynamic Invocation Interface* clause for descriptions of these request forms).

A *value* is anything that may be a legitimate (actual) parameter in a request. More particularly, a value is an instance of an OMG IDL data type. There are non-object values, as well as values that reference objects.

An *object reference* is a value that reliably denotes a particular object. Specifically, an object reference will identify the same object each time the reference is used in a request (subject to certain pragmatic limits of space and time). An object may be denoted by multiple, distinct object references.

A request may have parameters that are used to pass data to the target object; it may also have a request context that provides additional information about the request. A request context is a mapping from strings to strings.

A request causes a service to be performed on behalf of the client. One possible outcome of performing a service is returning to the client the results, if any, defined for the request.

If an abnormal condition occurs during the performance of a request, an exception is returned. The exception may carry additional return parameters particular to that exception.

The request parameters are identified by position. A parameter may be an input parameter, an output parameter, or an input-output parameter. A request may also return a single *return result value*, as well as the results stored into the output and input-output parameters.

The following semantics hold for all requests:

- Any aliasing of parameter values is neither guaranteed removed nor guaranteed to be preserved.
- The order in which aliased output parameters are written is not guaranteed.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- The return result and the values stored into the output and input-output parameters are undefined if an exception is returned.

For descriptions of the values and exceptions that are permitted, see Types on page 5 and Exceptions on page 8.

5.2.3 Object Creation and Destruction

Objects can be created and destroyed. From a client's point of view, there is no special mechanism for creating or destroying an object. Objects are created and destroyed as an outcome of issuing requests. The outcome of object creation is revealed to the client in the form of an object reference that denotes the new object.

5.2.4 Types

A *type* is an identifiable entity with an associated predicate (a single-argument mathematical function with a boolean result) defined over entities. An entity *satisfies* a type if the predicate is true for that entity. An entity that satisfies a type is called a *member of the type*.

Types are used in signatures to restrict a possible parameter or to characterize a possible result.

The *extension of a type* is the set of entities that satisfy the type at any particular time.

An *object type* is a type whose members are object references. In other words, an object type is satisfied only by object references.

Constraints on the data types in this model are shown in this sub clause.

5.2.4.1 Basic types

- 16-bit, 32-bit, and 64-bit signed and unsigned 2's complement integers.
- Single-precision (32-bit), double-precision (64-bit), and double-extended (a mantissa of at least 64 bits, a sign bit and an exponent of at least 15 bits) IEEE floating point numbers.
- Fixed-point decimal numbers of up to 31 significant digits.
- Characters, as defined in ISO Latin-1 (8859.1) and other single- or multi-byte character sets.
- A boolean type taking the values TRUE and FALSE.
- An 8-bit opaque detectable, guaranteed to *not* undergo any conversion during transfer between systems.
- Enumerated types consisting of ordered sequences of identifiers.
- A string type, which consists of a variable-length array of characters; the length of the string is a non-negative integer, and is available at run-time. The length may have a maximum bound defined.
- A wide character string type, which consists of a variable-length array of (fixed width) wide characters; the length of the wide string is a non-negative integer, and is available at run-time. The length may have a maximum bound defined.
- A container type "any," which can represent any possible basic or constructed type.
- Wide characters that may represent characters from any wide character set.
- Wide character strings, which consist of a length, available at runtime, and a variable-length array of (fixed width) wide characters.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.2.4.2 Constructed types

- A record type (called struct), which consists of an ordered set of (name,value) pairs.
- A discriminated union type, which consists of a discriminator (whose exact value is always available) followed by an instance of a type appropriate to the discriminator value.
- A sequence type, which consists of a variable-length array of a single type; the length of the sequence is available at run-time.
- An array type, which consists of a fixed-shape multidimensional array of a single type.
- An interface type, which specifies the set of operations that an instance of that type must support.
- A value type, which specifies state as well as a set of operations that an instance of that type must support.

Entities in a request are restricted to values that satisfy these type constraints. The legal entities are shown in Figure 5.1. No particular representation for entities is defined.

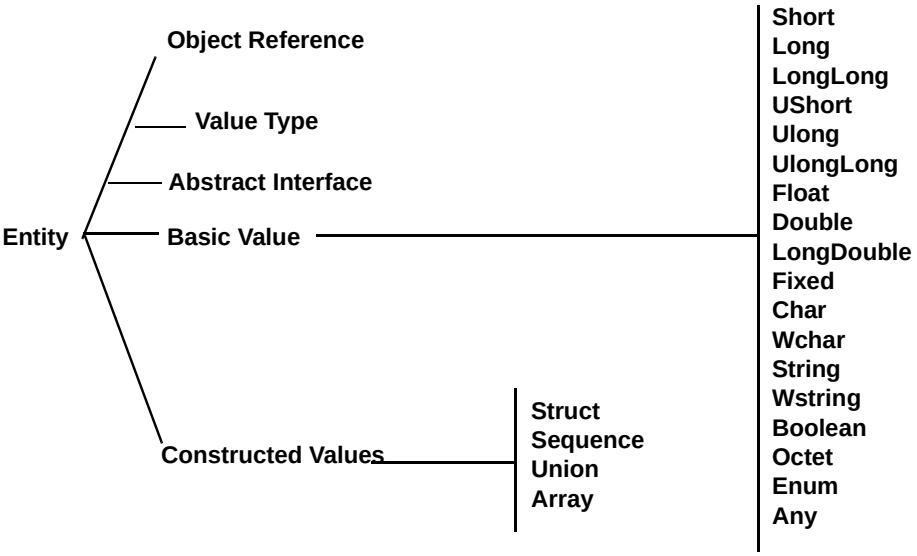


Figure 5.1 - Legal Values

5.2.5 Interfaces

An *interface* is a description of a set of possible operations that a client may request of an object, through that interface. It provides a syntactic description of how a service provided by an object supporting this interface, is accessed via this set of operations. An object *satisfies* an interface if it provides its service through the operations of the interface according to the specification of the operations (see Operations on page 7).

The *interface type* for a given interface is an object type, such that an object reference will satisfy the type, if and only if the referent object also satisfies the interface.

Interfaces are specified in OMG IDL. Interface inheritance provides the composition mechanism for permitting an object to support multiple interfaces. The *principal interface* is simply the most-specific interface that the object supports, and consists of all operations in the transitive closure of the interface inheritance graph.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Interfaces satisfy the Liskov substitution principle. If interface A is derived from interface B, then a reference to an object that supports interface A can be used where the formal type of a parameter is declared to be B.

5.2.6 Value Types

A *value type* is an entity, which shares many of the characteristics of interfaces and structs. It is a description of both a set of operations that a client may request and of state that is accessible to a client. Instances of a value type are always local concrete implementations in some programming language.

A value type, in addition to the operations and state defined for itself, may also inherit from other value types, and through multiple inheritance support other interfaces.

Value types are specified in OMG IDL.

An *abstract value type* describes a value type that is a “pure” bundle of operations with no state.

5.2.7 Abstract Interfaces

An *abstract interface* is an entity, which may at runtime represent either a regular interface (see Interfaces on page 6) or a value type (see Value Types on page 7). Like an abstract value type, it is a pure bundle of operations with no state. Unlike an abstract value type, it does not imply pass-by-value semantics, and unlike a regular interface type, it does not imply pass-by-reference semantics. Instead, the entity’s runtime type determines which of these semantics are used.

5.2.8 Operations

An *operation* is an identifiable entity that denotes the indivisible primitive of service provision that can be requested. The act of requesting an operation is referred to as *invoking the operation*. An operation is identified by an *operation identifier*.

An operation has a *signature* that describes the legitimate values of request parameters and returned results. In particular, a *signature* consists of:

- A specification of the parameters required in requests for that operation.
- A specification of the result of the operation.
- An identification of the user exceptions that may be raised by an invocation of the operation.
- A specification of additional contextual information that may affect the invocation.
- An indication of the execution semantics the client should expect from an invocation of the operation.

Operations are (potentially) *generic*, meaning that a single operation can be uniformly invoked on objects with different implementations, possibly resulting in observably different behavior. Genericity is achieved in this model via interface inheritance in IDL and the total decoupling of implementation from interface specification.

The general form for an operation signature is:

[oneway] <op_type_spec> <identifier> (param1, ..., paramL)
[raises(except1,...,exceptN)] [context(name1, ..., nameM)]

where:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- The optional **oneway** keyword indicates that best-effort semantics are expected of requests for this operation; the default semantics are exactly-once if the operation successfully returns results or at-most-once if an exception is returned.
- The **<op_type_spec>** is the type of the return result.
- The **<identifier>** provides a name for the operation in the interface.
- The operation parameters needed for the operation; they are flagged with the modifiers **in**, **out**, or **inout** to indicate the direction in which the information flows (with respect to the object performing the request).
- The optional **raises** expression indicates which user-defined exceptions can be signaled to terminate an invocation of this operation; if such an expression is not provided, no user-defined exceptions will be signaled.
- The optional **context** expression indicates which request context information will be available to the object implementation; no other contextual information is required to be transported with the request.

Parameters

A parameter is characterized by its mode and its type. The *mode* indicates whether the value should be passed from client to server (**in**), from server to client (**out**), or both (**inout**). The parameter's type constrains the possible value, which may be passed in the directions dictated by the mode.

Return Result

The return result is a distinguished **out** parameter.

Exceptions

An *exception* is an indication that an operation request was not performed successfully. An exception may be accompanied by additional, exception-specific information.

The additional, exception-specific information is a specialized form of record. As a record, it may consist of any of the types described in Types on page 5.

All signatures implicitly include the system exceptions; the standard system exceptions are described in System Exceptions on page 148.

Contexts

A *request context* provides additional, operation-specific information that may affect the performance of a request.

Execution Semantics

Two styles of execution semantics are defined by the object model:

- At-most-once: if an operation request returns successfully, it was performed exactly once; if it returns an exception indication, it was performed at-most-once.
- Best-effort: a best-effort operation is a request-only operation (i.e., it cannot return any results and the requester never synchronizes with the completion, if any, of the request).

The execution semantics to be expected is associated with an operation. This prevents a client and object implementation from assuming different execution semantics.

Note that a client is able to invoke an at-most-once operation in a synchronous or deferred-synchronous manner.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.2.9 Attributes

An interface may have attributes. An attribute is logically equivalent to declaring a pair of accessor functions: one to retrieve the value of the attribute and one to set the value of the attribute.

An attribute may be read-only, in which case only the retrieval accessor function is defined.

5.3 Object Implementation

This sub clause defines the concepts associated with object implementation (i.e., the concepts relevant to realizing the behavior of objects in a computational system).

The implementation of an object system carries out the computational activities needed to effect the behavior of requested services. These activities may include computing the results of the request and updating the system state. In the process, additional requests may be issued.

The implementation model consists of two parts: the execution model and the construction model. The execution model describes how services are performed. The construction model describes how services are defined.

5.3.1 The Execution Model: Performing Services

A requested service is performed in a computational system by executing code that operates upon some data. The data represents a component of the state of the computational system. The code performs the requested service, which may change the state of the system.

Code that is executed to perform a service is called a *method*. A method is an immutable description of a computation that can be interpreted by an execution engine. A method has an immutable attribute called a *method format* that defines the set of execution engines that can interpret the method. An *execution engine* is an abstract machine (not a program) that can interpret methods of certain formats, causing the described computations to be performed. An execution engine defines a dynamic context for the execution of a method. The execution of a method is called a *method activation.*

When a client issues a request, a method of the target object is called. The input parameters passed by the requestor are passed to the method and the output and input-output parameters and return result value (or exception and its parameters) are passed back to the requestor.

Performing a requested service causes a method to execute that may operate upon an object's persistent state. If the persistent form of the method or state is not accessible to the execution engine, it may be necessary to first copy the method or state into an execution context. This process is called *activation*; the reverse process is called *deactivation.*

5.3.2 The Construction Model

A computational object system must provide mechanisms for realizing behavior of requests. These mechanisms include definitions of object state, definitions of methods, and definitions of how the object infrastructure is to select the methods to execute and to select the relevant portions of object state to be made accessible to the methods. Mechanisms must also be provided to describe the concrete actions associated with object creation, such as association of the new object with appropriate methods.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

An *object implementation*—or *implementation*, for short—is a definition that provides the information needed to create an object and to allow the object to participate in providing an appropriate set of services. An implementation typically includes, among other things, definitions of the methods that operate upon the state of an object. It also typically includes information about the intended types of the object.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6 CORBA Overview

The Common Object Request Broker Architecture (CORBA) is structured to allow integration of a wide variety of object systems. The motivation for some of the features may not be apparent at first, but as we discuss the range of implementations, policies, optimizations, and usages we expect to encompass, the value of the flexibility becomes clearer.

6.1 Structure of an Object Request Broker

Figure 6.1 shows a request being sent by a client to an object implementation. The Client is the entity that wishes to perform an operation on the object and the Object Implementation is the code and data that actually implements the object.

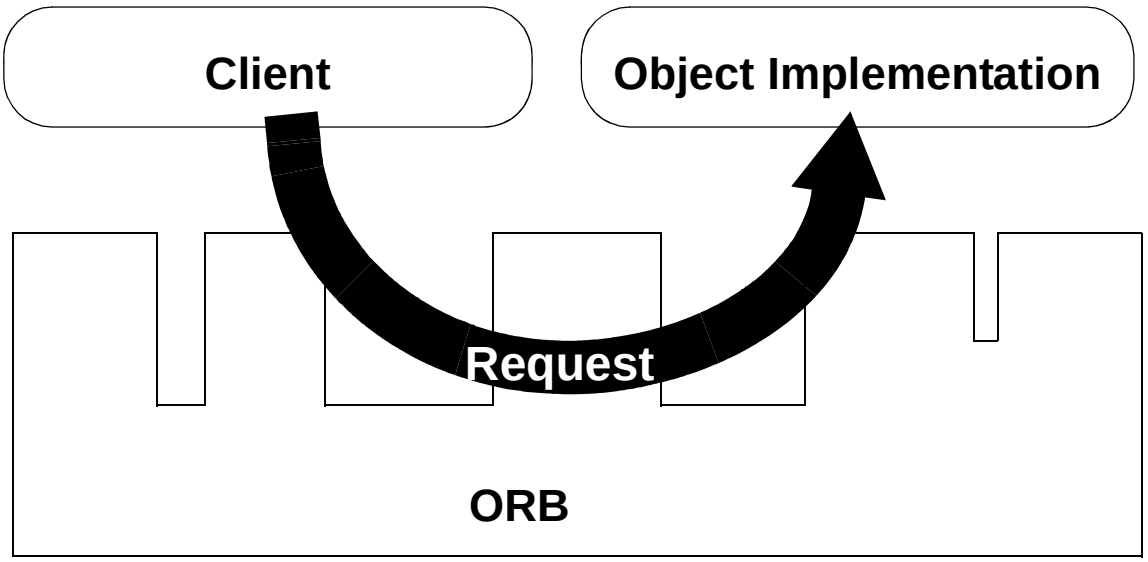


Figure 6.1 - A Request Being Sent Through the Object Request Broker

The ORB is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. The interface the client sees is completely independent of where the object is located, what programming language it is implemented in, or any other aspect that is not reflected in the object's interface.

Figure 6.2 shows the structure of an individual Object Request Broker (ORB). The interfaces to the ORB are shown by striped boxes, and the arrows indicate whether the ORB is called or performs an up-call across the interface.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

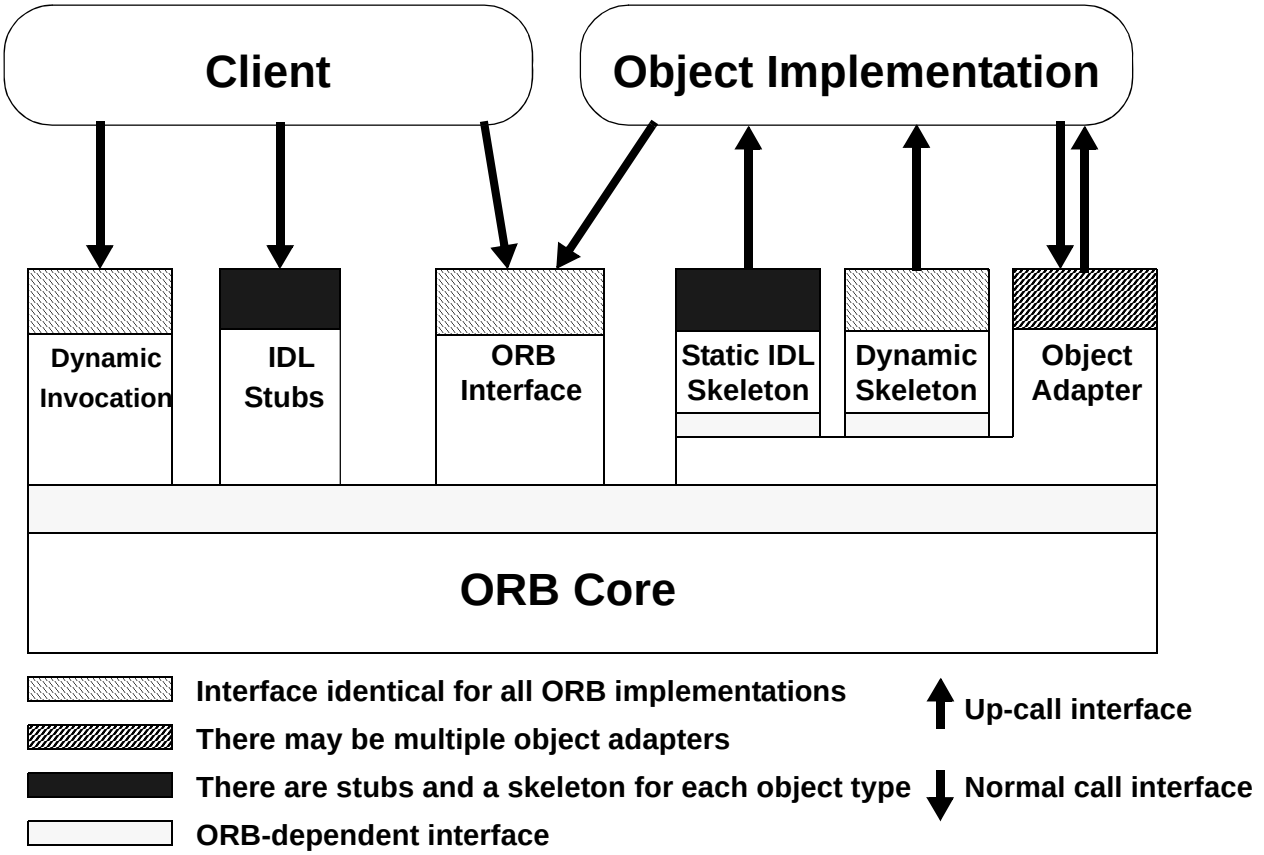


Figure 6.2 - The Structure of Object Request Interfaces

To make a request, the Client can use the Dynamic Invocation interface (the same interface independent of the target object's interface) or an OMG IDL stub (the specific stub depending on the interface of the target object). The Client can also directly interact with the ORB for some functions.

The Object Implementation receives a request as an up-call either through the OMG IDL generated skeleton or through a dynamic skeleton. The Object Implementation may call the Object Adapter and the ORB while processing a request or at other times.

Definitions of the interfaces to objects can be defined in two ways. 1) Interfaces can be defined statically in an interface definition language, called the OMG Interface Definition Language (OMG IDL). This language defines the types of objects according to the operations that may be performed on them and the parameters to those operations. 2) Alternatively, or in addition, interfaces can be added to an Interface Repository service. This service represents the components of an interface as objects, permitting run-time access to these components. In any ORB implementation, the Interface Definition Language (which may be extended beyond its definition in this document) and the Interface Repository have equivalent expressive power.

The client performs a request by having access to an Object Reference for an object and knowing the type of the object and the desired operation to be performed. The client initiates the request by calling stub routines that are specific to the object or by constructing the request dynamically (see Figure 6.3).

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

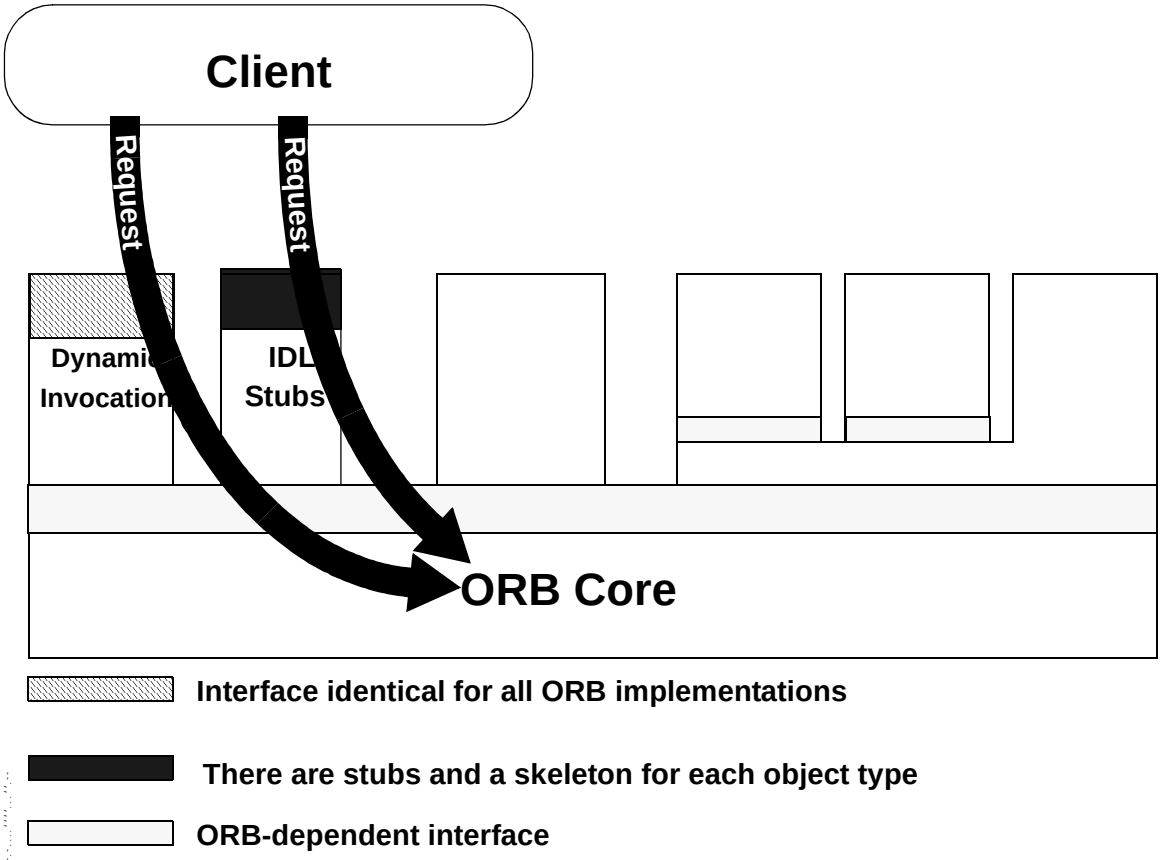


Figure 6.3 - A Client Using the Stub or Dynamic Invocation Interface

The dynamic and stub interface for invoking a request satisfy the same request semantics, and the receiver of the message cannot tell how the request was invoked.

The ORB locates the appropriate implementation code, transmits parameters, and transfers control to the Object Implementation through an IDL skeleton or a dynamic skeleton (see Figure 6.4). Skeletons are specific to the interface and the object adapter. In performing the request, the object implementation may obtain some services from the ORB through the Object Adapter. When the request is complete, control and output values are returned to the client.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

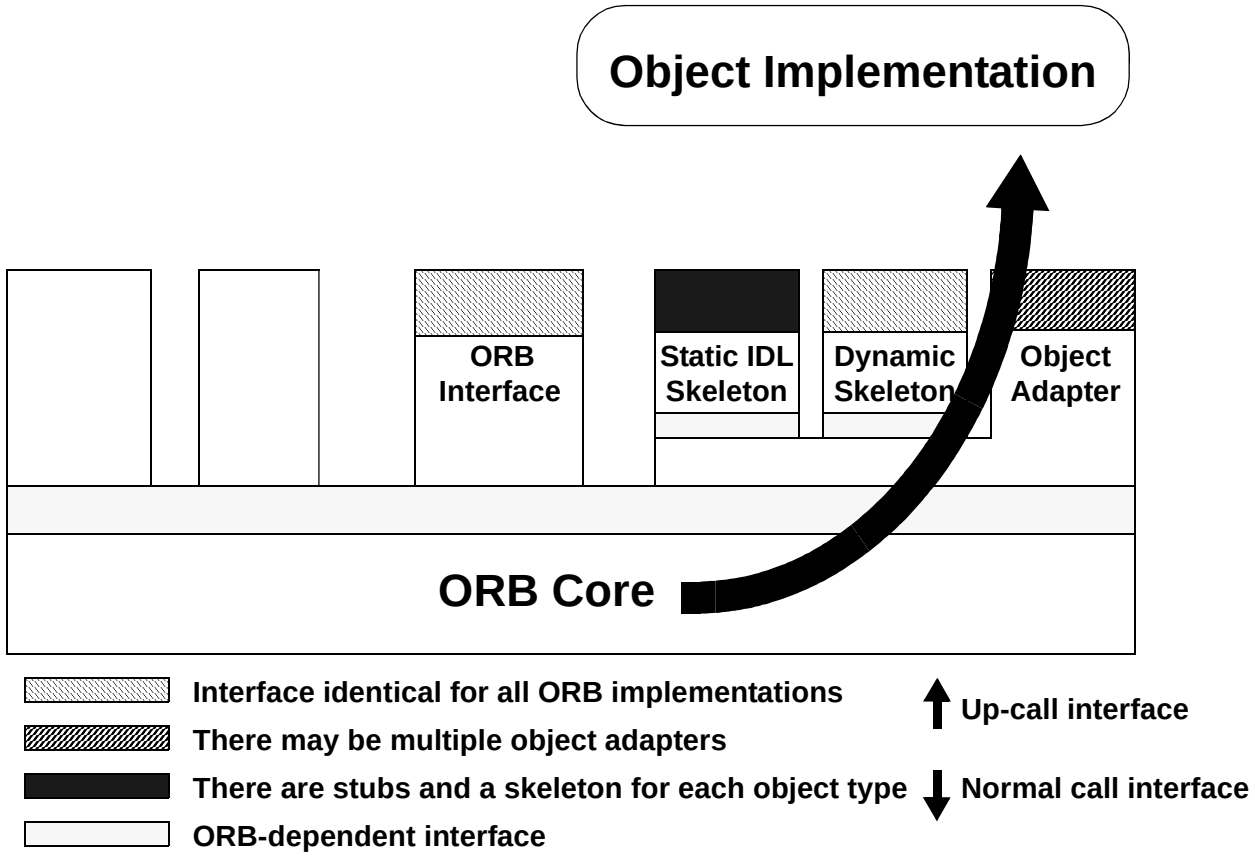


Figure 6.4 - An Object Implementation Receiving a Request

The Object Implementation may choose which Object Adapter to use. This decision is based on what kind of services the Object Implementation requires.

Figure 6.5 shows how interface and implementation information is made available to clients and object implementations. The interface is defined in OMG IDL and/or in the Interface Repository; the definition is used to generate the client Stubs and the object implementation Skeletons.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

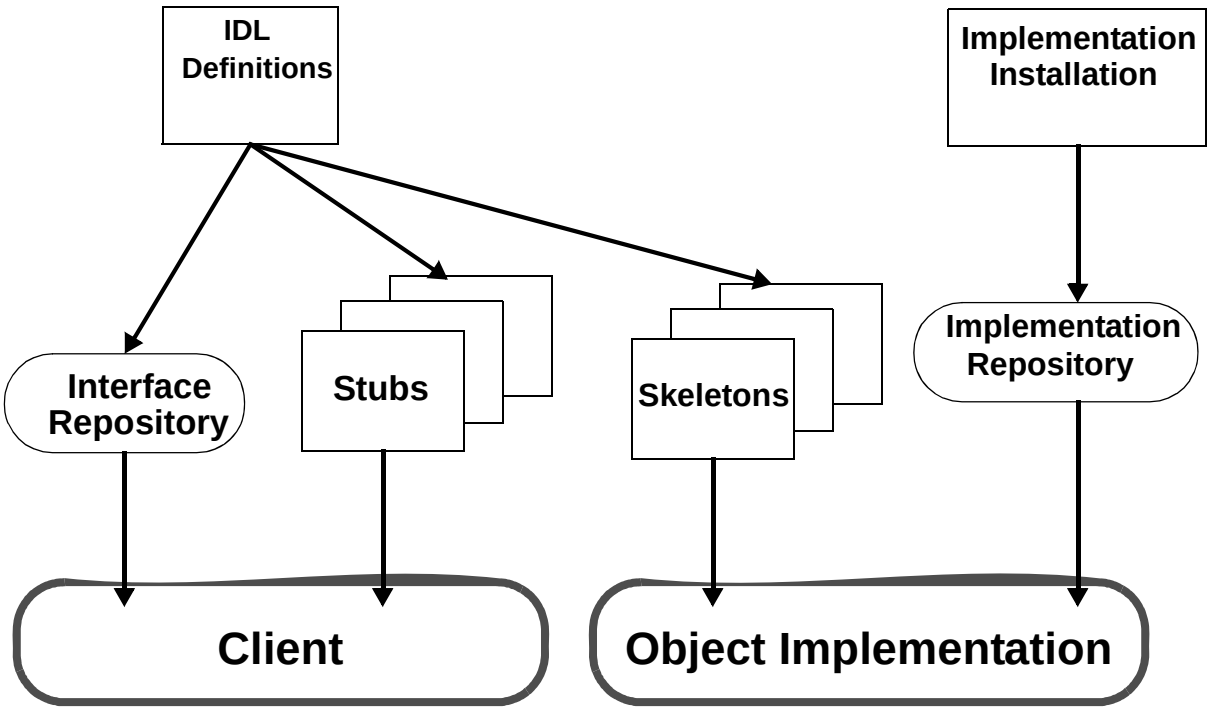


Figure 6.5 - Interface and Implementation Repositories

The object implementation information is provided at installation time and is stored in the Implementation Repository for use during request delivery.

6.1.1 Object Request Broker

In the architecture, the ORB is not required to be implemented as a single component, but rather it is defined by its interfaces. Any ORB implementation that provides the appropriate interface is acceptable. The interface is organized into three categories:

1. Operations that are the same for all ORB implementations.
2. Operations that are specific to particular types of objects.
3. Operations that are specific to particular styles of object implementations.

Different ORBs may make quite different implementation choices, and, together with the IDL compilers, repositories, and various Object Adapters, provide a set of services to clients and implementations of objects that have different properties and qualities.

There may be multiple ORB implementations (also described as multiple ORBs), which have different representations for object references and different means of performing invocations. It may be possible for a client to simultaneously have access to two object references managed by different ORB implementations. When two ORBs are intended to work together, those ORBs must be able to distinguish their object references. It is not the responsibility of the client to do so.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The ORB Core is that part of the ORB that provides the basic representation of objects and communication of requests. CORBA is designed to support different object mechanisms, and it does so by structuring the ORB with components above the ORB Core, which provide interfaces that can mask the differences between ORB Cores.

6.1.2 Clients

A client of an object has access to an object reference for the object, and invokes operations on the object. A client knows only the logical structure of the object according to its interface and experiences the behavior of the object through invocations. Although we will generally consider a client to be a program or process initiating requests on an object, it is important to recognize that something is a client relative to a particular object. For example, the implementation of one object may be a client of other objects.

Clients generally see objects and ORB interfaces through the perspective of a language mapping, bringing the ORB right up to the programmer's level. Clients are maximally portable and should be able to work without source changes on any ORB that supports the desired language mapping with any object instance that implements the desired interface. Clients have no knowledge of the implementation of the object, which object adapter is used by the implementation, or which ORB is used to access it.

6.1.3 Object Implementations

An object implementation provides the semantics of the object, usually by defining data for the object instance and code for the object's methods. Often the implementation will use other objects or additional software to implement the behavior of the object. In some cases, the primary function of the object is to have side-effects on other things that are not objects.

A variety of object implementations can be supported, including separate servers, libraries, a program per method, an encapsulated application, an object-oriented database, etc. Through the use of additional object adapters, it is possible to support virtually any style of object implementation.

Generally, object implementations do not depend on the ORB or how the client invokes the object. Object implementations may select interfaces to ORB-dependent services by the choice of Object Adapter.

6.1.4 Object References

An Object Reference is the information needed to specify an object within an ORB. Both clients and object implementations have an opaque notion of object references according to the language mapping, and thus are insulated from the actual representation of them. Two ORB implementations may differ in their choice of Object Reference representations.

The representation of an object reference handed to a client is only valid for the lifetime of that client.

All ORBs must provide the same language mapping to an object reference (usually referred to as an Object) for a particular programming language. This permits a program written in a particular language to access object references independent of the particular ORB. The language mapping may also provide additional ways to access object references in a typed way for the convenience of the programmer.

There is a distinguished object reference, guaranteed to be different from all object references, that denotes no object.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.5 OMG Interface Definition Language

The OMG Interface Definition Language (OMG IDL) defines the types of objects by specifying their interfaces. An interface consists of a set of named operations and the parameters to those operations. Note that although IDL provides the conceptual framework for describing the objects manipulated by the ORB, it is not necessary for there to be IDL source code available for the ORB to work. As long as the equivalent information is available in the form of stub routines or a run-time interface repository, a particular ORB may be able to function correctly.

IDL is the means by which a particular object implementation tells its potential clients what operations are available and how they should be invoked. From the IDL definitions, it is possible to map CORBA objects into particular programming languages or object systems.

6.1.6 Mapping of OMG IDL to Programming Languages

Different object-oriented or non-object-oriented programming languages may prefer to access CORBA objects in different ways. For object-oriented languages, it may be desirable to see CORBA objects as programming language objects. Even for non-object-oriented languages, it is a good idea to hide the exact ORB representation of the object reference, method names, etc. A particular mapping of OMG IDL to a programming language should be the same for all ORB implementations. Language mapping includes definition of the language-specific data types and procedure interfaces to access objects through the ORB. It includes the structure of the client stub interface (not required for object-oriented languages), the dynamic invocation interface, the implementation skeleton, the object adapters, and the direct ORB interface.

A language mapping also defines the interaction between object invocations and the threads of control in the client or implementation. The most common mappings provide synchronous calls, in that the routine returns when the object operation completes. Additional mappings may be provided to allow a call to be initiated and control returned to the program. In such cases, additional language-specific routines must be provided to synchronize the program's threads of control with the object invocation.

6.1.7 Client Stubs

Generally, the client stubs will present access to the OMG IDL-defined operations on an object in a way that is easy for programmers to predict once they are familiar with OMG IDL and the language mapping for the particular programming language. The stubs make calls on the rest of the ORB using interfaces that are private to, and presumably optimized for, the particular ORB Core. If more than one ORB is available, there may be different stubs corresponding to the different ORBs. In this case, it is necessary for the ORB and language mapping to cooperate to associate the correct stubs with the particular object reference.

6.1.8 Dynamic Invocation Interface

An interface is also available that allows the dynamic construction of object invocations, that is, rather than calling a stub routine that is specific to a particular operation on a particular object, a client may specify the object to be invoked, the operation to be performed, and the set of parameters for the operation through a call or sequence of calls. The client code must supply information about the operation to be performed and the types of the parameters being passed (perhaps obtaining it from an Interface Repository or other run-time source). The nature of the dynamic invocation interface may vary substantially from one programming language mapping to another.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.9 Implementation Skeleton

For a particular language mapping, and possibly depending on the object adapter, there will be an interface to the methods that implement each type of object. The interface will generally be an up-call interface, in that the object implementation writes routines that conform to the interface and the ORB calls them through the skeleton.

The existence of a skeleton does not imply the existence of a corresponding client stub (clients can also make requests via the dynamic invocation interface).

It is possible to write an object adapter that does not use skeletons to invoke implementation methods. For example, it may be possible to create implementations dynamically for languages such as Smalltalk.

6.1.10 Dynamic Skeleton Interface

An interface is available, which allows dynamic handling of object invocations. That is, rather than being accessed through a skeleton that is specific to a particular operation, an object's implementation is reached through an interface that provides access to the operation name and parameters in a manner analogous to the client side's Dynamic Invocation Interface. Purely static knowledge of those parameters may be used, or dynamic knowledge (perhaps determined through an Interface Repository) may also be used, to determine the parameters.

The implementation code must provide descriptions of all the operation parameters to the ORB, and the ORB provides the values of any input parameters for use in performing the operation. The implementation code provides the values of any output parameters, or an exception, to the ORB after performing the operation. The nature of the dynamic skeleton interface may vary substantially from one programming language mapping or object adapter to another, but will typically be an up-call interface.

Dynamic skeletons may be invoked both through client stubs and through the dynamic invocation interface; either style of client request construction interface provides identical results.

6.1.11 Object Adapters

An object adapter is the primary way that an object implementation accesses services provided by the ORB. There are expected to be a few object adapters that will be widely available, with interfaces that are appropriate for specific kinds of objects. Services provided by the ORB through an Object Adapter often include: generation and interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, mapping object references to implementations, and registration of implementations.

The wide range of object granularities, lifetimes, policies, implementation styles, and other properties make it difficult for the ORB Core to provide a single interface that is convenient and efficient for all objects. Thus, through Object Adapters, it is possible for the ORB to target particular groups of object implementations that have similar requirements with interfaces tailored to them.

6.1.12 ORB Interface

The ORB Interface is the interface that goes directly to the ORB, which is the same for all ORBs and does not depend on the object's interface or object adapter. Because most of the functionality of the ORB is provided through the object adapter, stubs, skeleton, or dynamic invocation, there are only a few operations that are common across all objects. These operations are useful to both clients and implementations of objects.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.13 Interface Repository

The Interface Repository is a service that provides persistent objects that represent the IDL information in a form available at run-time. The Interface Repository information may be used by the ORB to perform requests. Moreover, using the information in the Interface Repository, it is possible for a program to encounter an object whose interface was not known when the program was compiled, yet, be able to determine what operations are valid on the object and make an invocation on it.

In addition to its role in the functioning of the ORB, the Interface Repository is a common place to store additional information associated with interfaces to ORB objects. For example, debugging information, libraries of stubs or skeletons, routines that can format or browse particular kinds of objects might be associated with the Interface Repository.

6.1.14 Implementation Repository

The Implementation Repository contains information that allows the ORB to locate and activate implementations of objects. Although most of the information in the Implementation Repository is specific to an ORB or operating environment, the Implementation Repository is the conventional place for recording such information. Ordinarily, installation of implementations and control of policies related to the activation and execution of object implementations is done through operations on the Implementation Repository.

In addition to its role in the functioning of the ORB, the Implementation Repository is a common place to store additional information associated with implementations of ORB objects. For example, debugging information, administrative control, resource allocation, security, etc., might be associated with the Implementation Repository.

6.2 Example ORBs

There are a wide variety of ORB implementations possible within the Common ORB Architecture. This sub clause will illustrate some of the different options. Note that a particular ORB might support multiple options and protocols for communication.

6.2.1 Client- and Implementation-resident ORB

If there is a suitable communication mechanism present, an ORB can be implemented in routines resident in the clients and implementations. The stubs in the client either use a location-transparent IPC mechanism or directly access a location service to establish communication with the implementations. Code linked with the implementation is responsible for setting up appropriate databases for use by clients.

6.2.2 Server-based ORB

To centralize the management of the ORB, all clients and implementations can communicate with one or more servers whose job it is to route requests from clients to implementations. The ORB could be a normal program as far as the underlying operating system is concerned, and normal IPC could be used to communicate with the ORB.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.2.3 System-based ORB

To enhance security, robustness, and performance, the ORB could be provided as a basic service of the underlying operating system. Object references could be made unforgeable, reducing the expense of authentication on each request. Because the operating system could know the location and structure of clients and implementations, it would be possible for a variety of optimizations to be implemented, for example, avoiding marshalling when both are on the same machine.

6.2.4 Library-based ORB

For objects that are light-weight and whose implementations can be shared, the implementation might actually be in a library. In this case, the stubs could be the actual methods. This assumes that it is possible for a client program to get access to the data for the objects and that the implementation trusts the client not to damage the data.

6.3 Structure of a Client

A client of an object has an object reference that refers to that object. An object reference is a token that may be invoked or passed as a parameter to an invocation on a different object. Invocation of an object involves specifying the object to be invoked, the operation to be performed, and parameters to be given to the operation or returned from it.

The ORB manages the control transfer and data transfer to the object implementation and back to the client. In the event that the ORB cannot complete the invocation, an exception response is provided. Ordinarily, a client calls a routine in its program that performs the invocation and returns when the operation is complete.

Clients access object-type-specific stubs as library routines in their program (see Figure 6.6). The client program thus sees routines callable in the normal way in its programming language. All implementations will provide a language-specific data type to use to refer to objects, often an opaque pointer. The client then passes that object reference to the stub routines to initiate an invocation. The stubs have access to the object reference representation and interact with the ORB to perform the invocation. (See the *C Language Mapping* specification for additional, general information on language mapping of object references.)

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

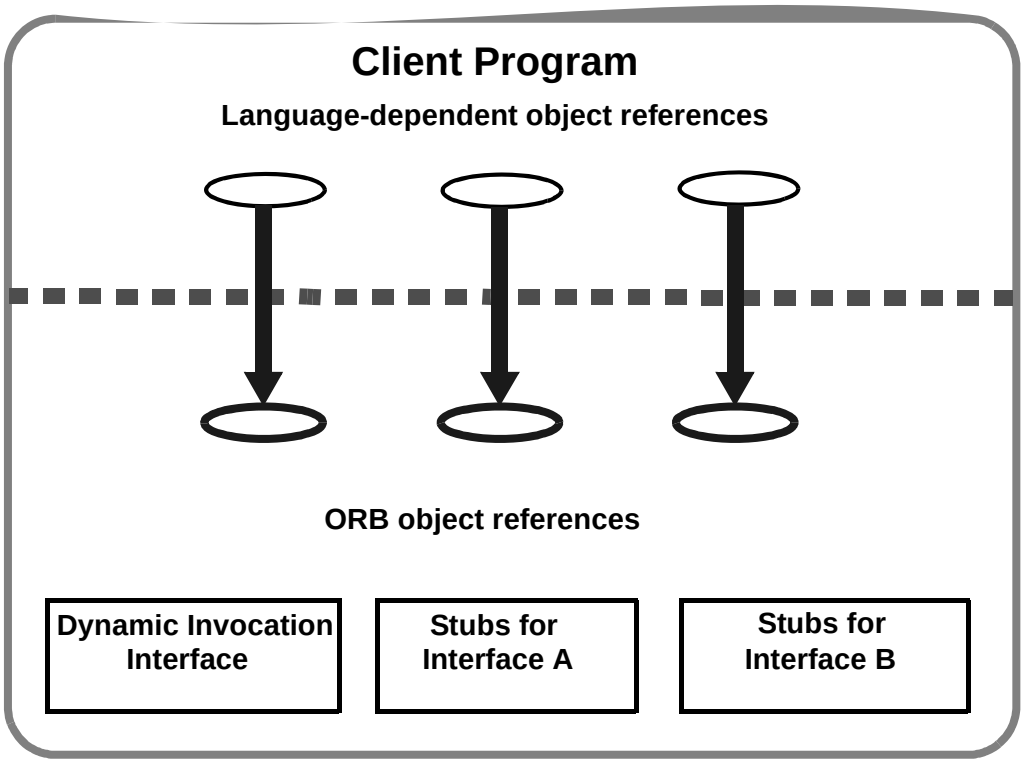


Figure 6.6 - The Structure of a Typical Client

An alternative set of library code is available to perform invocations on objects, for example when the object was not defined at compile time. In that case, the client program provides additional information to name the type of the object and the method being invoked, and performs a sequence of calls to specify the parameters and initiate the invocation.

Clients most commonly obtain object references by receiving them as output parameters from invocations on other objects for which they have references. When a client is also an implementation, it receives object references as input parameters on invocations to objects it implements. An object reference can also be converted to a string that can be stored in files or preserved or communicated by different means and subsequently turned back into an object reference by the ORB that produced the string.

6.4 Structure of an Object Implementation

An object implementation provides the actual state and behavior of an object. The object implementation can be structured in a variety of ways. Besides defining the methods for the operations themselves, an implementation will usually define procedures for activating and deactivating objects and will use other objects or non-object facilities to make the object state persistent, to control access to the object, as well as to implement the methods.

The object implementation (see Figure 6.7) interacts with the ORB in a variety of ways to establish its identity, to create new objects, and to obtain ORB-dependent services. It primarily does this via access to an Object Adapter, which provides an interface to ORB services that is convenient for a particular style of object implementation.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

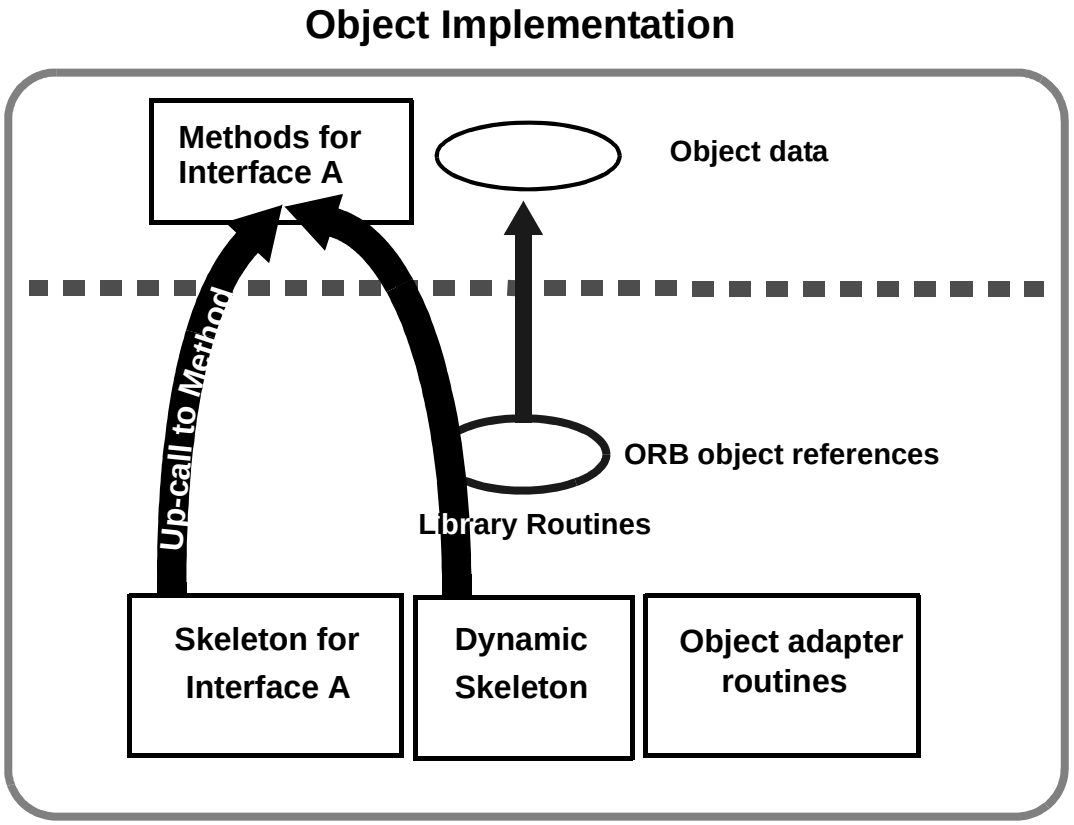


Figure 6.7 - The Structure of a Typical Object Implementation

Because of the range of possible object implementations, it is difficult to be definitive about how an object implementation is structured. See the clauses on the Portable Object Adapter.

When an invocation occurs, the ORB Core, object adapter, and skeleton arrange that a call is made to the appropriate method of the implementation. A parameter to that method specifies the object being invoked, which the method can use to locate the data for the object. Additional parameters are supplied according to the skeleton definition. When the method is complete, it returns, causing output parameters or exception results to be transmitted back to the client.

When a new object is created, the ORB may be notified so that it knows where to find the implementation for that object. Usually, the implementation also registers itself as implementing objects of a particular interface, and specifies how to start up the implementation if it is not already running.

Most object implementations provide their behavior using facilities in addition to the ORB and object adapter. For example, although the Portable Object Adapter provides some persistent data associated with an object (its OID or Object ID), that relatively small amount of data is typically used as an identifier for the actual object data stored in a storage service of the object implementation's choosing. With this structure, it is not only possible for different object implementations to use the same storage service, it is also possible for objects to choose the service that is most appropriate for them.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.5 Structure of an Object Adapter

An object adapter (see Figure 6.8) is the primary means for an object implementation to access ORB services such as object reference generation. An object adapter exports a public interface to the object implementation, and a private interface to the skeleton. It is built on a private ORB-dependent interface.

Object adapters are responsible for the following functions:

- Generation and interpretation of object references
- Method invocation
- Security of interactions
- Object and implementation activation and deactivation
- Mapping object references to the corresponding object implementations
- Registration of implementations

These functions are performed using the ORB Core and any additional components necessary. Often, an object adapter will maintain its own state to accomplish its tasks. It may be possible for a particular object adapter to delegate one or more of its responsibilities to the Core upon which it is constructed.

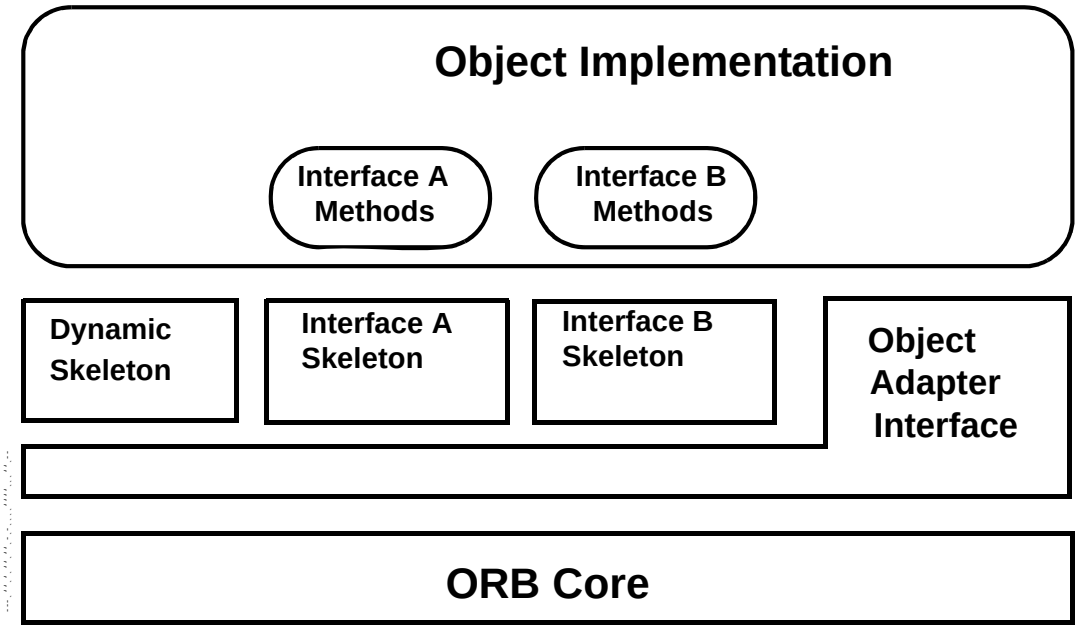


Figure 6.8 - The Structure of a Typical Object Adapter

As shown in Figure 6.8, the Object Adapter is implicitly involved in invocation of the methods, although the direct interface is through the skeletons. For example, the Object Adapter may be involved in activating the implementation or authenticating the request.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The Object Adapter defines most of the services from the ORB that the Object Implementation can depend on. Different ORBs will provide different levels of service and different operating environments may provide some properties implicitly and require others to be added by the Object Adapter. For example, it is common for Object Implementations to want to store certain values in the object reference for easy identification of the object on an invocation. If the Object Adapter allows the implementation to specify such values when a new object is created, it may be able to store them in the object reference for those ORBs that permit it. If the ORB Core does not provide this feature, the Object Adapter would record the value in its own storage and provide it to the implementation on an invocation. With Object Adapters, it is possible for an Object Implementation to have access to a service whether or not it is implemented in the ORB Core — if the ORB Core provides it, the adapter simply provides an interface to it; if not, the adapter must implement it on top of the ORB Core. Every instance of a particular adapter must provide the same interface and service for all the ORBs it is implemented on.

It is also not necessary for all Object Adapters to provide the same interface or functionality. Some Object Implementations have special requirements. For example, an object-oriented database system may wish to implicitly register its many thousands of objects without doing individual calls to the Object Adapter. In such a case, it would be impractical and unnecessary for the object adapter to maintain any per-object state. By using an object adapter interface that is tuned towards such object implementations, it is possible to take advantage of particular ORB Core details to provide the most effective access to the ORB.

6.6 CORBA Required Object Adapter

There are a variety of possible object adapters; however, since the object adapter interface is something that object implementations depend on, it is desirable that there be as few as practical. Most object adapters are designed to cover a range of object implementations, so only when an implementation requires radically different services or interfaces should a new object adapter be considered. In this sub clause, we briefly describe the object adapter defined in this specification.

6.6.1 Portable Object Adapter

This specification defines a Portable Object Adapter that can be used for most ORB objects with conventional implementations. (See the *Portable Object Adapter* clause for more information.) The intent of the POA, as its name suggests, is to provide an Object Adapter that can be used with multiple ORBs with a minimum of rewriting needed to deal with different vendors' implementations.

This specification allows several ways of using servers but it does not deal with the administrative issues of starting server programs. Once started, however, there can be a servant started and ended for a single method call, a separate servant for each object, or a shared servant for all instances of the object type. It allows for groups of objects to be associated by means of being registered with different instances of the POA object and allows implementations to specify their own activation techniques. If the implementation is not active when an invocation is performed, the POA will start one. The POA is specified in IDL, so its mapping to languages is largely automatic, following the language mapping rules. (The primary task left for a language mapping is the definition of the Servant type.)

6.7 The Integration of Foreign Object Systems

The Common ORB Architecture is designed to allow interoperation with a wide range of object systems (see Figure 6.9). Because there are many existing object systems, a common desire will be to allow the objects in those systems to be accessible via the ORB. For those object systems that are ORBs themselves, they may be connected to other ORBs through the mechanisms described throughout this manual.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

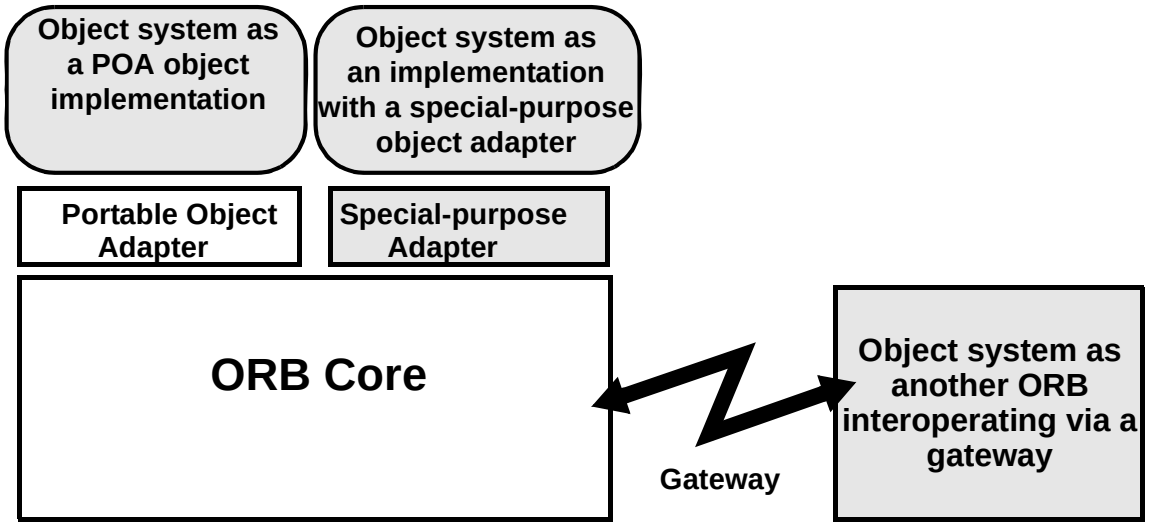


Figure 6.9 - Different Ways to Integrate Foreign Object Systems

For object systems that simply want to map their objects into ORB objects and receive invocations through the ORB, one approach is to have those object systems appear to be implementations of the corresponding ORB objects. The object system would register its objects with the ORB and handle incoming requests, and could act like a client and perform outgoing requests.

In some cases, it will be impractical for another object system to act like a POA object implementation. An object adapter could be designed for objects that are created in conjunction with the ORB and that are primarily invoked through the ORB. Another object system may wish to create objects without consulting the ORB, and might expect most invocations to occur within itself rather than through the ORB. In such a case, a more appropriate object adapter might allow objects to be implicitly registered when they are passed through the ORB.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7 OMG IDL Syntax and Semantics

This clause describes OMG Interface Definition Language (IDL) semantics and gives the syntax for OMG IDL grammatical constructs.

7.1 Overview

The OMG Interface Definition Language (IDL) is the language used to describe the interfaces that client objects call and object implementations provide. An interface definition written in OMG IDL completely defines the interface and fully specifies each operation's parameters. An OMG IDL interface provides the information needed to develop clients that use the interface's operations.

Clients are not written in OMG IDL, which is purely a descriptive language, but in languages for which mappings from OMG IDL concepts have been defined. The mapping of an OMG IDL concept to a client language construct will depend on the facilities available in the client language. For example, an OMG IDL exception might be mapped to a structure in a language that has no notion of exception, or to an exception in a language that does. The binding of OMG IDL concepts to several programming languages is described in this manual.

The description of OMG IDL's lexical conventions is presented in 7.2, Lexical Conventions. A description of OMG IDL preprocessing is presented in 7.3, Preprocessing. The scope rules for identifiers in an OMG IDL specification are described in 7.20, Names and Scoping.

OMG IDL is a declarative language. The grammar is presented in OMG IDL Grammar on page 37 and associated semantics is described in the rest of this clause either in place or through references to other sub clauses of this standard.

OMG IDL-specific pragmas (those not defined for C++) may appear anywhere in a specification; the textual location of these pragmas may be semantically constrained by a particular implementation.

A source file containing interface specifications written in OMG IDL must have a ".idl" extension.

The description of OMG IDL grammar uses a syntax notation that is similar to Extended Backus-Naur Format (EBNF). Table 7.1 lists the symbols used in this format and their meaning.

Table 7.1- IDL EBNF

Symbol	Meaning
::=	Is defined to be
|	Alternatively
<text>	Nonterminal
"text"	Literal
*	The preceding syntactic unit can be repeated zero or more times
+	The preceding syntactic unit can be repeated one or more times
{}	The enclosed syntactic units are grouped as a single syntactic unit
[]	The enclosed syntactic unit is optional—may occur zero or one time

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.2 Lexical Conventions

This sub clause[1] presents the lexical conventions of OMG IDL. It defines tokens in an OMG IDL specification and describes comments, identifiers, keywords, and literals—integer, character, and floating point constants and string literals.

An OMG IDL specification logically consists of one or more files. A file is conceptually translated in several phases.

The first phase is preprocessing, which performs file inclusion and macro substitution. Preprocessing is controlled by directives introduced by lines having # as the first character other than white space. The result of preprocessing is a sequence of tokens. Such a sequence of tokens, that is, a file after preprocessing, is called a translation unit.

OMG IDL uses the ASCII character set, except for string literals and character literals, which use the ISO Latin-1 (8859.1) character set. The ISO Latin-1 character set is divided into alphabetic characters (letters) digits, graphic characters, the space (blank) character, and formatting characters. Table 7.2 shows the ISO Latin-1 alphabetic characters; upper and lower case equivalences are paired. The ASCII alphabetic characters are shown in the left-hand column of Table 7.3.

Table 7.2- Characters

Char.	Description	Char.	Description
Aa	Upper/Lower-case A	Àà	Upper/Lower-case A with grave accent
Bb	Upper/Lower-case B	Áá	Upper/Lower-case A with acute accent
Cc	Upper/Lower-case C	Ââ	Upper/Lower-case A with circumflex accent
Dd	Upper/Lower-case D	Ãã	Upper/Lower-case A with tilde
Ee	Upper/Lower-case E	Ää	Upper/Lower-case A with diaeresis
Ff	Upper/Lower-case F	Åå	Upper/Lower-case A with ring above
Gg	Upper/Lower-case G	Ææ	Upper/Lower-case dipthong A with E
Hh	Upper/Lower-case H	Çç	Upper/Lower-case C with cedilla
Ii	Upper/Lower-case I	Èè	Upper/Lower-case E with grave accent
Jj	Upper/Lower-case J	Éé	Upper/Lower-case E with acute accent
Kk	Upper/Lower-case K	Êê	Upper/Lower-case E with circumflex accent
Ll	Upper/Lower-case L	Ëë	Upper/Lower-case E with diaeresis
Mm	Upper/Lower-case M	Ìì	Upper/Lower-case I with grave accent
Nn	Upper/Lower-case N	Íí	Upper/Lower-case I with acute accent
Oo	Upper/Lower-case O	Îî	Upper/Lower-case I with circumflex accent
Pp	Upper/Lower-case P	Ïï	Upper/Lower-case I with diaeresis

1. This sub clause is an adaptation of *The Annotated C++ Reference Manual,* Clause 2; it differs in the list of legal keywords and punctuation.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 7.2- Characters

Char.	Description	Char.	Description
Qq	Upper/Lower-case Q	Ññ	Upper/Lower-case N with tilde
Rr	Upper/Lower-case R	Òò	Upper/Lower-case O with grave accent
Ss	Upper/Lower-case S	Óó	Upper/Lower-case O with acute accent
Tt	Upper/Lower-case T	Ôô	Upper/Lower-case O with circumflex accent
Uu	Upper/Lower-case U	Õõ	Upper/Lower-case O with tilde
Vv	Upper/Lower-case V	Öö	Upper/Lower-case O with diaeresis
Ww	Upper/Lower-case W	Øø	Upper/Lower-case O with oblique stroke
Xx	Upper/Lower-case X	Ùù	Upper/Lower-case U with grave accent
Yy	Upper/Lower-case Y	Úú	Upper/Lower-case U with acute accent
Zz	Upper/Lower-case Z	Ûû	Upper/Lower-case U with circumflex accent
		Üü	Upper/Lower-case U with diaeresis
		ß	Lower-case German sharp S
		ÿ	Lower-case Y with diaeresis

Table 7.3 lists the decimal digit characters.

Table 7.3- Decimal Digits

0 1 2 3 4 5 6 7 8 9

Table 7.4 shows the graphic characters.

Table 7.4 - Graphic Characters

Character	Description	Character	Description
!	exclamation point	¡	inverted exclamation mark
"	double quote	¢	cent sign
#	number sign	£	pound sign
$	dollar sign	¤	currency sign
%	percent sign	¥	yen sign
&	ampersand		broken bar
'	apostrophe	§	section/paragraph sign
(	left parenthesis	¨	diaeresis

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 7.4 - Graphic Characters

Character	Description	Character	Description
)	right parenthesis	©	copyright sign
*	asterisk	ª	feminine ordinal indicator
+	plus sign	«	left angle quotation mark
,	comma	¬	not sign
-	hyphen, minus sign		soft hyphen
.	period, full stop	®	registered trade mark sign
/	solidus	¯	macron
:	colon	°	ring above, degree sign
;	semicolon	±	plus-minus sign
<	less-than sign	²	superscript two
=	equals sign	³	superscript three
>	greater-than sign	´	acute
?	question mark	m	micro
@	commercial at	¶	pilcrow
[	left square bracket	•	middle dot
\	reverse solidus	¸	cedilla
]	right square bracket	¹	superscript one
^	circumflex	º	masculine ordinal indicator
_	low line, underscore	»	right angle quotation mark
‘	grave		vulgar fraction 1/4
{	left curly bracket		vulgar fraction 1/2
|	vertical line		vulgar fraction 3/4
}	right curly bracket	¿	inverted question mark
~	tilde	¥	multiplication sign
		³	division sign

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The formatting characters are shown in Table 7.5.

Table 7.5 Formatting Characters

Description	Abbreviation	ISO 646 Octal Value
alert	BEL	007
backspace	BS	010
horizontal tab	HT	011
newline	NL, LF	012
vertical tab	VT	013
form feed	FF	014
carriage return	CR	015

7.2.1 Tokens

There are five kinds of tokens: identifiers, keywords, literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collective, “white space”) as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to be the longest string of characters that could possibly constitute a token.

7.2.2 Comments

The characters /* start a comment, which terminates with the characters */. These comments do not nest. The characters / / start a comment, which terminates at the end of the line on which they occur. The comment characters //, /*, and */ have no special meaning within a // comment and are treated just like other characters. Similarly, the comment characters // and /* have no special meaning within a /* comment. Comments may contain alphabetic, digit, graphic, space, horizontal tab, vertical tab, form feed, and newline characters.

7.2.3 Identifiers

An identifier is an arbitrarily long sequence of ASCII alphabetic, digit, and underscore (“_”) characters. The first character must be an ASCII alphabetic character. All characters are significant.

When comparing two identifiers to see if they collide:

- Upper- and lower-case letters are treated as the same letter. Table 7.2 defines the equivalence mapping of upper- and lower-case letters.
- All characters are significant.

Identifiers that differ only in case collide, and will yield a compilation error under certain circumstances. An identifier for a given definition must be spelled identically (e.g., with respect to case) throughout a specification.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

There is only one namespace for OMG IDL identifiers in each scope. Using the same identifier for a constant and an interface, for example, produces a compilation error.

For example:

```
module M {
    typedef long Foo;
    const long thing = 1;
    interface thing {          // error: reuse of identifier
        void doit (
            in Foo foo         // error: Foo and foo collide and refer to different things
        );

readonly attribute long Attribute; // error: Attribute collides with keyword attribute
    };
};
```

7.2.3.1 Escaped Identifiers

As IDL evolves, new keywords that are added to the IDL language may inadvertently collide with identifiers used in existing IDL and programs that use that IDL. Fixing these collisions will require not only the IDL to be modified, but programming language code that depends upon that IDL will have to change as well. The language mapping rules for the renamed IDL identifiers will cause the mapped identifier names (e.g., method names) to be changed.

To minimize the amount of work, users may lexically "escape" identifiers by prepending an underscore (_) to an identifier. This is a purely lexical convention that ONLY turns off keyword checking. The resulting identifier follows all the other rules for identifier processing. For example, the identifier **_AnIdentifier** is treated as if it were **AnIdentifier**.

The following is a non-exclusive list of implications of these rules:

- The underscore does not appear in the Interface Repository.
- The underscore is not used in the DII and DSI.
- The underscore is not transmitted over "the wire."
- Case sensitivity rules are applied to the identifier after stripping off the leading underscore.

For example:

```
module M {
    interface thing {
        attribute boolean abstract;      // error: abstract collides with
                                         //    keyword abstract
        attribute boolean _abstract;     // ok: abstract is an identifier
    };
};
```

To avoid unnecessary confusion for readers of IDL, it is recommended that interfaces only use the escaped form of identifiers when the unescaped form clashes with a newly introduced IDL keyword. It is also recommended that interface designers avoid defining new identifiers that are known to require escaping. Escaped literals are only recommended for IDL that expresses legacy interface, or for IDL that is mechanically generated.

© ISO/IEC 2008 - All rights reserved
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.2.4 Keywords

The identifiers listed in Table 7.6 are reserved for use as keywords and may not be used otherwise, unless escaped with a leading underscore.

Table 7.6 - Keywords

abstract	exception	inout	provides	truncatable
any	emits	interface	public	typedef
attribute	enum	local	publishes	typeid
boolean	eventtype	long	raises	typeprefix
case	factory	module	readonly	unsigned
char	FALSE	multiple	setraises	union
component	finder	native	sequence	uses
const	fixed	Object	short	ValueBase
consumes	float	octet	string	valuetype
context	getraises	oneway	struct	void
custom	home	out	supports	wchar
default	import	primarykey	switch	wstring
double	in	private	TRUE	

Keywords must be written exactly as shown in the above list. Identifiers that collide with keywords (see 7.2.3, Identifiers) are illegal. For example, "**boolean**" is a valid keyword; "**Boolean**" and "**BOOLEAN**" are illegal identifiers.

For example:

```
module M {
    typedef Long Foo;                 // Error: keyword is long not Long
    typedef boolean BOOLEAN;          // Error: BOOLEAN collides with
                                      // the keyword boolean;
};
```

OMG IDL specifications use the characters shown in Table 7.7 as punctuation.

Table 7.7 - Punctuation

;	{	}	:	,	=	+	-	(	)	<	>	[	]
'	"	\	|	^	&	*	/	%	~				

In addition, the tokens listed in Table 7.8 are used by the preprocessor.

Table 7.8 - Tokens

#	##	!	||	&&

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.2.5 Literals

This sub clause describes the following literals:

- Integer
- Character
- Floating-point
- String
- Fixed-point

7.2.5.1 Integer Literals

An integer literal consisting of a sequence of digits is taken to be decimal (base ten) unless it begins with 0 (digit zero). A sequence of digits starting with 0 is taken to be an octal integer (base eight). The digits 8 and 9 are not octal digits. A sequence of digits preceded by 0x or 0X is taken to be a hexadecimal integer (base sixteen). The hexadecimal digits include a or A through f or F with decimal values ten through fifteen, respectively. For example, the number twelve can be written 12, 014, or 0XC.

7.2.5.2 Character Literals

A character literal is one or more characters enclosed in single quotes, as in 'x.' Character literals have type **char**.

A character is an 8-bit quantity with a numerical value between 0 and 255 (decimal). The value of a space, alphabetic, digit, or graphic character literal is the numerical value of the character as defined in the ISO Latin-1 (8859.1) character set standard (See Table 7.2 on page 28, Table 7.3 on page 29, and Table 7.4 on page 29). The value of a null is 0. The value of a formatting character literal is the numerical value of the character as defined in the ISO 646 standard (see Table 7.5 on page 31). The meaning of all other characters is implementation-dependent.

Nongraphic characters must be represented using escape sequences as defined below in Table 7.9. Note that escape sequences must be used to represent single quote and backslash characters in character literals.

Table 7.9 - Escape Sequences

Description	Escape Sequence
newline	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form feed	\f
alert	\a
backslash	\\
question mark	\?

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 7.9 - Escape Sequences

Description	Escape Sequence
single quote	\'
double quote	\"
octal number	\ooo
hexadecimal number	\xhh
unicode character	\uhhhh

If the character following a backslash is not one of those specified, the behavior is undefined. An escape sequence specifies a single character.

The escape \ooo consists of the backslash followed by one, two, or three octal digits that are taken to specify the value of the desired character. The escape \xhh consists of the backslash followed by x followed by one or two hexadecimal digits that are taken to specify the value of the desired character.

The escape \uhhhh consists of a backslash followed by the character 'u,' followed by one, two, three, or four hexadecimal digits. This represents a unicode character literal. Thus the literal "\u002E" represents the unicode period '.' character and the literal "\u3BC" represents the unicode greek small letter 'mu.' The \u escape is valid only with wchar and wstring types. Because a wide string literal is defined as a sequence of wide character literals a sequence of \u literals can be used to define a wide string literal. Attempts to set a char type to a \u defined literal or a string type to a sequence of \u literals result in an error.

A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. The value of a character constant is implementation dependent if it exceeds that of the largest char.

Wide character literals have an **L** prefix, for example:

```
const wchar C1 = L'X';
```

Attempts to assign a wide character literal to a non-wide character constant or to assign a non-wide character literal to a wide character constant result in a compile-time diagnostic.

Both wide and non-wide character literals must be specified using characters from the ISO 8859-1 character set.

7.2.5.3 Floating-point Literals

A floating-point literal consists of an integer part, a decimal point, a fraction part, an e or E, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of decimal (base ten) digits. Either the integer part or the fraction part (but not both) may be missing; either the decimal point or the letter e (or E) and the exponent (but not both) may be missing.

7.2.5.4 String Literals

A string literal is a sequence of characters (as defined in 7.2.5.2, Character Literals), with the exception of the character with numeric value 0, surrounded by double quotes, as in "...".

Adjacent string literals are concatenated. Characters in concatenated strings are kept distinct. For example,

```
"\xA" "B"
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

contains the two characters ‘\xA’ and ‘B’ after concatenation (and not the single hexadecimal character ‘\xAB’).

The size of a string literal is the number of character literals enclosed by the quotes, after concatenation. Within a string, the double quote character **"** must be preceded by a \.

A string literal may not contain the character ‘\0’.

Wide string literals have an L prefix, for example:

```
const wstring S1 = L"Hello";
```

Attempts to assign a wide string literal to a non-wide string constant or to assign a non-wide string literal to a wide string constant result in a compile-time diagnostic.

Both wide and non-wide string literals must be specified using characters from the ISO 8859-1 character set.

A wide string literal shall not contain the wide character with value zero.

7.2.5.5 Fixed-Point Literals

A fixed-point decimal literal consists of an integer part, a decimal point, a fraction part and a d or D. The integer and fraction parts both consist of a sequence of decimal (base 10) digits. Either the integer part or the fraction part (but not both) may be missing; the decimal point (but not the letter d (or D)) may be missing.

7.3 Preprocessing

OMG IDL is preprocessed according to the specification of the preprocessor in “International Organization for Standardization. 1998. ISO/IEC 14882 Standard for the C++ Programming Language. Geneva: International Organization for Standardization.” The preprocessor may be implemented as a separate process or built into the IDL compiler.

Lines beginning with # (also called “directives”) communicate with this preprocessor. White space may appear before the #. These lines have syntax independent of the rest of OMG IDL; they may appear anywhere and have effects that last (independent of the OMG IDL scoping rules) until the end of the translation unit. The textual location of OMG IDL-specific pragmas may be semantically constrained.

A preprocessing directive (or any line) may be continued on the next line in a source file by placing a backslash character (“\”), immediately before the newline at the end of the line to be continued. The preprocessor effects the continuation by deleting the backslash and the newline before the input sequence is divided into tokens. A backslash character may not be the last character in a source file.

A preprocessing token is an OMG IDL token (see 7.2.1, Tokens), a file name as in a **#include** directive, or any single character other than white space that does not match another preprocessing token.

The primary use of the preprocessing facilities is to include definitions from other OMG IDL specifications. Text in files included with a #include directive is treated as if it appeared in the including file, except that **RepositoryId** related pragmas are handled in a special way. The special handling of these pragmas is described in 14.7, RepositoryIds.

Note that whether a particular IDL compiler generates code for included files is an implementation-specific issue. To support separate compilation, IDL compilers may not generate code for included files, or do so only if explicitly instructed.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.4 OMG IDL Grammar

(1) <specification>::=<import>* <definition>+

(2) <definition>::=<type_dcl> ";"
| <const_dcl> ";"
| <except_dcl> ";"
| <interface> ";"
| <module> ";"
| <value> ";"
| <type_id_dcl> ";"
| <type_prefix_dcl> ";"
| <event> ";"
| <component> ";"
| <home_dcl> ";"

(3) <module>::="module" <identifier> "{" <definition>+ "}"

(4) <interface>::=<interface_dcl>
| <forward_dcl>

(5) <interface_dcl>::=<interface_header> "{" <interface_body> "}"

(6) <forward_dcl>::=["abstract" | "local"] "interface" <identifier>

(7) <interface_header>::=["abstract" | "local"] "interface" <identifier>
[<interface_inheritance_spec>]

(8) <interface_body>::=<export>*

(9) <export>::=<type_dcl> ";"
| <const_dcl> ";"
| <except_dcl> ";"
| <attr_dcl> ";"
| <op_dcl> ";"
| <type_id_dcl> ";"
| <type_prefix_dcl> ";"

(10) <interface_inheritance_spec>::=":" <interface_name>
{ "," <interface_name> }*

(11) <interface_name>::=<scoped_name>

(12) <scoped_name>::=<identifier>
| "::" <identifier>
| <scoped_name> "::" <identifier>

(13) <value> ::= (<value_dcl> | <value_abs_dcl> | <value_box_dcl> | <value_forward_dcl>)

(14) <value_forward_dcl> ::=["abstract"] "valuetype" <identifier>

(15) <value_box_dcl> ::="valuetype" <identifier> <type_spec>

(16) <value_abs_dcl> ::="abstract" "valuetype" <identifier>
[<value_inheritance_spec>]
"{" <export>* "}"

(17) <value_dcl> ::=<value_header> "{" < value_element>* "}"

(18) <value_header> ::=["custom"] "valuetype" <identifier>
[<value_inheritance_spec>]

(19) <value_inheritance_spec> ::=[":" ["truncatable"] <value_name>
{ "," <value_name> }*]

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
                [ “supports” <interface_name>
                { “,” <interface_name> }*  ]
(20)  <value_name> ::=<scoped_name>
(21)  <value_element> ::=<export> | < state_member> | <init_dcl>
(22)  <state_member> ::=( “public” | “private” )
                <type_spec> <declarators> “;”
(23)  <init_dcl> ::=“factory” <identifier>
                “(“  [ <init_param_decls> ]  “)”
                [ <raises_expr> ] “;”
(24)  <init_param_decls> ::=<init_param_decl> { “,” <init_param_decl> }*
(25)  <init_param_decl> ::=<init_param_attribute> <param_type_spec> <simple_declarator>
(26)  <init_param_attribute> ::=“in”
(27)  <const_dcl>::=“const” <const_type>
                <identifier> “=” <const_exp>
(28)  <const_type>::=<integer_type>
                                | <char_type>
                                | <wide_char_type>
                                | <boolean_type>
                                | <floating_pt_type>
                                | <string_type>
                                | <wide_string_type>
                                | <fixed_pt_const_type>
                                | <scoped_name>
                                | <octet_type>
(29)  <const_exp>::=<or_expr>
(30)  <or_expr>::=<xor_expr>
                                | <or_expr> “|” <xor_expr>
(31)  <xor_expr>::=<and_expr>
                                | <xor_expr> “^” <and_expr>
(32)  <and_expr>::=<shift_expr>
                                | <and_expr> “&” <shift_expr>
(33)  <shift_expr>::=<add_expr>
                                | <shift_expr> “>>” <add_expr>
                                | <shift_expr> “<<” <add_expr>
(34)  <add_expr>::=<mult_expr>
                                | <add_expr> “+” <mult_expr>
                                | <add_expr> “-” <mult_expr>
(35)  <mult_expr>::=<unary_expr>
                                | <mult_expr> “*” <unary_expr>
                                | <mult_expr> “/” <unary_expr>
                                | <mult_expr> “%” <unary_expr>
(36)  <unary_expr>::=<unary_operator> <primary_expr>
                                | <primary_expr>
(37)  <unary_operator>::=“-”
                                | “+”
                                | “~”
(38)  <primary_expr>::=<scoped_name>
                                | <literal>
```


© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

| “(” <const_exp> “)”

(39) <literal>::=<integer_literal>
| <string_literal>
| <wide_string_literal>
| <character_literal>
| <wide_character_literal>
| <fixed_pt_literal>
| <floating_pt_literal>
| <boolean_literal>

(40) <boolean_literal>::=“TRUE”
| “FALSE”

(41) <positive_int_const>::=<const_exp>

(42) <type_dcl>::=“typedef” <type_declarator>
| <struct_type>
| <union_type>
| <enum_type>
| “native” <simple_declarator>
| <constr_forward_decl>

(43) <type_declarator>::=<type_spec> <declarators>

(44) <type_spec>::=<simple_type_spec>
| <constr_type_spec>

(45) <simple_type_spec>::=<base_type_spec>
| <template_type_spec>
| <scoped_name>

(46) <base_type_spec>::=<floating_pt_type>
| <integer_type>
| <char_type>
| <wide_char_type>
| <boolean_type>
| <octet_type>
| <any_type>
| <object_type>
| <value_base_type>

(47) <template_type_spec>::=<sequence_type>
| <string_type>
| <wide_string_type>
| <fixed_pt_type>

(48) <constr_type_spec>::=<struct_type>
| <union_type>
| <enum_type>

(49) <declarators>::=<declarator> { “,” <declarator> }*

(50) <declarator>::=<simple_declarator>
| <complex_declarator>

(51) <simple_declarator>::=<identifier>

(52) <complex_declarator>::=<array_declarator>

(53) <floating_pt_type>::=“float”
| “double”
| “long” “double”

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(54) <integer_type>::=<signed_int>
| <unsigned_int>
(55) <signed_int>::=<signed_short_int>
| <signed_long_int>
| <signed_longlong_int>
(56) <signed_short_int>::="short"
(57) <signed_long_int>::="long"
(58) <signed_longlong_int>::="long" "long"
(59) <unsigned_int>::=<unsigned_short_int>
| <unsigned_long_int>
| <unsigned_longlong_int>
(60) <unsigned_short_int>::="unsigned" "short"
(61) <unsigned_long_int>::="unsigned" "long"
(62) <unsigned_longlong_int>::="unsigned" "long" "long"
(63) <char_type>::="char"
(64) <wide_char_type>::="wchar"
(65) <boolean_type>::="boolean"
(66) <octet_type>::="octet"
(67) <any_type>::="any"
(68) <object_type>::="Object"
(69) <struct_type>::="struct" <identifier> "{" <member_list> "}"
(70) <member_list>::=<member>$^{+}$
(71) <member>::=<type_spec> <declarators> ";"
(72) <union_type>::="union" <identifier> "switch"
"(" <switch_type_spec> ")"
"{" <switch_body> "}"
(73) <switch_type_spec>::=<integer_type>
| <char_type>
| <boolean_type>
| <enum_type>
| <scoped_name>
(74) <switch_body>::=<case>$^{+}$
(75) <case>::=<case_label>$^{+}$ <element_spec> ";"
(76) <case_label>::="case" <const_exp> ":"
| "default" ":"
(77) <element_spec>::=<type_spec> <declarator>
(78) <enum_type>::="enum" <identifier>
"{" <enumerator> { "," <enumerator> }* "}"
(79) <enumerator>::=<identifier>
(80) <sequence_type>::="sequence" "<" <simple_type_spec> "," <positive_int_const> ">"
| "sequence" "<" <simple_type_spec> ">"
(81) <string_type>::="string" "<" <positive_int_const> ">"
| "string"
(82) <wide_string_type>::="wstring" "<" <positive_int_const> ">"
| "wstring"
(83) <array_declarator>::=<identifier> <fixed_array_size>$^{+}$

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(84) <fixed_array_size>::=“[” <positive_int_const> “]”

(85) <attr_dcl> ::= <readonly_attr_spec>
| <attr_spec>

(86) <except_dcl>::=“exception” <identifier> “{“ <member>* “}”

(87) <op_dcl>::=[<op_attribute>] <op_type_spec>
<identifier> <parameter_dcls>
[<raises_expr>] [<context_expr>]

(88) <op_attribute>::=“oneway”

(89) <op_type_spec>::=<param_type_spec>
| “void”

(90) <parameter_dcls>::=“(” <param_dcl> { “,” <param_dcl> }* “)”
| “(” “)”

(91) <param_dcl>::=<param_attribute> <param_type_spec> <simple_declarator>

(92) <param_attribute>::=“in”
| “out”
| “inout”

(93) <raises_expr>::=“raises” “(” <scoped_name>
{ “,” <scoped_name> }* “)”

(94) <context_expr>::=“context” “(” <string_literal>
{ “,” <string_literal> }* “)”

(95) <param_type_spec>::=<base_type_spec>
| <string_type>
| <wide_string_type>
| <scoped_name>

(96) <fixed_pt_type>::=“fixed” “<“ <positive_int_const> “,” <positive_int_const> “>”

(97) <fixed_pt_const_type>::=“fixed”

(98) <value_base_type>::= “ValueBase”

(99) <constr_forward_decl>::=“struct” <identifier>
| “union” <identifier>

(100) <import> ::= “import” <imported_scope> “;”

(101) <imported_scope> ::= <scoped_name> | <string_literal>

(102) <type_id_dcl> ::=“typeid” <scoped_name> <string_literal>

(103) <type_prefix_dcl>::=“typeprefix” <scoped_name> <string_literal>

(104) <readonly_attr_spec> ::= “readonly” “attribute” <param_type_spec>
<readonly_attr_declarator>

(105) <readonly_attr_declarator>::= <simple_declarator> <raises_expr>
| <simple_declarator>
{ “,” <simple_declarator> }*

(106) <attr_spec> ::= “attribute” <param_type_spec>
<attr_declarator>

(107) <attr_declarator> ::=<simple_declarator> <attr_raises_expr>
| <simple_declarator>
{ “,” <simple_declarator> }*

(108) <attr_raises_expr> ::=<get_excep_expr> [<set_excep_expr>]
| <set_excep_expr>

(109) <get_excep_expr> ::= “getraises” <exception_list>

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(110) <set_excep_expr> ::= “setraises” <exception_list>

(111) <exception_list>::= “(” <scoped_name>
{ “,” <scoped_name> }* “)”

NOTE: Grammar rules 1 through 111 with the exception of the last three lines of rule 2 constitutes the portion of IDL that is not related to components.

(112) <component> ::=<component_dcl>
| <component_forward_dcl>

(113) <component_forward_dcl> ::= “component” <identifier>

(114) <component_dcl> ::= <component_header>
“{” <component_body> “}”

(115) <component_header> ::= “component” <identifier>
[<component_inheritance_spec>]
[<supported_interface_spec>]

(116) <supported_interface_spec> ::= “supports” <scoped_name>
{ “,” <scoped_name> }*

(117) <component_inheritance_spec> ::= “:” <scoped_name>

(118) <component_body> ::=<component_export>*

(119) <component_export> ::=<provides_dcl> “;”
| <uses_dcl> “;”
| <emits_dcl> “;”
| <publishes_dcl> “;”
| <consumes_dcl> “;”
| <attr_dcl> “;”

(120) <provides_dcl> ::= “provides” <interface_type> <identifier>

(121) <interface_type> ::= <scoped_name>
| “Object”

(122) <uses_dcl> ::= “uses” [“multiple”]
< interface_type> <identifier>

(123) <emits_dcl> ::= “emits” <scoped_name> <identifier>

(124) <publishes_dcl> ::= “publishes” <scoped_name> <identifier>

(125) <consumes_dcl> ::= “consumes” <scoped_name> <identifier>

(126) <home_dcl> ::= <home_header> <home_body>

(127) <home_header> ::= “home” <identifier>
[<home_inheritance_spec>]
[<supported_interface_spec>]
“manages” <scoped_name>
[<primary_key_spec>]

(128) <home_inheritance_spec> ::= “:” <scoped_name>

(129) <primary_key_spec> ::= “primarykey” <scoped_name>

(130) <home_body> ::= “{” <home_export>* “}”

(131) <home_export ::= <export>
| <factory_dcl> “;”
| <finder_dcl> “;”

(132) <factory_dcl> ::= “factory” <identifier>
“(“ [<init_param_decls>] “)”
[<raises_expr>]

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(133) <finder_dcl> ::= “finder” <identifier>
“(“ [<init_param_decls>] “)”
[<raises_expr>]
(134) <event> ::= (<event_dcl> | <event_abs_dcl> |
<event_forward_dcl>)
(135) <event_forward_dcl> ::=[“abstract”] “eventtype” <identifier>
(136) <event_abs_dcl> ::=“abstract” “eventtype” <identifie
[<value_inheritance_spec>]
“{” <export>* “}”
(137) <event_dcl>::=<event_header> “{” <value_element> * “}”
(138) <event_header>::=[“custom”] “eventtype”
<identifier> [<value_inheritance_spec>]

7.5 OMG IDL Specification

An OMG IDL specification consists of one or more type definitions, constant definitions, exception definitions, or module definitions. The syntax is:

(1)<specification>::=<import>* <definition>+
(2) <definition>::=<type_dcl> “;”
| <const_dcl> “;”
| <except_dcl> “;”
| <interface> “;”
| <module> “;”
| <value> “;”
| <type_id_dcl> “;”
| <type_prefix_dcl> “;”
| <event> “;”
| <component> “;”
| <home_dcl> “;”

See Import Declaration on page 44, for the specification of <import>.

See Module Declaration on page 45, for the specification of <module>.

See Interface Declaration on page 45, for the specification of <interface>.

See Value Declaration on page 50, for the specification of <value>.

See Constant Declaration on page 55, Type Declaration on page 59, and Exception Declaration on page 71 respectively for specifications of **<const_dcl>**, **<type_dcl>**, and **<except_dcl>**.

See Repository Identity Related Declarations on page 75, for specification of Repository Identity declarations which include <type_id_dcl> and <type_prefix_dcl>.

See Event Declaration on page 77, for specification of <event>.

See Component Declaration on page 79, for specification of <component>.

See Section 7.18, <$paratext>, on page 84, for specification of <home_dcl>.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.6 Import Declaration

The grammar for the import statement is described by the following Backus Naur Form (BNF):

(100)<import> ::= "import" <imported_scope> ";"
(101) <imported_scope> ::= <scoped_name> | <string_literal>

The **<imported_scope>** non-terminal may be either a fully-qualified scoped name denoting an IDL name scope, or a string containing the interface repository ID of an IDL name scope, i.e., a definition object in the repository whose interface derives from **CORBA::Container**.

The definition of import obviates the need to define the meaning of IDL constructs in terms of "file scopes." This specification defines the concepts of a specification as a unit of IDL expression. In the abstract, a specification consists of a finite sequence of ISO Latin-1 characters that form a legal IDL sentence. The physical representation of the specification is of no consequence to the definition of IDL, though it is generally associated with a file in practice.

Any scoped name that begins with the scope token ("**::**") is resolved relative to the global scope of the specification in which it is defined. In isolation, the scope token represents the scope of the specification in which it occurs.

A specification that imports name scopes must be interpreted in the context of a well-defined set of IDL specifications whose union constitutes the space from within which name scopes are imported. By "a well-defined set of IDL specifications," we mean any identifiable representation of IDL specifications, such as an interface repository. The specific representation from which name scopes are imported is not specified, nor is the means by which importing is implemented, nor is the means by which a particular set of IDL specifications (such as an interface repository) is associated with the context in which the importing specification is to be interpreted.

The effects of an import statement are as follows:

- The contents of the specified name scope are visible in the context of the importing specification. Names that occur in IDL declarations within the importing specification may be resolved to definitions in imported scopes.
- Imported IDL name scopes exist in the same space as names defined in subsequent declarations in the importing specification.
- IDL module definitions may re-open modules defined in imported name scopes.
- Importing an inner name scope (i.e., a name scope nested within one or more enclosing name scopes) does not implicitly import the contents of any of the enclosing name scopes.
- When a name scope is imported, the names of the enclosing scopes in the fully-qualified pathname of the enclosing scope are *exposed* within the context of the importing specification, but their contents are not imported. An importing specification may not redefine or reopen a name scope that has been exposed (but not imported) by an import statement.
- Importing a name scope recursively imports all name scopes nested within it.
- For the purposes of this specification, name scopes that can be imported (i.e., specified in an import statement) include the following: **modules**, **interfaces**, **valuetypes**, and **eventtypes**.
- Redundant imports (e.g., importing an inner scope and one of its enclosing scopes in the same specification) are disregarded. The union of all imported scopes is visible to the importing program.
- This specification does not define a particular form for generated stubs and skeletons in any given programming language. In particular, it does not imply any normative relationship between units specification and units of generation and/or compilation for any language mapping.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.7 Module Declaration

A module definition satisfies the following syntax:

(3)<module>::=“module” <identifier> “{“ <definition>+ “}”

The module construct is used to scope OMG IDL identifiers; see CORBA Module on page 86 for details.

7.8 Interface Declaration

An interface definition satisfies the following syntax:

(4) <interface>::=<interface_dcl>
| <forward_dcl>
(5) <interface_dcl>::=<interface_header> “{” <interface_body> “}”
(6) <forward_dcl>::=[“abstract” | “local”] “interface” <identifier>
(7) <interface_header>::=[“abstract” | “local”] “interface” <identifier>
[<interface_inheritance_spec>]
(8) <interface_body>::=<export>*
(9) <export>::=<type_dcl> “;”
| <const_dcl> “;”
| <except_dcl> “;”
| <attr_dcl> “;”
| <op_dcl> “;”
| <type_id_decl> “;”
| <type_prefix_decl> “;”

7.8.1 Interface Header

The interface header consists of three elements:

1. An optional modifier specifying if the interface is an abstract interface.
2. The interface name. The name must be preceded by the keyword ***interface***, and consists of an identifier that names the interface.
3. An optional inheritance specification. The inheritance specification is described in the next sub clause.

The **<identifier>** that names an interface defines a legal type name. Such a type name may be used anywhere an **<identifier>** is legal in the grammar, subject to semantic constraints as described in the following sub clauses. Since one can only hold references to an object, the meaning of a parameter or structure member, which is an interface type is as a *reference* to an object supporting that interface. Each language binding describes how the programmer must represent such interface references.

Abstract interfaces have slightly different rules and semantics from “regular” interfaces, as described in Abstract Interface on page 49. They also follow different language mapping rules.

Local interfaces have slightly different rules and semantics from “regular” interfaces, as described in Local Interface on page 50. They also follow different language mapping rules.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.8.2 Interface Inheritance Specification

The syntax for inheritance is as follows:

(10) <interface_inheritance_spec>::=":" <interface_name>
{ "," <interface_name> }*
(11) <interface_name>::=<scoped_name>
(12) <scoped_name>::=<identifier>
| "::" <identifier>
| <scoped_name> "::" <identifier>

Each **<scoped_name>** in an **<interface_inheritance_spec>** must be the name of a previously defined interface or an alias to a previously defined interface. See Interface Inheritance on page 47 for the description of inheritance.

7.8.3 Interface Body

The interface body contains the following kinds of declarations:

- Constant declarations, which specify the constants that the interface exports. Constant declaration syntax is described in Constant Declaration on page 55.
- Type declarations, which specify the type definitions that the interface exports. Type declaration syntax is described in Type Declaration on page 59.
- Exception declarations, which specify the exception structures that the interface exports. Exception declaration syntax is described in Exception Declaration on page 71.
- Attribute declarations, which specify the associated attributes exported by the interface. Attribute declaration syntax is described in Attribute Declaration on page 74.
- Operation declarations, which specify the operations that the interface exports and the format of each, including operation name, the type of data returned, the types of all parameters of an operation, legal exceptions that may be returned as a result of an invocation, and contextual information that may affect method dispatch. Operation declaration syntax is described in Operation Declaration on page 71.

Empty interfaces are permitted (that is, those containing no declarations).

Some implementations may require interface-specific pragmas to precede the interface body.

7.8.4 Forward Declaration

A forward declaration declares the name of an interface without defining it. This permits the definition of interfaces that refer to each other. The syntax is: optionally either the keyword **abstract** or the keyword **local**, followed by the keyword **interface**, followed by an <identifier> that names the interface.

Multiple forward declarations of the same interface name are legal.

It is illegal to inherit from a forward-declared interface whose definition has not yet been seen:

```
module Example {
    interface base;             // Forward declaration

    // ...
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    interface derived : base {};      // Error
    interface base {};                // Define base
    interface derived : base {};      // OK
};
```

7.8.5 Interface Inheritance

An interface can be derived from another interface, which is then called a *base* interface of the derived interface. A derived interface, like all interfaces, may declare new elements (constants, types, attributes, exceptions, and operations). In addition, unless redefined in the derived interface, the elements of a base interface can be referred to as if they were elements of the derived interface. The name resolution operator ("::") may be used to refer to a base element explicitly; this permits reference to a name that has been redefined in the derived interface.

A derived interface may redefine any of the type, constant, and exception names that have been inherited; the scope rules for such names are described in Names and Scoping on page 87.

An interface is called a direct base if it is mentioned in the **<interface_inheritance_spec>** and an indirect base if it is not a direct base but is a base interface of one of the interfaces mentioned in the **<interface_inheritance_spec>**.

An interface may be derived from any number of base interfaces. Such use of more than one direct base interface is often called multiple inheritance. The order of derivation is not significant.

An abstract interface may only inherit from other abstract interfaces.

An interface may not be specified as a direct base interface of a derived interface more than once; it may be an indirect base interface more than once. Consider the following example:

```
interface A { ... }
interface B: A { ... }
interface C: A { ... }
interface D: B, C { ... }
interface E: A, B { ... };                    // OK
```

The relationships between these interfaces is shown in Figure 7.1. This "diamond" shape is legal, as is the definition of E on the right.

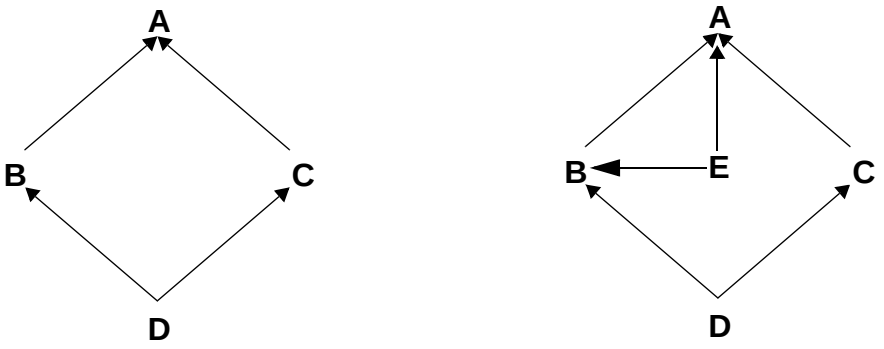


Figure 7.1 - Legal Multiple Inheritance Example

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

References to base interface elements must be unambiguous. A Reference to a base interface element is ambiguous if the name is declared as a constant, type, or exception in more than one base interface. Ambiguities can be resolved by qualifying a name with its interface name (that is, using a **<scoped_name>**). It is illegal to inherit from two interfaces with the same operation or attribute name, or to redefine an operation or attribute name in the derived interface.

So for example in:

```
interface A {
    typedef long L1;
    short opA(in L1 l_1);
};

interface B {
    typedef short L1;
    L1 opB(in long l);
};

interface C: B, A {
    typedef L1 L2;          // Error: L1 ambiguous
    typedef A::L1 L3;       // A::L1 is OK
    B::L1 opC(in L3 l_3);   // all OK no ambiguities
};
```

References to constants, types, and exceptions are bound to an interface when it is defined (i.e., replaced with the equivalent global **<scoped_name>**s). This guarantees that the syntax and semantics of an interface are not changed when the interface is a base interface for a derived interface. Consider the following example:

```
const long L = 3;

interface A {
    typedef float coord[L]:
    void f (in coord s);          // s has three floats
};

interface B {
    const long L = 4;
};

interface C: B, A { };                    // what is C::f()'s signature?
```

The early binding of constants, types, and exceptions at interface definition guarantees that the signature of operation **f** in interface **C** is

```
typedef float coord[3];
void f (in coord s);
```

which is identical to that in interface **A**. This rule also prevents redefinition of a constant, type, or exception in the derived interface from affecting the operations and attributes inherited from a base interface.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Interface inheritance causes all identifiers defined in base interfaces, both direct and indirect, to be visible in the current naming scope. A type name, constant name, enumeration value name, or exception name from an enclosing scope can be redefined in the current scope. An attempt to use an ambiguous name without qualification produces a compilation error. Thus in:

```
interface A {
    typedef string<128> string_t;
};

interface B {
    typedef string<256> string_t;
};

interface C: A, B {
    attribute string_t      Title;      // Error: string_t ambiguous
    attribute A::string_t   Name;       // OK
    attribute B::string_t   City;       // OK
};
```

operation and attribute names are used at run-time by both the stub and dynamic interfaces. As a result, all operations and attributes that might apply to a particular object must have unique names. This requirement prohibits redefining an operation or attribute name in a derived interface, as well as inheriting two operations or attributes with the same name.

```
interface A {
    void make_it_so();
};

interface B: A {
    short make_it_so(in long times);   // Error: redefinition of make_it_so
};
```

For a complete summary of allowable inheritance and supporting relationships among interfaces and valuetypes see Table 7.10 on page 55.

7.8.6 Abstract Interface

An interface declaration containing the keyword **abstract** in its header, declares an abstract interface. The following special rules apply to abstract interfaces:

- Abstract interfaces may only inherit from other abstract interfaces.
- Value types may support any number of abstract interfaces.

See Semantics of Abstract Interfaces on page 173 for CORBA implementation semantics associated with abstract interfaces.

For a complete summary of allowable inheritance and supporting relationships among interfaces and valuetypes see Table 7.10 on page 55.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.8.7 Local Interface

An interface declaration containing the keyword **local** in its header declares a local interface. An interface declaration not containing the keyword **local** is referred to as an unconstrained interface. An object implementing a local interface is referred to as a local object. The following special rules apply to local interfaces:

- A local interface may inherit from other local or unconstrained interfaces.
- An unconstrained interface may not inherit from a local interface. An interface derived from a local interface must be explicitly declared local.
- A valuetype may support a local interface.
- Any IDL type, including an unconstrained interface, may appear as a parameter, attribute, return type, or exception declaration of a local interface.
- A local interface is a local type, as is any non-interface type declaration constructed using a local interface or other local type. For example, a struct, union, or exception with a member that is a local interface is also itself a local type.
- A local type may be used as a parameter, attribute, return type, or exception declaration of a local interface or of a valuetype.
- A local type may not appear as a parameter, attribute, return type, or exception declaration of an unconstrained interface.

For a complete summary of allowable inheritance and supporting relationships among interfaces and valuetypes see Table 7.10 on page 55.

See LocalObject Operations on page 113 for CORBA implementation semantics associated with local objects.

7.9 Value Declaration

There are several kinds of value type declarations: “regular” value types, boxed value types, abstract value types, and forward declarations.

A value declaration satisfies the following syntax:

(13) <value> ::= (<value_dcl> | <value_abs_dcl> | <value_box_dcl> | <value_forward_dcl>)

7.9.1 Regular Value Type

A regular value type satisfies the following syntax:

(17)<value_dcl> ::=<value_header> “{“ < value_element>* “}”
(18) <value_header> ::=[“custom”] “valuetype” <identifier>
[<value_inheritance_spec>]
(21)<value_element> ::=<export>
| < state_member> |
| <init_dcl>

7.9.1.1 Value Header

The value header consists of two elements:

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

1. The value type's name and optional modifier specifying whether the value type uses custom marshaling.
2. An optional value inheritance specification. The value inheritance specification is described below.

7.9.1.2 Value Element

A value can contain all the elements that an interface can as well as the definition of state members, and initializers for that state.

7.9.1.3 Value Inheritance Specification

(19)<value_inheritance_spec> ::=[":" ["truncatable"] <value_name>
{ "," <value_name> }*]
["supports" <interface_name>
{ "," <interface_name> }*]

(20) <value_name> ::=<scoped_name>

Each **<value_name>** in a **<value_inheritance_spec>** must be the name of a previously defined value type or an alias to a previously defined value type. Each **<interface_name>** in a **<value_inheritance_spec>** must be the name of a previously defined interface or an alias to a previously defined interface. See Valuetype Inheritance on page 54 for the description of value type inheritance.

The **truncatable** modifier may not be used if the value type being defined is a custom value.

A valuetype that supports a local interface does not itself become *local* (i.e., unmarshalable) as a result of that support.

7.9.1.4 State Members

(22)<state_member> ::=("public" | "private")
<type_spec> <declarators> ";"

Each **<state_member>** defines an element of the state, which is marshaled and sent to the receiver when the value type is passed as a parameter. A state member is either public or private. The annotation directs the language mapping to hide or expose the different parts of the state to the clients of the value type. The private part of the state is only accessible to the implementation code and the marshaling routines.

A valuetype that has a state member that is *local* (i.e., non-marshalable like a local interface), is itself rendered *local*. That is, such valuetypes behave similar to local interfaces when an attempt is made to marshal them.

Note that certain programming languages may not have the built in facilities needed to distinguish between the public and private members. In these cases, the language mapping specifies the rules that programmers are responsible for following.

7.9.1.5 Initializers

(23) <init_dcl> ::="factory" <identifier>
"(" [<init_param_decls>] ")"
[<raises_expr>] ";"

(24) <init_param_decls> ::=<init_param_decl> { "," <init_param_decl> }*

(25) <init_param_decl> ::=<init_param_attribute> <param_type_spec> <simple_declarator>

(26) <init_param_attribute> ::="in"

In order to ensure portability of value implementations, designers may also define the signatures of initializers (or constructors) for non-abstract value types. Syntactically these look like local operation signatures except that they are prefixed with the keyword **factory**, have no return type, and must use only **in** parameters. There may be any number of

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

factory declarations. The names of the initializers are part of the name scope of the value type. Initializers defined in a valuetype are not inherited by derived valuetypes, and hence the names of the initializers are free to be reused in a derived valuetype.

If no initializers are specified in IDL, the value type does not provide a portable way of creating a runtime instance of its type. There is no default initializer. This allows the definition of IDL value types, which are not intended to be directly instantiated by client code.

7.9.1.6 Value Type Example

```
interface Tree {
    void print()
};

valuetype WeightedBinaryTree {
    // state definition
        private unsigned long weight;
        private WeightedBinaryTree left;
        private WeightedBinaryTree right;
    // initializer
        factory init(in unsigned long w);
    // local operations
        WeightSeq pre_order();
        WeightSeq post_order();
};
valuetype WTree: WeightedBinaryTree supports Tree {};
```

7.9.2 Boxed Value Type

(15)<value_box_dcl> ::=“valuetype” <identifier> <type_spec>

It is often convenient to define a value type with no inheritance or operations and with a single state member. A shorthand IDL notation is used to simplify the use of value types for this kind of simple containment, referred to as a “value box.”

Since a value box of a valuetype adds no additional properties to a valuetype, it is an error to box valuetypes.

Value box is particularly useful for strings and sequences. Basically one does not have to create what is in effect an additional namespace that will contain only one name.

An example is the following IDL:

```
module Example {
    interface Foo {
        ... /* anything */
    };
    valuetype FooSeq sequence<Foo>;
    interface Bar {
        void doIt (in FooSeq seq1);
    };
};
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The above IDL provides similar functionality to writing the following IDL. However the type identities (repository IDs) would be different.

```
module Example {
    interface Foo {
        ... /* anything */
    };
    valuetype FooSeq {
        public sequence<Foo> data;
    };
    interface Bar {
        void doIt (in FooSeq seq);
    };
};
```

The former is easier to manipulate after it is mapped to a concrete programming language.

Any IDL type may be used to declare a value box except for a valuetype.

The declaration of a boxed value type does not open a new scope. Thus a construction such as

```
valuetype FooSeq sequence <FooSeq>;
```

is not legal IDL. The identifier being declared as a boxed value type cannot be used subsequent to its initial use and prior to the completion of the boxed value declaration.

7.9.3 Abstract Value Type

(16) <value_abs_dcl> ::=“abstract” “valuetype” <identifier>
[<value_inheritance_spec>]
“{“ <export>* “}”

Value types may also be abstract. They are called abstract because an abstract value type may not be instantiated. No **<state_member>** or **<initializers>** may be specified. However, local operations may be specified. Essentially they are a bundle of operation signatures with a purely local implementation.

Note that a concrete value type with an empty state is not an abstract value type.

7.9.4 Value Forward Declaration

(14)<value_forward_dcl> ::=[“abstract”] “valuetype” <identifier>

A forward declaration declares the name of a value type without defining it. This permits the definition of value types that refer to each other. The syntax consists simply of the keyword **valuetype** followed by an **<identifier>** that names the value type.

Multiple forward declarations of the same value type name are legal.

Boxed value types cannot be forward declared; such a forward declaration would refer to a normal value type.

It is illegal to inherit from a forward-declared value type whose definition has not yet been seen.

It is illegal for a value type to support a forward-declared interface whose definition has not yet been seen.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.9.5 Valuetype Inheritance

The terminology that is used to describe value type inheritance is directly analogous to that used to describe interface inheritance (see Interface Inheritance on page 47).

The name scoping and name collision rules for valuetypes are identical to those for interfaces. In addition, no valuetype may be specified as a direct abstract base of a derived valuetype more than once; it may be an indirect abstract base more than once. See Interface Inheritance on page 47 for a detailed description of the analogous properties for interfaces.

Values may be derived from other values and can support an interface and any number of abstract interfaces.

Once implementation (state) is specified at a particular point in the inheritance hierarchy, all derived value types (which must of course implement the state) may only derive from a single (concrete) value type. They can however derive from other additional abstract values and support an additional interface.

The single immediate base concrete value type, if present, must be the first element specified in the inheritance list of the value declaration's IDL. It may be followed by other abstract values from which it inherits. The interface and abstract interfaces that it supports are listed following the **supports** keyword.

While a valuetype may only directly support one interface, it is possible for the valuetype to support other interfaces as well through inheritance. In this case, the supported interface must be derived, directly or indirectly, from each interface that the valuetype supports through inheritance. This rule does not apply to abstract interfaces that the valuetype supports. For example:

```
interface I1 { };
interface I2 { };
interface I3: I1, I2 { };

abstract valuetype V1 supports I1 { };
abstract valuetype V2 supports I2 { };
valuetype V3: V1, V2 supports I3 { }; // legal
valuetype V4: V1 supports I2 { }; // illegal
```

A stateful value that derives from another stateful value may specify that it is **truncatable**. This means that it is to "truncate" (see Value instance -> Value type on page 160) an instance to be an instance of any of its truncatable parent (stateful) value types under certain conditions. Note that all the intervening types in the inheritance hierarchy must be truncatable in order for truncation to a particular type to be allowed.

Because custom values require an exact type match between the sending and receiving context, **truncatable** may not be specified for a custom value type.

Non-custom value types may not (transitively) inherit from custom value types.

Boxed value types may not be derived from, nor may they derive from, anything else.

These rules are summarized in the following table.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 7.10

May inherit from:	Interface	Abstract Interface	Abstract Value	Stateful Value	Boxed value
Interface	multiple	multiple	no	no	no
Abstract Interface	no	multiple	no	no	no
Abstract Value	supports single	supports multiple	multiple	no	no
Stateful Value	supports single	supports multiple	multiple	single (may be truncatable)	no
Boxed Value	no	no	no	no	no

7.10 Constant Declaration

This sub clause describes the syntax for constant declarations.

7.10.1 Syntax

The syntax for a constant declaration is:

(27)<const_dcl>::=“const” <const_type>
<identifier> “=” <const_exp>
(28) <const_type>::=<integer_type>
| <char_type>
| <wide_char_type>
| <boolean_type>
| <floating_pt_type>
| <string_type>
| <wide_string_type>
| <fixed_pt_const_type>
| <scoped_name>
| <octet_type>
(29) <const_exp>::=<or_expr>
(30) <or_expr>::=<xor_expr>
| <or_expr> “|” <xor_expr>
(31) <xor_expr>::=<and_expr>
| <xor_expr> “^” <and_expr>
(32) <and_expr>::=<shift_expr>
| <and_expr> “&” <shift_expr>
(33) <shift_expr>::=<add_expr>
| <shift_expr> “>>” <add_expr>

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

| <shift_expr> “<<” <add_expr>
(34) <add_expr>::=<mult_expr>
| <add_expr> “+” <mult_expr>
| <add_expr> “-” <mult_expr>
(35) <mult_expr>::=<unary_expr>
| <mult_expr> “*” <unary_expr>
| <mult_expr> “/” <unary_expr>
| <mult_expr> “%” <unary_expr>
(36) <unary_expr>::=<unary_operator> <primary_expr>
| <primary_expr>
(37) <unary_operator>::=“-”
| “+”
| “~”
(38) <primary_expr>::=<scoped_name>
| <literal>
| “(” <const_exp> “)”
(39) <literal>::=<integer_literal>
| <string_literal>
| <wide_string_literal>
| <character_literal>
| <wide_character_literal>
| <fixed_pt_literal>
| <floating_pt_literal>
| <boolean_literal>
(40) <boolean_literal>::=“TRUE”
| “FALSE”
(41) <positive_int_const>::=<const_exp>

7.10.2 Semantics

The **<scoped_name>** in the **<const_type>** production must be a previously defined name of an **<integer_type>**, **<char_type>**, **<wide_char_type>**, **<boolean_type>**, **<floating_pt_type>**, **<string_type>, <wide_string_type>**, **<octet_type>**, or **<enum_type>** constant.

Octet literals have integer value in the range 0..255. If the right hand side of an **octet** constant declaration is outside this range it shall be flagged as a compile time error.

Integer literals have positive integer values. Constant integer literals are considered **unsigned long** unless the value is too large, then they are considered **unsigned long long**. Unary minus is considered an operator, not a part of an integer literal. Only integer values can be assigned to integer type (**short**, **long**, **long long**) constants, and **octet** constants. Only positive integer values can be assigned to unsigned integer type constants. If the value of the right hand side of an integer constant declaration is too large to fit in the actual type of the constant on the left hand side, for example

const short s = 655592;

or is inappropriate for the actual type of the left hand side, for example

const octet o = -54;

it shall be flagged as a compile time error.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Floating point literals have floating point values. Only floating point values can be assigned to floating point type (**float**, **double**, **long double**) constants. Constant floating point literals are considered **double** unless the value is too large, then they are considered **long double**. If the value of the right hand side is too large to fit in the actual type of the constant to which it is being assigned, it shall be flagged as a compile time error. Truncation on the right for floating point types is OK.

Fixed point literals have fixed point values. Only fixed point values can be assigned to fixed point type constants. If the fixed point value in the expression on the right hand side is too large to fit in the actual fixed point type of the constant on the left hand side, then it shall be flagged as a compile time error. Truncation on the right for fixed point types is OK.

If the type of an integer constant is **long** or **unsigned long**, then each subexpression of the associated constant expression is treated as an **unsigned long** by default, or a signed **long** for negated literals or negative integer constants. It is an error if any subexpression values exceed the precision of the assigned type (**long** or **unsigned long**), or if a final expression value (of type **unsigned long**) exceeds the precision of the target type (**long**).

If the type of an integer constant is **long long** or **unsigned long long**, then each subexpression of the associated constant expression is treated as an **unsigned long long** by default, or a signed **long long** for negated literals or negative integer constants. It is an error if any subexpression values exceed the precision of the assigned type (**long long** or **unsigned long long**), or if a final expression value (of type **unsigned long long**) exceeds the precision of the target type (**long long**).

If the type of a floating-point constant is **double**, then each subexpression of the associated constant expression is treated as a **double.** It is an error if any subexpression value exceeds the precision of **double**.

If the type of a floating-point constant is **long double**, then each subexpression of the associated constant expression is treated as a **long double**. It is an error if any subexpression value exceeds the precision of **long double**.

An infix operator can combine two integer types, floating point types or fixed point types, but not mixtures of these. Infix operators are applicable only to integer, floating point, and fixed point types.

Integer expressions are evaluated using the imputed type of each argument of a binary operator in turn. If either argument is **unsigned long long**, use **unsigned long long**. If either argument is **long long**, use **long long**. If either argument is **unsigned long**., use **unsigned long**. Otherwise use **long**. The final result of an integer arithmetic expression must fit in the range of the declared type of the constant, otherwise an error shall be flagged by the compiler. In addition to the integer types, the final result of an integer arithmetic expression can be assigned to an **octet** constant, subject to it fitting in the range for **octet** type.

Floating point expressions are evaluated using the imputed type of each argument of a binary operator in turn. If either argument is **long double**, use **long double**. Otherwise use **double**. The final result of a floating point arithmetic expression must fit in the range of the declared type of the constant, otherwise an error shall be flagged by the compiler.

Fixed-point decimal constant expressions are evaluated as follows. A fixed-point literal has the apparent number of total and fractional digits. For example, **0123.450d** is considered to be **fixed<7,3>** and **3000.00d** is **fixed<6,2>**. Prefix operators do not affect the precision; a prefix **+** is optional, and does not change the result. The upper bounds on the number of digits and scale of the result of an infix expression, **fixed<d1,s1> op fixed<d2,s2>**, are shown in the following table.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Op	Result: fixed<d,s>
+	fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>
-	fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>
*	fixed<d1+d2, s1+s2>
/	fixed<(d1-s1+s2) + sinf, sinf>

A quotient may have an arbitrary number of decimal places, denoted by a scale of s_{inf}. The computation proceeds pairwise, with the usual rules for left-to-right association, operator precedence, and parentheses. All intermediate computations shall be performed using double precision (i.e., 62 digit) arithmetic. If an individual computation between a pair of fixed-point literals actually generates more than 31 significant digits, then a 31-digit result is retained as follows:

fixed<d,s> => fixed<31, 31-d+s>

Leading and trailing zeros are not considered significant. The omitted digits are discarded; rounding is not performed. The result of the individual computation then proceeds as one literal operand of the next pair of fixed-point literals to be computed.

Unary (**+ -**) and binary (*** / + -**) operators are applicable in floating-point and fixed-point expressions. Unary (**+ - ~**) and binary (*** / % + - << >> & | ^**) operators are applicable in integer expressions.

The "~" unary operator indicates that the bit-complement of the expression to which it is applied should be generated. For the purposes of such expressions, the values are 2's complement numbers. As such, the complement can be generated as follows:

Integer Constant Expression Type	Generated 2's Complement Numbers
long	long -(value+1)
unsigned long	unsigned long (2**32-1) - value
long long	long long -(value+1)
unsigned long long	unsigned long (2**64-1) - value

The "%" binary operator yields the remainder from the division of the first expression by the second. If the second operand of "%" is 0, the result is undefined; otherwise

(a/b)*b + a%b

is equal to a. If both operands are non-negative, then the remainder is non-negative; if not, the sign of the remainder is implementation dependent.

The "<<" binary operator indicates that the value of the left operand should be shifted left the number of bits specified by the right operand, with 0 fill for the vacated bits. The right operand must be in the range 0 <= right operand < 64.

The ">>" binary operator indicates that the value of the left operand should be shifted right the number of bits specified by the right operand, with 0 fill for the vacated bits. The right operand must be in the range 0 <= right operand < 64.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The “&” binary operator indicates that the logical, bitwise AND of the left and right operands should be generated.

The “|” binary operator indicates that the logical, bitwise OR of the left and right operands should be generated.

The “^” binary operator indicates that the logical, bitwise EXCLUSIVE-OR of the left and right operands should be generated.

<positive_int_const> must evaluate to a positive integer constant.

An octet constant can be defined using an integer literal or an integer constant expression, for example:

Values for an octet constant outside the range 0 - 255 shall cause a compile-time error.

An enum constant can only be defined using a scoped name for the enumerator. The scoped name is resolved using the normal scope resolution rules 7.20, Names and Scoping. For example:

```
enum Color { red, green, blue };
const Color FAVORITE_COLOR = red;

module M {
    enum Size { small, medium, large };
};
const M::Size MYSIZE = M::medium;
```

The constant name for the RHS of an enumerated constant definition must denote one of the enumerators defined for the enumerated type of the constant. For example:

```
const Color col = red;    // is OK but
const Color another = M::medium; // is an error
```

7.11 Type Declaration

OMG IDL provides constructs for naming data types; that is, it provides C language-like declarations that associate an identifier with a type. OMG IDL uses the **typedef** keyword to associate a name with a data type. A name is also associated with a data type via the **struct**, **union**, **enum**, and **native** declarations. The syntax is:

```
(42)  <type_dcl>::=“typedef” <type_declarator>
                          |  <struct_type>
                          |  <union_type>
                          |  <enum_type>
                          |  “native” <simple_declarator>
                          |  <constr_forward_decl>
(43)  <type_declarator>::=<type_spec> <declarators>
```

For type declarations, OMG IDL defines a set of type specifiers to represent typed values. The syntax is as follows:

```
(44)  <type_spec>::=<simple_type_spec>
                          |  <constr_type_spec>
(45)  <simple_type_spec>::=<base_type_spec>
                          |  <template_type_spec>
                          |  <scoped_name>
(46)  <base_type_spec>::=<floating_pt_type>
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

| <integer_type>
| <char_type>
| <wide_char_type>
| <boolean_type>
| <octet_type>
| <any_type>
| <object_type>
| <value_base_type>

(47) <template_type_spec>::=<sequence_type>
| <string_type>
| <wide_string_type>
| <fixed_pt_type>

(48) <constr_type_spec>::=<struct_type>
| <union_type>
| <enum_type>

(49) <declarators>::=<declarator> { “,” <declarator> }*

(50) <declarator>::=<simple_declarator>
| <complex_declarator>

(51) <simple_declarator>::=<identifier>

(52) <complex_declarator>::=<array_declarator>

The **<scoped_name>** in **<simple_type_spec>** must be a previously defined type introduced by an interface declaration (**<interface_dcl>** - see 7.8, Interface Declaration), a value declaration (**<value_dcl>**, **<value_box_dcl>** or **<abstract_value_dcl>** - see 7.9, Value Declaration) or a type declaration (**<type_dcl>** - see 7.11, Type Declaration). Note that exceptions are not considered types in this context.

As seen above, OMG IDL type specifiers consist of scalar data types and type constructors. OMG IDL type specifiers can be used in operation declarations to assign data types to operation parameters. The next sub clauses describe basic and constructed type specifiers.

7.11.1 Basic Types

The syntax for the supported basic types is as follows:

(53) <floating_pt_type>::=“float”
| “double”
| “long” “double”

(54) <integer_type>::=<signed_int>
| <unsigned_int>

(55) <signed_int>::=<signed_short_int>
| <signed_long_int>
| <signed_longlong_int>

(56) <signed_short_int>::=“short”

(57) <signed_long_int>::=“long”

(58) <signed_longlong_int>::=“long” “long”

(59) <unsigned_int>::=<unsigned_short_int>
| <unsigned_long_int>
| <unsigned_longlong_int>

(60) <unsigned_short_int>::=“unsigned” “short”

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(61) <unsigned_long_int>::=“unsigned” “long”
(62) <unsigned_longlong_int>::=“unsigned” “long” “long”
(63) <char_type>::=“char”
(64) <wide_char_type>::=“wchar”
(65) <boolean_type>::=“boolean”
(66) <octet_type>::=“octet”
(67) <any_type>::=“any”

Each OMG IDL data type is mapped to a native data type via the appropriate language mapping. Conversion errors between OMG IDL data types and the native types to which they are mapped can occur during the performance of an operation invocation. The invocation mechanism (client stub, dynamic invocation engine, and skeletons) may signal an exception condition to the client if an attempt is made to convert an illegal value. The standard system exceptions that are to be raised in such situations are defined in 8.12, Exceptions.

7.11.1.1 Integer Types

OMG IDL integer types are **short**, **unsigned short**, **long**, **unsigned long**, **long long**, and **unsigned long long** representing integer values in the range indicated below in Table 7.11.

Table 7.11

short	-2^{15} .. $2^{15} - 1$
long	-2^{31} .. $2^{31} - 1$
long long	-2^{63} .. $2^{63} - 1$
unsigned short	0 .. $2^{16} - 1$
unsigned long	0 .. $2^{32} - 1$
unsigned long long	0 .. $2^{64} - 1$

7.11.1.2 Floating-Point Types

OMG IDL floating-point types are **float**, **double**, and **long double**. The **float** type represents IEEE single-precision floating point numbers; the **double** type represents IEEE double-precision floating point numbers. The **long double** data type represents an IEEE double-extended floating-point number, which has an exponent of at least 15 bits in length and a signed fraction of at least 64 bits. See *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985, for a detailed specification.

7.11.1.3 Char Type

OMG IDL defines a **char** data type that is an 8-bit quantity that (1) encodes a single-byte character from any byte-oriented code set, or (2) when used in an array, encodes a multi-byte character from a multi-byte code set. In other words, an implementation is free to use any code set internally for encoding character data, though conversion to another form may be required for transmission.

The ISO 8859-1 (Latin1) character set standard defines the meaning and representation of all possible graphic characters used in OMG IDL (i.e., the space, alphabetic, digit, and graphic characters defined in Table 7.2 on page 28, Table 7.3 on page 29, and Table 7.4 on page 29). The meaning and representation of the null and formatting characters (see Table 7.5 on page 31) is the numerical value of the character as defined in the ASCII (ISO 646) standard. The meaning of all other characters is implementation-dependent.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

During transmission, characters may be converted to other appropriate forms as required by a particular language binding. Such conversions may change the representation of a character but maintain the character's meaning. For example, a character may be converted to and from the appropriate representation in international character sets.

7.11.1.4 Wide Char Type

OMG IDL defines a **wchar** data type that encodes wide characters from any character set. As with character data, an implementation is free to use any code set internally for encoding wide characters, though, again, conversion to another form may be required for transmission. The size of **wchar** is implementation-dependent.

7.11.1.5 Boolean Type

The **boolean** data type is used to denote a data item that can only take one of the values **TRUE** and **FALSE**.

7.11.1.6 Octet Type

The **octet** type is an 8-bit quantity that is guaranteed not to undergo any conversion when transmitted by the communication system.

7.11.1.7 Any Type

The **any** type permits the specification of values that can express any OMG IDL type.

An **any** logically contains a **TypeCode** (see 8.11, TypeCodes) and a value that is described by the **TypeCode**. Each IDL language mapping provides operations that allow programers to insert and access the **TypeCode** and value contained in an any.

7.11.2 Constructed Types

Structs, **unions**, and **enums** are the constructed types. Their syntax is presented below:

(42)<type_dcl>::="typedef" <type_declarator>
| <struct_type>
| <union_type>
| <enum_type>
| "native" <simple_declarator>
| <constr_forward_decl>
(48)<constr_type_spec>::=<struct_type>
| <union_type>
| <enum_type>
(99)<constr_forward_decl>::="struct" <identifier>
| "union" <identifier>

7.11.2.1 Structures

The syntax for **struct** type is:

(69)<struct_type>::="struct" <identifier> "{" <member_list> "}"
(70) <member_list>::=<member>$^{+}$
(71) <member>::=<type_spec> <declarators> ";"

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The **<identifier>** in **<struct_type>** defines a new legal type. Structure types may also be named using a **typedef** declaration.

Name scoping rules require that the member declarators in a particular structure be unique. The value of a **struct** is the value of all of its members.

7.11.2.2 Discriminated Unions

The discriminated **union** syntax is:

(72) <union_type>::=“union” <identifier> “switch”
“(” <switch_type_spec> “)”
“{” <switch_body> “}”
(73) <switch_type_spec>::=<integer_type>
| <char_type>
| <boolean_type>
| <enum_type>
| <scoped_name>
(74) <switch_body>::=<case>+
(75) <case>::=<case_label>+ <element_spec> “;”
(76) <case_label>::=“case” <const_exp> “:”
| “default” “:”
(77) <element_spec>::=<type_spec> <declarator>

OMG IDL unions are a cross between the C **`union`** and **`switch`** statements. IDL unions must be discriminated; that is, the union header must specify a typed tag field that determines which union member to use for the current instance of a call. The **<identifier>** following the **union** keyword defines a new legal type. Union types may also be named using a **typedef** declaration. The **<const_exp>** in a **<case_label>** must be consistent with the **<switch_type_spec>**. A **default** case can appear at most once. The **<scoped_name>** in the **<switch_type_spec>** production must be a previously defined **integer**, **char**, **boolean**, or **enum** type.

Case labels must match or be automatically castable to the defined type of the discriminator. Name scoping rules require that the element declarators in a particular union be unique. If the **<switch_type_spec>** is an **<enum_type>**, the identifier for the enumeration is in the scope of the union; as a result, it must be distinct from the element declarators.

It is not required that all possible values of the union discriminator be listed in the **<switch_body>**. The value of a union is the value of the discriminator together with one of the following:

- If the discriminator value was explicitly listed in a **case** statement, the value of the element associated with that **case** statement;
- If a default **case** label was specified, the value of the element associated with the default **case** label;
- No additional value.

The values of the constant expressions for the case labels of a single union definition must be distinct. A union type can contain a default label only where the values given in the non-default labels do not cover the entire range of the union's discriminant type.

Access to the discriminator and the related element is language-mapping dependent.

NOTE: While any ISO Latin-1 (8859.1) IDL character literal may be used in a **<case_label>** in a union definition whose discriminator type is **char**, not all of these characters are present in all transmission code sets that may be negotiated by GIOP

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

or in all native code sets that may be used by implementation language compilers and runtimes. When an attempt is made to marshal to CDR a **union** whose discriminator value of **char** type is not available in the negotiated transmission code set, or to demarshal from CDR a **union** whose discriminator value of **char** type is not available in the native code set, a **DATA_CONVERSION** system exception is raised. Therefore, to ensure portability and interoperability, care must be exercised when assigning the **<case_label>** for a **union** member whose discriminator type is **char**. Due to these issues, use of **char** types as the discriminator type for **union**s is not recommended.

7.11.2.3 Constructed Recursive Types and IForward Declarations

The IDL syntax allows the generation of recursive structures and unions via members that have a sequence type. The element type of a recursive sequence struct or union member must identify a struct, union, or valuetype. (A valuetype is allowed to have a member of its own type either directly or indirectly through a member of a constructed type—see 7.9.1.6, Value Type Example.) For example, the following is legal:

```
struct Foo {
    long value;
    sequence<Foo> chain;      // Deprecated (see  Section 7.11.6)
}
```

See Sequences on page 66 for details of the **sequence** template type.

IDL supports recursive types via a forward declaration for structures and unions (as well as for valuetypes—see 7.9.1.6, Value Type Example). Because anonymous types are deprecated (see Deprecated Anonymous Types on page 69), the previous example is better written as:

```
struct Foo;                              // Forward declaration
typedef sequence<Foo> FooSeq;
struct Foo {
    long value;
    FooSeq chain;
};
```

The forward declaration for the structure enables the definition of the sequence type **FooSeq**, which is used as the type of the recursive member.

Forward declarations are legal for structures and unions. A structure or union type is termed incomplete until its full definition is provided; that is, until the scope of the structure or union definition is closed by a terminating "}." For example:

```
struct Foo;      // Introduces Foo type name,
                 // Foo is incomplete now
                 // ...
struct Foo {
    // ...
};               // Foo is complete at this point
```

If a structure or union is forward declared, a definition of that structure or union must follow the forward declaration in the same source file. Compilers shall issue a diagnostic if this rule is violated. Multiple forward declarations of the same structure or union are legal.

If a sequence member of a structure or union refers to an incomplete type, the structure or union itself remains incomplete until the member's definition is completed. For example:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
struct Foo;
typedef sequence<Foo> FooSeq;
struct Bar {
    long value;
    FooSeq chain; //Use of incomplete type
};                //Bar itself remains incomplete
struct Foo {
    // ...
};                //Foo and Bar are complete
```

Compilers shall issue a diagnostic if this rule is violated.

Recursive definitions can span multiple levels. For example:

```
union Bar;                  // Forward declaration
typedef sequence<Bar> BarSeq;
union Bar switch(long) {    // Define incomplete union
    case 0:
        long l_mem;
    case 1:
        struct Foo {
            double d_mem;
            BarSeq nested;     // OK, recurse on enclosing
                               // incomplete type
    } s_mem;
};
```

An incomplete type can only appear as the element type of a sequence definition. A sequence with incomplete element type is termed an *incomplete sequence type*. For example:

```
struct Foo;                         // Forward declaration
typedef sequence<Foo> FooSeq;       // incomplete
```

An incomplete sequence type can appear only as the element type of another sequence, or as the member type of a structure or union definition. For example:

```
struct Foo;                         // Forward declaration
typedef sequence<Foo> FooSeq;       // OK
typedef sequence<FooSeq> FooTree; // OK

interface I {
    FooSeq op1();       // Illegal, FooSeq is incomplete
    void op2(           // Illegal, FooTree is incomplete
        in FooTree t
    );
};

struct Foo {            // Provide definition of Foo
    long l_mem;
    FooSeq chain;       // OK
    FooTree tree;       // OK
};
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

```
interface J {
    FooSeq op1();       // OK, FooSeq is complete
    void op2(
        in FooTree t    // OK, FooTree is complete
    );
};
```

Compilers shall issue a diagnostic if this rule is violated.

7.11.2.4 Enumerations

Enumerated types consist of ordered lists of identifiers. The syntax is:

(78)<enum_type>::=“enum” <identifier>
“{” <enumerator> { “,” <enumerator> }* “}”
(79) <enumerator>::=<identifier>

A maximum of 2^{32} identifiers may be specified in an enumeration; as such, the enumerated names must be mapped to a native data type capable of representing a maximally-sized enumeration. The order in which the identifiers are named in the specification of an enumeration defines the relative order of the identifiers. Any language mapping that permits two enumerators to be compared or defines successor/predecessor functions on enumerators must conform to this ordering relation. The **<identifier>** following the **enum** keyword defines a new legal type. Enumerated types may also be named using a **typedef** declaration.

7.11.3 Template Types

The template types are:

(47)<template_type_spec>::=<sequence_type>
| <string_type>
| <wide_string_type>
| <fixed_pt_type>

7.11.3.1 Sequences

OMG IDL defines the sequence type **sequence**. A sequence is a one-dimensional array with two characteristics: a maximum size (which is fixed at compile time) and a length (which is determined at run time).

The syntax is:

(80)<sequence_type>::=“sequence” “<” <simple_type_spec> “,” <positive_int_const> “>”
| “sequence” “<” <simple_type_spec> “>”

The second parameter in a sequence declaration indicates the maximum size of the sequence. If a positive integer constant is specified for the maximum size, the sequence is termed a bounded sequence. If no maximum size is specified, size of the sequence is unspecified (unbounded).

Prior to passing a bounded or unbounded sequence as a function argument (or as a field in a structure or union), the length of the sequence must be set in a language-mapping dependent manner. After receiving a sequence result from an operation invocation, the length of the returned sequence will have been set; this value may be obtained in a language-mapping dependent manner.

A sequence type may be used as the type parameter for another sequence type. For example, the following:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

typedef sequence< sequence<long> > Fred;

declares Fred to be of type “unbounded sequence of unbounded sequence of long.” Note that for nested sequence declarations, white space must be used to separate the two “>” tokens ending the declaration so they are not parsed as a single “>>” token.

7.11.3.2 Strings

OMG IDL defines the string type **string** consisting of all possible 8-bit quantities except null. A string is similar to a sequence of char. As with sequences of any type, prior to passing a string as a function argument (or as a field in a structure or union), the length of the string must be set in a language-mapping dependent manner. The syntax is:

(81)<string_type>::=“string” “<” <positive_int_const> “>”
| “string”

The argument to the string declaration is the maximum size of the string. If a positive integer maximum size is specified, the string is termed a bounded string. If no maximum size is specified, the string is termed an unbounded string.

Strings are singled out as a separate type because many languages have special built-in functions or standard library functions for string manipulation. A separate string type may permit substantial optimization in the handling of strings compared to what can be done with sequences of general types.

7.11.3.3 Wstrings

The **wstring** data type represents a sequence of wchar, except the wide character null. The type wstring is similar to that of type string, except that its element type is wchar instead of char. The actual length of a wstring is set at run-time and, if the bounded form is used, must be less than or equal to the bound.

The syntax for defining a wstring is:

(82) <wide_string_type>::=“wstring” “<” <positive_int_const> “>”
| “wstring”

7.11.3.4 Fixed Type

The **fixed** data type represents a fixed-point decimal number of up to 31 significant digits. The scale factor is a non-negative integer less than or equal to the total number of digits (note that constants with effectively negative scale, such as 10000, are always permitted).

The **fixed** data type will be mapped to the native fixed point capability of a programming language, if available. If there is not a native fixed point type, then the IDL mapping for that language will provide a fixed point data type. Applications that use the IDL fixed point type across multiple programming languages must take into account differences between the languages in handling rounding, overflow, and arithmetic precision.

The syntax of fixed type is:

(96)<fixed_pt_type>::=“fixed” “<“ <positive_int_const> “,” <positive_int_const> “>”
(97) <fixed_pt_const_type>::=“fixed”

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.11.4 Complex Declarator

7.11.4.1 Arrays

OMG IDL defines multidimensional, fixed-size arrays. An array includes explicit sizes for each dimension.

The syntax for arrays is:

(83)<array_declarator>::=<identifier> <fixed_array_size>+
(84) <fixed_array_size>::="[" <positive_int_const> "]"

The array size (in each dimension) is fixed at compile time. When an array is passed as a parameter in an operation invocation, all elements of the array are transmitted.

The implementation of array indices is language mapping specific; passing an array index as a parameter may yield incorrect results.

7.11.5 Native Types

OMG IDL provides a declaration for use by object adapters to define an opaque type whose representation is specified by the language mapping for that object adapter.

The syntax is:

(42)<type_dcl>::="native" <simple_declarator>
(51)<simple_declarator>::=<identifier>

This declaration defines a new type with the specified name. A native type is similar to an IDL basic type. The possible values of a native type are language-mapping dependent, as are the means for constructing them and manipulating them. Any interface that defines a native type requires each language mapping to define how the native type is mapped into that programming language.

A native type may be used only to define operation parameters, results, and exceptions. If a native type is used for an exception, it must be mapped to a type in a programming language that can be used as an exception. Native type parameters are permitted only in operations of **local interface**s or **valuetype**s. Any attempt to transmit a value of a native type in a remote invocation may raise the MARSHAL standard system exception.

It is recommended that native types be mapped to equivalent type names in each programming language, subject to the normal mapping rules for type names in that language. For example, in a hypothetical Object Adapter IDL module

```
module HypotheticalObjectAdapter {
    native Servant;
    interface HOA {
        Object activate_object(in Servant x);
    };
};
```

The IDL type Servant would map to **HypotheticalObjectAdapter::Servant** in C++ and the **activate_object** operation would map to the following C++ member function signature:

```
          CORBA::Object_ptr activate_object(
HypotheticalObjectAdapter::Servant x);
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The definition of the C++ type **`HypotheticalObjectAdapter::Servant`** would be provided as part of the C++ mapping for the HypotheticalObjectAdapter module.

NOTE: The native type declaration is provided specifically for use in object adapter interfaces, which require parameters whose values are concrete representations of object implementation instances. It is strongly recommended that it not be used in service or application interfaces. The native type declaration allows object adapters to define new primitive types without requiring changes to the OMG IDL language or to the OMG IDL compiler.

7.11.6 Deprecated Anonymous Types

IDL currently permits the use of anonymous types in a number of places. For example:

```
struct Foo {
    long value;
    sequence<Foo> chain;      // Legal (but deprecated)
}
```

Anonymous types cause a number of problems for language mappings and are therefore deprecated by this specification. Anonymous types will be removed in a future version, so new IDL should avoid use of anonymous types and use a typedef to name such types instead. Compilers need not issue a warning if a deprecated construct is encountered.

The following (non-exhaustive) examples illustrate deprecated uses of anonymous types.

Anonymous bounded string and bounded wide string types are deprecated. This rule affects constant definitions, attribute declarations, return value and parameter type declarations, sequence and array element declarations, and structure, union, exception, and valuetype member declarations. For example:

```
const string<5> GREETING = “Hello”;          // Deprecated

interface Foo {
    readonly attribute wstring<5> name;      // Deprecated
    wstring<5> op(in wstring<5> param);      // Deprecated
};
typedef sequence<wstring<5> > WS5Seq;        // Deprecated
typedef wstring<5> NameVector [10];          // Deprecated
struct A {
    wstring<5> mem;                          // Deprecated
};
// Anonymous member type in unions, exceptions,
// and valuetypes are deprecated as well.
```

This is better written as:

```
typedef string<5> GreetingType;
const GreetingType GREETING = “Hello”;

typedef wstring<5> ShortWName;
interface Foo {
    readonly attribute ShortWName name;
    ShortWName op(in ShortWName param);
};
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
typedef sequence<ShortWName> NameSeq;
typedef ShortWName NameVector[10];
struct A {
    GreetingType mem;
};
```

Anonymous fixed-point types are deprecated. This rule affects attribute declarations, return value and parameter type declarations, sequence and array element declarations, and structure, union, exception, and valuetype member declarations.

```
struct Foo {
    fixed<10,5> member;                    // Deprecated
};
```

This is better written as:

```
typedef fixed<10,5> MyType;
struct Foo {
    MyType member;
};
```

Anonymous member types in structures, unions, exceptions, and valuetypes are deprecated:

```
union U switch(long) {
    case 1:
        long array_mem[10];                // Deprecated
    case 2:
        sequence<long> seq_mem;            // Deprecated
    case 3:
        string<5> bstring_mem;
};
```

This is better written as:

```
typedef long LongArray[10];
typedef sequence<long> LongSeq;
typedef string<5> ShortName;
union U switch (long) {
    case 1:
        LongArray array_mem;
    case 2:
        LongSeq seq_mem;
    case 3:
        ShortName bstring_mem;
};
```

Anonymous array and sequence elements are deprecated:

```
typedef sequence<sequence<long> > NumberTree;    // Deprecated
typedef fixed<10,2> FixedArray[10];
```

This is better written as:

© ISO/IEC 2008 - All rights reserved
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
typedef sequence<long> ListOfNumbers;
typedef sequence<ListOfNumbers> NumberTree;
typedef fixed<10,2> Fixed_10_2;
typedef Fixed_10_2 FixedArray[10];
```

The preceding examples are not exhaustive. They simply illustrate the rule that, for a type to be used in the definition of another type, constant, attribute, return value, parameter, or member, that type must have a name. Note that the following example is not deprecated (even though stylistically poor):

```
struct Foo {
    struct Bar {
        long l_mem;
        double d_mem;
    } bar_mem_1;            // OK, not anonymous
    Bar bar_mem_2;          // OK, not anonymous
};
typedef sequence<Foo::Bar> FooBarSeq;      // Scoped names are OK
```

7.12 Exception Declaration

Exception declarations permit the declaration of struct-like data structures, which may be returned to indicate that an exceptional condition has occurred during the performance of a request. The syntax is as follows:

(86)<except_dcl>::=“exception” <identifier> “{“ <member>* “}”

Each exception is characterized by its OMG IDL identifier, an exception type identifier, and the type of the associated return value (as specified by the **<member>** in its declaration). If an exception is returned as the outcome to a request, then the value of the exception identifier is accessible to the programmer for determining which particular exception was raised.

If an exception is declared with members, a programmer will be able to access the values of those members when an exception is raised. If no members are specified, no additional information is accessible when an exception is raised.

An identifier declared to be an exception identifier may thereafter appear only in a raises clause of an operation declaration, and nowhere else.

A set of standard system exceptions is defined corresponding to standard run-time errors, which may occur during the execution of a request. These standard system exceptions are documented in 8.12, Exceptions.

7.13 Operation Declaration

Operation declarations in OMG IDL are similar to C function declarations. The syntax is:

(87) <op_dcl>::=[<op_attribute>] <op_type_spec>
<identifier> <parameter_dcls>
[<raises_expr>] [<context_expr>]

(88) <op_attribute>::=“oneway”

(89) <op_type_spec>::=<param_type_spec>
| “void”

An operation declaration consists of:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- An optional operation attribute that specifies which invocation semantics the communication system should provide when the operation is invoked. Operation attributes are described in 7.13.1, Operation Attribute.
- The type of the operation's return result; the type may be any type that can be defined in OMG IDL. Operations that do not return a result must specify the **void** type.
- An identifier that names the operation in the scope of the interface in which it is defined.
- A parameter list that specifies zero or more parameter declarations for the operation. Parameter declaration is described in 7.13.2, Parameter Declarations.
- An optional raises expression that indicates which exceptions may be raised as a result of an invocation of this operation. Raises expressions are described in 7.13.3, Raises Expressions.
- An optional context expression that indicates which elements of the request context may be consulted by the method that implements the operation. Context expressions are described in 7.13.4, Context Expressions.

Some implementations and/or language mappings may require operation-specific pragmas to immediately precede the affected operation declaration.

7.13.1 Operation Attribute

The operation attribute specifies which invocation semantics the communication service must provide for invocations of a particular operation. An operation attribute is optional. The syntax for its specification is as follows:

(88)<op_attribute>::="oneway"

When a client invokes an operation with the **oneway** attribute, the invocation semantics are best-effort, which does not guarantee delivery of the call; best-effort implies that the operation will be invoked at most once. An operation with the **oneway** attribute must not contain any output parameters and must specify a **void** return type. An operation defined with the **oneway** attribute may not include a raises expression; invocation of such an operation, however, may raise a standard system exception.

If an **<op_attribute>** is not specified, the invocation semantics is at-most-once if an exception is raised; the semantics are exactly-once if the operation invocation returns successfully.

7.13.2 Parameter Declarations

Parameter declarations in OMG IDL operation declarations have the following syntax:

(90)<parameter_dcls>::="(" <param_dcl> { "," <param_dcl> }* ")"
| "(" ")"
(91) <param_dcl>::=<param_attribute> <param_type_spec> <simple_declarator>
(92) <param_attribute>::="in"
| "out"
| "inout"
(95)<param_type_spec>::=<base_type_spec>
| <string_type>
| <wide_string_type>
| <scoped_name>

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A parameter declaration must have a directional attribute that informs the communication service in both the client and the server of the direction in which the parameter is to be passed. The directional attributes are:

- **in** - the parameter is passed from client to server.
- **out** - the parameter is passed from server to client.
- **inout** - the parameter is passed in both directions.

It is expected that an implementation will *not* attempt to modify an **in** parameter. The ability to even attempt to do so is language-mapping specific; the effect of such an action is undefined.

If an exception is raised as a result of an invocation, the values of the return result and any **out** and **inout** parameters are undefined.

7.13.3 Raises Expressions

There are two kinds of raises expressions as described in this sub clause.

7.13.3.1 Raises Expression

A **raises** expression specifies which exceptions may be raised as a result of an invocation of the operation or accessing (invoking the _get operation of) a readonly attribute. The syntax for its specification is as follows:

(93)<raises_expr>::=“raises” “(” <scoped_name>
{ “,” <scoped_name> }* “)”

The **<scoped_name>**s in the **raises** expression must be previously defined exceptions or native types. If a native type is used as an exception for an operation, the operation must appear in either a local interface or a valuetype.

In addition to any operation-specific exceptions specified in the **raises** expression, there are a standard set of system exceptions that may be signalled by the ORB. These standard system exceptions are described in 8.12.3, Standard System Exception Definitions. However, standard system exceptions may *not* be listed in a **raises** expression.

The absence of a **raises** expression on an operation implies that there are no operation-specific exceptions. Invocations of such an operation are still liable to receive one of the standard system exceptions.

7.13.3.2 getraises and setraises Expressions

getraises and **setraises** expressions specify which exceptions may be raised as a result of an invocation of the accessor (**_get**) and a mutator (**_set**) functions of an attribute. The syntax for its specification is as follows:

(108)<attr_raises_expr> ::=<get_excep_expr> [<set_excep_expr>]
| <set_excep_expr>
(109) <get_excep_expr> ::= “getraises” <exception_list>
(110) <set_excep_expr> ::= “setraises” <exception_list>
(111) <exception_list::= “(” <scoped_name>
{ “,” <scoped_name> } * “)”

The **<scoped_name>**s in the **getraises** and **setraises** expressions must be previously defined exceptions.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

In addition to any attribute-specific exceptions specified in the **getraises** and **setraises** expressions, there are a standard set of exceptions that may be signalled by the ORB. These standard exceptions are described in 8.12.3, Standard System Exception Definitions. However, standard exceptions may *not* be listed in a **getraises** or **setraises** expression.

The absence of a **getraises** or **setraises** expression on an attribute implies that there are no accessor-specific or mutator-exceptions respectively. Invocations of such an accessor or mutator are still liable to receive one of the standard exceptions.

NOTE: The exceptions associated with the accessor operation corresponding to a **readonly attribute** is specified using a simple **raises** expression as specified in 7.13.3.1, Raises Expression. The **getraises** and **setraises** expressions are used only in **attribute**s that are not **readonly**.

7.13.4 Context Expressions

A **context** expression specifies which elements of the client's context may affect the performance of a request by the object. The syntax for its specification is as follows:

(94)<context_expr>::="context" "(" <string_literal>
{ "," <string_literal> }* ")"

The run-time system guarantees to make the value (if any) associated with each **<string_literal>** in the client's context available to the object implementation when the request is delivered. The ORB and/or object is free to use information in this *request context* during request resolution and performance.

The absence of a context expression indicates that there is no request context associated with requests for this operation.

Each **string_literal** is a non-empty string. If the character '*' appears in **string_literal**, it must appear only once, as the last character of **string_literal**, and must be preceded by one or more characters other than '*'.

The mechanism by which a client associates values with the context identifiers is described in 8.6, Context Object.'

7.14 Attribute Declaration

An interface can have attributes as well as operations; as such, attributes are defined as part of an interface. An attribute definition is logically equivalent to declaring a pair of accessor functions; one to retrieve the value of the attribute and one to set the value of the attribute.

The syntax for **attribute** declaration is:

(85)<attr_dcl>::= <readonly_attr_spec>
| <attr_spec>
(104)<readonly_attr_spec> ::= "readonly" "attribute" <param_type_spec> <readonly_attr_declarator>
(105) <readonly_attr_declarator>::= <simple_declarator> <raises_expr>
| <simple_declarator>
{ "," <simple_declarator> }*
(106) <attr_spec> ::= "attribute" <param_type_spec> <attr_declarator>
(107) <attr_declarator> ::=<simple_declarator> <attr_raises_expr>
| <simple_declarator>
{ "," <simple_declarator> }*

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The optional **readonly** keyword indicates that there is only a single accessor function—the retrieve value function. Consider the following example:

```
interface foo {
    enum material_t {rubber, glass};
    struct position_t {
        float x, y;
    };

    attribute float radius;
    attribute material_t material;
    readonly attribute position_t position;
    • • •
};
```

The attribute declarations are equivalent to the following pseudo-specification fragment, assuming that one of the leading '_'s is removed by application of the Escaped Identifier rule described in Escaped Identifiers on page 32.

```
• • •
float       __get_radius ();
void        __set_radius (in float r);
material_t  __get_material ();
void        __set_material (in material_t m);
position_t  __get_position ();
• • •
```

The actual accessor function names are language-mapping specific. The attribute name is subject to OMG IDL's name scoping rules; the accessor function names are guaranteed *not* to collide with any legal operation names specifiable in OMG IDL.

Attributes are inherited. An attribute name *cannot* be redefined to be a different type. See 7.19, CORBA Module for more information on redefinition constraints and the handling of ambiguity.

7.15 Repository Identity Related Declarations

Two constructs that are provided for specifying information related to Repository Id are described in this sub clause.

7.15.1 Repository Identity Declaration

The syntax of a repository identity declaration is as follows:

(102)<type_id_dcl> ::="typeid" <scoped_name> <string_literal>

A repository identifier declaration includes the following elements:

- the keyword **typeid.**
- a <*scoped_name*> that denotes the named IDL construct to which the repository identifier is assigned.
- a string literal that must contain a valid repository identifier value.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The <*scoped_name*> is resolved according to normal IDL name resolution rules, based on the scope in which the declaration occurs. It must denote a previously-declared name of one of the following IDL constructs:

- module
- interface
- component
- home
- facet
- receptacle
- event sink
- event source
- finder
- factory
- event type
- value type
- value type member
- value box
- constant
- typedef
- exception
- attribute
- operation
- enum
- local

The value of the string literal is assigned as the repository identity of the specified type definition. This value will be returned as the **RepositoryId** by the interface repository definition object corresponding to the specified type definition. Language mappings constructs, such as Java helper classes, that return repository identifiers shall return the values declared for their corresponding definitions.

At most one repository identity declaration may occur for any named type definition. An attempt to redefine the repository identity for a type definition is illegal, regardless of the value of the redefinition.

If no explicit repository identity declaration exists for a type definition, the repository identifier for the type definition shall be an IDL format repository identifier, as defined in 14.7.1, OMG IDL Format.

7.15.2 Repository Identifier Prefix Declaration

The syntax of a repository identifier prefix declaration is as follows:

(103) <type_prefix_dcl>::=“typeprefix” <scoped_name> <string_literal>

A repository identifier declaration includes the following elements:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- The keyword **typeprefix**.
- A <*scoped_name*> that denotes an IDL name scope to which the prefix applies.
- A string literal that must contain the string to be prefixed to repository identifiers in the specified name scope.

The <*scoped_name*> is resolved according to normal IDL name resolution rules, based on the scope in which the declaration occurs. It must denote a previously-declared name of one of the following IDL constructs:

- module
- interface (including abstract or local interface)
- value type (including abstract, custom, and box value types)
- event type (including abstract and custom value types)
- specification scope (**::**)

The specified string is prefixed to the body of all repository identifiers in the specified name scope, whose values are assigned by default. The specified string shall be a list of one or more identifiers, separated by the “/” characters. These identifiers are arbitrarily long sequences of alphabetic, digit, underscore (“_”), hyphen (“-”), and period (“.”) characters. The string shall not contain a trailing slash ("/"), and it shall not begin with the characters underscore ("_"), hyphen ("-") or period ("."). To elaborate:

By “prefixed to the body of a repository identifier,” we mean that the specified string is inserted into the default IDL format repository identifier immediately after the format name and colon (“IDL:”) at the beginning of the identifier. A forward slash (‘/’) character is inserted between the end of the specified string and the remaining body of the repository identifier.

The prefix is only applied to repository identifiers whose values are not explicitly assigned by a typeid declaration. The prefix is applied to all such repository identifiers in the specified name scope, including the identifier of the construct that constitutes the name scope.

7.15.3 Repository Id Conflict

In IDL that contains both pragma prefix/ID declarations (as defined in Pragma Directives for RepositoryId on page 277) and typeprefix/typeid declarations (as defined in Repository Identity Declaration on page 75 and Repository Identifier Prefix Declaration on page 76), if the repository id for an IDL element computed by using pragmas and typeid/typeprefix are not identical it is an error. Note that this rule applies only when the repository id value computation uses explicitly declared values from declarations of both kinds. If the repository id computed using explicitly declared values of one kind conflicts with 9ne computed with implicit values of the other kind, the repository id based on explicitly declared values shall prevail.

7.16 Event Declaration

Event type is a specialization of value type dedicated to asynchronous component communication. There are several kinds of event type declarations: “regular” event types, abstract event types, and forward declarations.

An event declaration satisfies the following syntax:

(134)<event> ::= (<event_dcl> | <event_abs_dcl> | <event_forward_dcl>)

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.16.1 Regular Event Type

A regular event type satisfies the following syntax:

(137) <event_dcl>::=<event_header> “{” <value_element> * “}”
(138) <event_header>::=[“custom”] “eventtype”
<identifier> [<value_inheritance_spec>]

7.16.1.1 Event Header

The event header consists of two elements:

- The event type’s name and optional modifier specifying whether the event type uses custom marshaling.
- An optional value inheritance specification described in 7.9.1.3, Value Inheritance Specification.

7.16.1.2 Event Element

An event can contain all the elements that a value can as described in 7.9.1.2, Value Element (i.e., attributes, operations, initializers, state members).

7.16.2 Abstract Event Type

(136)<event_abs_dcl> ::=“abstract” “eventtype” <identifier>
[<value_inheritance_spec>]
“{” <export>* “}”

Event types may also be abstract. They are called abstract because an abstract event type may not be instantiated. No <state_member> or <initializers> may be specified. However, local operations may be specified. Essentially they are a bundle of operation signatures with a purely local implementation.

Note that a concrete event type with an empty state is not an abstract event type.

7.16.3 Event Forward Declaration

(135)<event_forward_dcl> ::=[“abstract”] “eventtype” <identifier>

A forward declaration declares the name of an event type without defining it. This permits the definition of event types that refer to each other. The syntax consists simply of the keyword **eventtype** followed by an **<identifier>** that names the event type.

Multiple forward declarations of the same event type name are legal.

It is illegal to inherit from a forward-declared event type whose definition has not yet been seen.

7.16.4 Eventtype Inheritance

As event type is a specialization of value type then event type inheritance is directly analogous to value inheritance (see 7.9.1.3, Value Inheritance Specification for a detailed description of the analogous properties for valuetypes). In addition, an event type could inherit from a single immediate base concrete event type, which must be the first element specified in the inheritance list of the event declaration’s IDL. It may be followed by other abstract values or events from which it inherits.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.17 Component Declaration

7.17.1 Component

A component declaration describes an interface for a component. The salient characteristics of a component declaration are as follows:

- A component declaration specifies the name of the component.
- A component declaration may specify a list of interfaces that the component supports.
- Component declarations support single inheritance from other component definitions.
- Component declarations may include in its body any attribute declarations that are legal in normal interface declarations, together with declarations of facets and receptacles of the component, and the event sources and sinks that the component defines.

7.17.1.1 Syntax

The syntax for declaring a component is as follows:

(112)<component> ::=<component_dcl>
| <component_forward_dcl>
(113) <component_forward_dcl> ::= “component” <identifier>
(114) <component_dcl> ::= <component_header>
“{” <component_body> “}”

<component_forward_dcl> is described in 7.17.1.2, Forward Declaration.

<component_header> is described in 7.17.2, Component Header.

<component_body> is described in 7.17.3, Component Body.

7.17.1.2 Forward Declaration

A forward declaration declares the name of a component without defining it. This permits the definition of components that refer to each other. The syntax consists simply of the keyword **component** followed by an **<identifier>** that names the component. The actual definition must follow later in the specification.

Multiple forward declarations of the same component name are legal.

It is illegal to inherit from a forward-declared component whose definition has not yet been seen.

7.17.2 Component Header

A **<component_header>** declares the primary characteristics of a component interface.

7.17.2.1 Syntax

The syntax for declaring a component header is as follows:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(115) <component_header> ::= “component” <identifier>
[<component_inheritance_spec>]
[<supported_interface_spec>]
(116) <supported_interface_spec> ::= “supports” <scoped_name>
{ “,” <scoped_name> }*
(117) <component_inheritance_spec> ::= “:” <scoped_name>

A component header comprises the following elements:

- The keyword **component**.
- An <*identifier*> that names the component type.
- An optional <*inheritance_spec*>, consisting of a colon and a single <*scoped_name*> that must denote a previously-defined component type.
- An optional <*supported_interface_spec*> that must denote one or more previously-defined IDL interfaces.

7.17.2.2 Supported interfaces

A component may optionally support one or more interfaces. When a component definition header includes a supports clause as follows:

component <*component_name*> supports <*interface_name*> { … };

For further details see the *CORBA Components* specification, Clause 1, Supported Interfaces.

7.17.2.3 Component Inheritance

A component may optionally inherit from a component that supports one or more interfaces. This is specified by using the inheritance construct that looks like:

component <component_name> : <component_name> { ... };

The following rules apply to component inheritance:

- A derived component type may not directly support an interface.
- The interface for a derived component type is derived from the interface of its base component type.
- A component type may have at most one base component type.
- The features of a component that are inherited by the derived component are:
  - the **provides** statements
  - the **uses** statements
  - the **emits** statements
  - the **publishes** statements
  - the **consumes** statements
  - attributes

See 7.17.2.3, Component Inheritance for details of component inheritance.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.17.3 Component Body

(118) <component_body> ::=<component_export>*
(119) <component_export> ::=<provides_dcl> ";"
| <uses_dcl> ";"
| <emits_dcl> ";"
| <publishes_dcl> ";"
| <consumes_dcl> ";"
| <attr_dcl> ";"

A component forms a naming scope, nested within the scope in which the component is declared. A component body can contain the following kinds of declarations:

- Facet declarations (**provides**)
- Receptacle declarations (**uses**)
- Event source declarations (**emits** or **publishes**)
- Event sink declarations (**consumes**)
- Attribute declarations (**attribute** and **readonly attribute**)

These declarations and their meanings are described in detail in the *CORBA Components* specification, Component Model clause, "Facets and Navigation" through "Events" sub clauses.

7.17.3.1 Facets and Navigation

A component type may provide several independent interfaces to its clients in the form of facets. Facets are intended to be the primary vehicle through which a component exposes its functional application behavior to clients during normal execution. A component may exhibit zero or more facets.

Syntax

A facet is declared with the following syntax:

(120) <provides_dcl> ::= "provides" <interface_type> <identifier>
(121) <interface_type> ::= <scoped_name>
| "Object"

The interface type shall be either the keyword **Object**, or a scoped name that denotes a previously-declared interface type that is not a component interface (i.e., is not the interface corresponding to a component definition). The identifier names the facet within the scope of the component, allowing multiple facets of the same type to be provided by the component.

See the *CORBA Components* specification, Component Model clause, "Facets and Navigation" for further details.

7.17.3.2 Receptacles

A component definition can describe the ability to accept object references upon which the component may invoke operations. When a component accepts an object reference in this manner, the relationship between the component and the referent object is called a *connection*; they are said to be *connected*. The conceptual point of connection is called a *receptacle*. A receptacle is an abstraction that is concretely manifested on a component as a set of operations for establishing and managing connections. A component may exhibit zero or more receptacles.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Syntax

The syntax for describing a receptacle is as follows:

(122) <uses_dcl> ::= “uses” [“multiple”]
< interface_type> <identifier>

A receptacle declaration comprises the following elements:

- The keyword **uses**.
- The optional keyword **multiple**. The presence of this keyword indicates that the receptacle may accept multiple connections simultaneously, and results in different operations on the component’s associated interface.
- An <*interface_type*>, which must be either the keyword **Object** or a scoped name that denotes the interface type that the receptacle will accept. The scoped name must denote a previously-defined non-component interface type.
- An <*identifier*> that names the receptacle in the scope of the component.

See the *CORBA Components* specification (Part 3), Component Model clause, “Receptacles” sub clause for further details.

7.17.4 Event Sources—publishers and emitters

An event source embodies the potential for the component to generate events of a specified type, and provides mechanisms for associating consumers with sources.

There are two categories of event sources, *publishers* and *emitters*. Both are implemented using event channels supplied by the container. An emitter can be connected to at most one consumer. A publisher can be connected through the channel to an arbitrary number of consumers, who are said to *subscribe* to the publisher event source. A component may exhibit zero or more emitters and publishers.

7.17.4.1 Publishers

Syntax

The syntax for an event publisher is as follows:

(124)<publishes_dcl> ::= “publishes” <scoped_name> <identifier>

A publisher declaration consists of the following elements:

- The keyword **publishes**.
- A <*scoped_name*> that denotes a previously-defined event type.
- An <*identifier*> that names the publisher event source in the scope of the component.

See the *CORBA Components* specification, Component Model clause, “Publisher” sub clause for further details.

7.17.4.2 Emitters

Syntax

The syntax for an emitter declaration is as follows:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(123)<emits_dcl> ::= “emits” <scoped_name> <identifier>

An emitter declaration consists of the following elements:

- The keyword **emits**.
- A <*scoped_name*> that denotes a previously-defined event type.
- An <*identifier*> that names the event source in the scope of the component.

See the *CORBA Components* specification, Component Model clause, “Emitters” sub clause for further details.

7.17.5 Event Sinks

An event sink embodies the potential for the component to receive events of a specified type. An event sink is, in essence, a special-purpose facet whose type is an event consumer. External entities, such as clients or configuration services, can obtain the reference for the consumer interface associated with the sink.

A component may exhibit zero or more consumers.

See the *CORBA Components* specification, Component Model clause, “Event Sinks” sub clause for further details.

Syntax

The syntax for an event sink declaration is as follows:

(125)<consumes_dcl> ::= “consumes” <scoped_name> <identifier>

An event sink declaration contains the following elements:

- The keyword **consumes**.
- A <*scoped_name*> that denotes a previously-defined event type.
- An <*identifier*> that names the event sink in the component’s scope.

See the *CORBA Components* specification, Component Model clause, “Event Sinks” sub clause for further details.

7.17.6 Basic and Extended Components

A component that satisfies the following properties is known as a *Basic Component*:

- It does not inherit from another component.
- Its declaration does not contain any provides statements.
- Its declaration does not contain any uses statements.
- Its declaration does not contain any publishes, emits, or consumes statements.

In effect a declaration of a *Basic Component* fits the pattern:

“component” <identifier> [<supported_interface_spec>]
“{“ {<attr_dcl> “;”}* “}”

A component that is not a *Basic Component* is referred to as an *Extended Component*.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.18 Home Declaration

A home declaration describes an interface for managing instances of a specified component type.

7.18.1 Home

The salient characteristics of a home declaration are as follows:

- A home declaration must specify exactly one component type that it manages. Multiple homes may manage the same component type.
- A home declaration may specify a primary key type. Primary keys are values assigned by the application environment that uniquely identify component instances managed by a particular home. Primary key types must be value types derived from **Components::PrimaryKeyBase**. There are more specific constraints placed on primary key types, which are specified in the *CORBA Components* specification, Component Model clause, “Primary key type constraints” sub clause.
- Home declarations may include any declarations that are legal in normal interface declarations.
- Home declarations support single inheritance from other home definitions, subject to a number of constraints that are described in the *CORBA Components* specification, Component Model clause, “Home inheritance” sub clause.
- Home declarations may specify a list of interfaces that the home supports.

Syntax

The syntax for a home definition is as follows:

(126) <home_dcl> ::= <home_header> <home_body>

<home_header> is described in “Home Header.”

<home_body> is described in “Home Body.”

7.18.2 Home Header

A <*home_header*> describes fundamental characteristics of a home interface.

Syntax

The syntax for a home header declaration is as follows:

(127) <home_header> ::= “home” <identifier>
[<home_inheritance_spec>]
[<supported_interface_spec>]
“manages” <scoped_name>
[<primary_key_spec>]
(128) <home_inheritance_spec> ::= “:” <scoped_name>
(129) <primary_key_spec> ::= “primarykey” <scoped_name>

A <*home_header*> consists of the following elements:

- The keyword **home**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- An <*identifier*> that names the home in the enclosing name scope.
- An optional <*home_inheritance_spec*>, consisting of a colon "**:**" and a single <*scoped_name*> that denotes a previously defined home type.
- An optional <*supported_interface_spec*> that must denote one or more previously defined IDL interfaces.
- The keyword **manages**.
- A <*scoped_name*> that denotes a previously defined component type.
- An optional primary key definition, consisting of the keyword **primarykey** followed by a <*scoped_name*> that denotes a previously defined value type that is derived from the abstract value type **Components::PrimaryKeyBase.** Additional constraints on primary keys are described in the *CORBA Components* specification, Component Model clause, "Primary key type constraints" sub clause.

Details of semantics can be found in the *CORBA Components* specification, Component Model clause, "Homes" sub clause.

7.18.3 Home Body

(130) <home_body> ::= "{" <home_export>* "}"
(131) <home_export ::= <export>
| <factory_dcl> ";"
| <finder_dcl> ";"

7.18.3.1 Operation Declarations

A home body may include zero or more operation declarations, where the operation may be a *factory* operation, a *finder* operation, or a normal operation or attribute.

Factory operations

The syntax of a factory operation is as follows:

(132) <factory_dcl> ::= "factory" <identifier>
"(" [<init_param_decls>] ")"
[<raises_expr>]

A factor operation declaration consists of the following elements:

- The keyword **factory**.
- An <identifier> that names the operation in the scope of the home declaration.
- An optional list of initialization parameters (<*init_param_decls*>) enclosed in parentheses.
- An optional <*raises_expr*> declaring exceptions that may be raised by the operation.

A factory declaration has an implicit return value of type reference to component.

See the *CORBA Components* specification, Component Model clause, "Factory operations" sub clause for further details.

Finder operations

The syntax of a finder operation is as follows:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(133) <finder_dcl> ::= “finder” <identifier>
“(“ [<init_param_decls>] “)”
[<raises_expr>]

A finder operation declaration consists of the following elements:

- The keyword **finder**.
- An identifier that names the operation in the scope of the storage home declaration.
- An optional list of initialization parameters (<*init_param_decls*>) enclosed in parentheses.
- An optional <*raises_expr*> declaring exceptions that may be raised by the operation.

A finder declaration has an implicit return value of type reference to component.

See the *CORBA Components* specification, Component Model clause, “Finder operations” sub clause for further details.

7.19 CORBA Module

Names defined by the CORBA specification are in a module named CORBA. In an OMG IDL specification, however, OMG IDL keywords such as **Object** must not be preceded by a “**CORBA::**” prefix. Other interface names such as **TypeCode** are not OMG IDL keywords, so they must be referred to by their fully scoped names (e.g., **CORBA::TypeCode**) within an OMG IDL specification.

For example in:

```
#include <orb.idl>
module M {
    typedef CORBA::Object myObjRef;     // Error: keyword Object scoped
    typedef TypeCode myTypeCode;        // Error: TypeCode undefined
    typedef CORBA::TypeCode TypeCode;// OK
};
```

The file **orb.idl** contains the IDL definitions for the **CORBA** module. Except for **CORBA::TypeCode**, the file **orb.idl** must be included in IDL files that use names defined in the **CORBA** module. IDL files that use **CORBA::TypeCode** may obtain its definition by including either the file **orb.idl** or the file **TypeCode.idl**.

The exact contents of **TypeCode.idl** are implementation dependent. One possible implementation of **TypeCode.idl** may be:

```
// PIDL
#ifndef _TYPECODE_IDL_
#define _TYPECODE_IDL_
#pragma prefix "omg.org"
module CORBA {
    interface TypeCode;
};
#endif // _TYPECODE_IDL_
```

For IDL compilers that implicitly define **CORBA::TypeCode**, **TypeCode.idl** could consist entirely of a comment as shown below:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
// PIDL
// CORBA::TypeCode implicitly built into the IDL compiler
// Hence there are no declarations in this file
```

Because the compiler implicitly contains the required declaration, this file meets the requirement for compliance.

The version of **CORBA** specified in this release of the specification is version **<x.y>**, and this is reflected in the IDL for the **CORBA** module by including the following pragma version (see 14.7.5.3, The Version Pragma):

```
#pragma version CORBA <x.y>
```

as the first line immediately following the very first **CORBA** module introduction line, which in effect associates that version number with the **CORBA** entry in the **IR**. The version number in that version pragma line must be changed whenever any changes are made to any remotely accessible parts of the **CORBA** module in an officially released OMG standard.

7.20 Names and Scoping

OMG IDL identifiers are case insensitive; that is, two identifiers that differ only in the case of their characters are considered redefinitions of one another. However, all references to a definition must use the same case as the defining occurrence. This allows natural mappings to case-sensitive languages. For example:

```
module M {
    typedef long Long;      // Error: Long clashes with keyword long
    typedef long TheThing;
    interface I {
        typedef long MyLong;
        myLong op1(                 // Error: inconsistent capitalization
            in TheThing thething;  // Error: TheThing clashes with thething
        );
    };
};
```

7.20.1 Qualified Names

A qualified name (one of the form <scoped-name>::<identifier>) is resolved by first resolving the qualifier <scoped-name> to a scope S, and then locating the definition of <identifier> within S. The identifier must be directly defined in S or (if S is an interface) inherited into S. The <identifier> is not searched for in enclosing scopes.

When a qualified name begins with “::”, the resolution process starts with the file scope and locates subsequent identifiers in the qualified name by the rule described in the previous paragraph.

Every OMG IDL definition in a file has a global name within that file. The global name for a definition is constructed as follows.

Prior to starting to scan a file containing an OMG IDL specification, the name of the current root is initially empty (“”) and the name of the current scope is initially empty (“”). Whenever a **module** keyword is encountered, the string “::” and the associated identifier are appended to the name of the current root; upon detection of the termination of the **module**, the trailing “::” and identifier are deleted from the name of the current root. Whenever an **interface**, **struct**, **union**, or **exception** keyword is encountered, the string “::” and the associated identifier are appended to the name of the current scope; upon detection of the termination of the **interface**, **struct**, **union**, or **exception**, the trailing “::” and identifier

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

are deleted from the name of the current scope. Additionally, a new, unnamed, scope is entered when the parameters of an operation declaration are processed; this allows the parameter names to duplicate other identifiers; when parameter processing has completed, the unnamed scope is exited.

The global name of an OMG IDL definition is the concatenation of the current root, the current scope, a "::", and the <identifier>, which is the local name for that definition.

Note that the global name in an OMG IDL files correspond to an absolute **ScopedName** in the Interface Repository. (See 14.5.1, Supporting Type Definitions').

Inheritance causes all identifiers defined in base interfaces, both direct and indirect, to be visible in derived interfaces. Such identifiers are considered to be semantically the same as the original definition. Multiple paths to the same original identifier (as results from the diamond shape in Figure 7.1 on page 47) do not conflict with each other.

Inheritance introduces multiple global OMG IDL names for the inherited identifiers. Consider the following example:

```
interface A {
    exception E {
        long L;
    };
    void f() raises(E);
};

interface B: A {
    void g() raises(E);
};
```

In this example, the exception is known by the global names **::A::E** and **::B::E**.

Ambiguity can arise in specifications due to the nested naming scopes. For example:

```
interface A {
    typedef string<128> string_t;
};

interface B {
    typedef string<256> string_t;
};

interface C: A, B {
    attribute string_t      Title;      // Error: Ambiguous
    attribute A::string_t   Name;       // OK
    attribute B::string_t   City;       // OK
};
```

The declaration of attribute **Title** in interface **C** is ambiguous, since the compiler does not know which **string_t** is desired. Ambiguous declarations yield compilation errors.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.20.2 Scoping Rules and Name Resolution

Contents of an entire OMG IDL file, together with the contents of any files referenced by #include statements, forms a naming scope. Definitions that do not appear inside a scope are part of the global scope. There is only a single global scope, irrespective of the number of source files that form a specification.

The following kinds of definitions form scopes:

- module
- interface
- valuetype
- struct
- union
- operation
- exception
- eventtype
- component
- home

The scope for module, interface, valuetype, struct, exception, eventtype, component, and home begins immediately following its opening '{' and ends immediately preceding its closing '}'. The scope of an operation begins immediately following its '(' and ends immediately preceding its closing ')'. The scope of a union begins immediately following the '(' following the keyword **switch**, and ends immediately preceding its closing '}'. The appearance of the declaration of any of these kinds in any scope, subject to semantic validity of such declaration, opens a nested scope associated with that declaration.

An identifier can only be defined once in a scope. However, identifiers can be redefined in nested scopes. An identifier declaring a module is considered to be defined by its first occurrence in a scope. Subsequent occurrences of a module declaration with the same identifier within the same scope reopens the module and hence its scope, allowing additional definitions to be added to it.

The name of an interface, value type, struct, union, exception, or a module may not be redefined within the immediate scope of the interface, value type, struct, union, exception, or the module. For example:

```
module M {
    typedef short M;          // Error: M is the name of the module
                              //        in the scope of which the typedef is.
    interface I {
        void i (in short j);  // Error: i clashes with the interface name I
    };
};
```

An identifier from a surrounding scope is introduced into a scope if it is used in that scope. An identifier is not introduced into a scope by merely being visible in that scope. The use of a scoped name introduces the identifier of the outermost scope of the scoped name. For example in:

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
module M {
    module Inner1 {
        typedef string S1;
    };

    module Inner2 {
        typedef string inner1;      // OK
    };
}
```

The declaration of **Inner2::inner1** is OK because the identifier **Inner1**, while visible in module **Inner2**, has not been introduced into module **Inner2** by actual use of it. On the other hand, if module **Inner2** were:

```
module Inner2{
    typedef Inner1::S1 S2;      // Inner1 introduced
    typedef string inner1;      // Error
    typedef string S1;          // OK
};
```

The definition of **inner1** is now an error because the identifier **Inner1** referring to the **module Inner1** has been introduced in the scope of module **Inner2** in the first line of the module declaration. Also, the declaration of **S1** in the last line is OK since the identifier **S1** was not introduced into the scope by the use of **Inner1::S1** in the first line.

Only the first identifier in a qualified name is introduced into the current scope. This is illustrated by **Inner1::S1** in the example above, which introduces "**Inner1**" into the scope of "**Inner2**" but does not introduce "**S1**." A qualified name of the form "**::X::Y::Z**" does not cause "**X**" to be introduced, but a qualified name of the form "**X::Y::Z**" does.

Enumeration value names are introduced into the enclosing scope and then are treated like any other declaration in that scope. For example:

```
interface A {
    enum E { E1, E2, E3 };      // line 1

    enum BadE { E3, E4, E5 }; // Error: E3 is already introduced
                              // into the A scope in line 1 above
};

interface C {
    enum AnotherE { E1, E2, E3 };
};

interface D : C, A {
    union U switch ( E ) {
        case A::E1 : boolean b;// OK.
        case E2 : long l;       // Error: E2 is ambiguous (notwithstanding
                                // the switch type specification!!)
    };
};
```

Type names defined in a scope are available for immediate use within that scope. In particular, see 7.11.2, Constructed Types on cycles in type definitions.

© ISO/IEC 2008 - All rights reserved
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A name can be used in an unqualified form within a particular scope; it will be resolved by successively searching farther out in enclosing scopes, while taking into consideration inheritance relationships among interfaces. For example:

```
module M {
    typedef long ArgType;
    typedef ArgType AType;          // line l1
    interface B {
        typedef string ArgType;     // line l3
        ArgType opb(in AType i);    // line l2
    };
};

module N {
    typedef char ArgType;           // line l4
    interface Y : M::B {
        void opy(in ArgType i);     // line l5
    };
};
```

The following scopes are searched for the declaration of **ArgType** used on **line l5**:

1. Scope of **N::Y** before the use of **ArgType**.
2. Scope of **N::Y**'s base interface **M::B**. (inherited scope).
3. Scope of **module N** before the definition of **N::Y**.
4. Global scope before the definition of **N**.

M::B::ArgType is found in **step 2** in **line l3**, and that is the definition that is used in **line l5**, hence **ArgType** in **line l5** is **string**. It should be noted that **ArgType** is not **char** in **line l5**. Now if **line l3** were removed from the definition of interface **M::B**, then **ArgType** on **line l5** would be **char** from **line l4**, which is found in **step 3**.

Following analogous search steps for the types used in the operation **M::B::opb** on **line l2**, the type of **AType** used on **line l2** is **long** from the **typedef** in **line l1** and the return type **ArgType** is **string** from **line l3.**

7.20.3 Special Scoping Rules for Type Names

Once a type has been defined anywhere within the scope of a module, interface or valuetype, it may not be redefined except within the scope of a nested module, interface or valuetype, or within the scope of a derived interface or valuetype. For example:

```
typedef short TempType;         // Scope of TempType begins here

module M {
    typedef string ArgType;     // Scope of ArgType begins here
    struct S {
        ::M::ArgType a1;         // Nothing introduced here
        M::ArgType a2;           // M introduced here
        ::TempType temp;        // Nothing introduced here
    };                  // Scope of (introduced) M ends here
    // ...
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
};                      // Scope of ArgType ends here

// Scope of global TempType ends here (at end of file)
```

The scope of an introduced type name is from the point of introduction to the end of its enclosing scope.

However, if a *type* name is *introduced* into a scope that is nested in a non-module scope definition, its *potential* scope extends over all its enclosing scopes out to the enclosing non-module scope. (For types that are defined outside an inon-module scope, the scope and the potential scope are identical.) For example:

```
module M {
    typedef long ArgType;
    const long I = 10;
    typedef short Y;

    interface A {
        struct S {
            struct T {
                ArgType x[I];   // ArgType and I introduced
                long y;         // a new y is defined, the existing Y
                                // is not used
            } m;
        };
        typedef string ArgType;     // Error: ArgType redefined
        enum I { I1, I2 };          // Error: I redefined
        typedef short Y;            // OK
    };  // Potential scope of  ArgType and I ends here

    interface B : A {
        typedef long ArgType  // OK, redefined in derived interface
        struct S {            // OK, redefined in derived interface
            ArgType x;        // x is a long
            A::ArgType y;     // y is a string
        };
    };
};
```

A type may not be redefined within its scope or potential scope, as shown in the preceding example. This rule prevents type names from changing their meaning throughout a non-module scope definition, and ensures that reordering of definitions in the presence of introduced types does not affect the semantics of a specification.

Note that, in the following, the definition of **M::A::U::I** is legal because it is outside the potential scope of the I introduced in the definition of **M::A::S::T::ArgType**. However, the definition of **M::A::I** is still illegal because it is within the potential scope of the I introduced in the definition of **M::A::S::T::ArgType**.

```
module M {
    typedef long ArgType;
    const long I = 10;

    interface A {
        struct S {
            struct T {
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
                ArgType x[I];    // ArgType and I introduced
            } m;
        };
        struct U {
            long I;              // OK, I is not a type name
        };
        enum I { I1, I2 };      // Error: I redefined
    }; // Potential scope of  ArgType and I ends here
};
```

Note that redefinition of a type after use in a module is OK as in the example:

```
typedef long ArgType;
module M {
    struct S {
        ArgType x;          // x is a long
    };

    typedef string ArgType;     // OK!
    struct T {
        ArgType y;          // Ugly but OK, y is a string
    };
};
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8 ORB Interface

8.1 Overview

This clause introduces the operations that are implemented by the ORB core, and describes some basic ones, while providing reference to the description of the remaining operations that are described elsewhere. The **ORB** interface is the interface to those ORB functions that do not depend on which object adapter is used. These operations are the same for all ORBs and all object implementations, and can be performed either by clients of the objects or implementations. The **Object** interface contains operations that are implemented by the ORB, and are accessed as implicit operations of the Object Reference. The **ValueBase** interface contains operations that are implemented by the ORB, and are accessed as implicit operations of the **ValueBase** Reference.

Because the operations in this sub clause are implemented by the ORB itself, they are not in fact operations on objects, although they are described that way for the **Object** or **ValueBase** interface operations and the language binding will, for consistency, make them appear that way.

8.2 The ORB Operations

The **ORB** interface contains the operations that are available to both clients and servers. These operations do not depend on any specific object adapter or any specific object reference.

```
module CORBA {

    interface NVList;          // forward declaration
    interface OperationDef;    // forward declaration
    interface TypeCode;        // forward declaration

    typedef short PolicyErrorCode;

    // for the definition of consts see PolicyErrorCode on page 128

    typedef unsigned long PolicyType;

    interface Request;                  // forward declaration
    typedef sequence <Request> RequestSeq;

    native AbstractBase;

    exception PolicyError {PolicyErrorCode reason;};

    typedef string RepositoryId;
    typedef string Identifier;

    // StructMemberSeq defined in Chapter 10
    // UnionMemberSeq defined in Chapter 10
    // EnumMemberSeq defined in Chapter 10

    typedef unsigned short ServiceType;
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
typedef unsigned long ServiceOption;
typedef unsigned long ServiceDetailType;

typedef CORBA::OctetSeq ServiceDetailData;
typedef sequence<ServiceOption> ServiceOptionSeq;

const ServiceType Security = 1;

struct ServiceDetail {
    ServiceDetailType service_detail_type;
    ServiceDetailData service_detail;
};

typedef sequence<ServiceDetail> ServiceDetailSeq;

struct ServiceInformation {
    ServiceOptionSeq service_options;
    ServiceDetailSeq service_details;
};

native ValueFactory;

typedef string ORBid;

interface ORB {

    typedef string ObjectId;
    typedef sequence <ObjectId> ObjectIdList;

    exception InvalidName {};

    ORBid id();

    string object_to_string (
        in Object           obj
    );

    Object string_to_object (
        in string           str
    );

    // Dynamic Invocation related operations

    void create_list (
        in long             count,
        out NVList          new_list
    );

    void create_operation_list (
        in OperationDef     oper,
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    out NVList              new_list
);

void get_default_context (
    out Context             ctx
);

void send_multiple_requests_oneway(
    in RequestSeq       req
);

void send_multiple_requests_deferred(
    in RequestSeq       req
);

boolean poll_next_response();

void get_next_response(
    out Request             req
) raises (WrongTransaction);

// Service information operations

boolean get_service_information (
    in ServiceType service_type,
    out ServiceInformation service_information
);

ObjectIdList list_initial_services ();

// Initial reference operation

Object resolve_initial_references (
    in ObjectId identifier
) raises (InvalidName);

// Type code creation operations

TypeCode create_struct_tc (
    in RepositoryId id,
    in Identifier name,
    in StructMemberSeq members
);

TypeCode create_union_tc (
    in RepositoryId id,
    in Identifier name,
    in TypeCode discriminator_type,
    in UnionMemberSeq members
);
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

```
TypeCode create_enum_tc (
    in RepositoryId id,
    in Identifier name,
    in EnumMemberSeq members
);

TypeCode create_alias_tc (
    in RepositoryId id,
    in Identifier name,
    in TypeCode original_type
);

TypeCode create_exception_tc (
    in RepositoryId id,
    in Identifier name,
    in StructMemberSeq members
);

TypeCode create_interface_tc (
    in RepositoryId id,
    in Identifier name
);

TypeCode create_string_tc (
    in unsigned long bound
);

TypeCode create_wstring_tc (
    in unsigned long bound
);

TypeCode create_fixed_tc (
    in unsigned short digits,
    in short scale
);

TypeCode create_sequence_tc (
    in unsigned long bound,
    in TypeCode element type
);

TypeCode create_recursive_sequence_tc (// deprecated
    in unsigned long bound,
    in unsigned long offset
);

TypeCode create_array_tc (
    in unsigned long length,
    in TypeCode element_type
);
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
TypeCode create_value_tc (
    in RepositoryId         id,
    in Identifier           name,
    in ValueModifier        type_modifier,
    in TypeCode             concrete_base,
    in ValueMembersSeq      members
);

TypeCode create_value_box_tc (
    in RepositoryId         id,
    in Identifier           name,
    in TypeCode             boxed_type
);

TypeCode create_native_tc (
    in RepositoryId         id,
    in Identifier           name
);

TypeCode create_recursive_tc(
    in RepositoryId         id
);

TypeCode create_abstract_interface_tc(
    in RepositoryId         id,
    in Identifier           name
);

TypeCode create_local_interface_tc(
    in RepositoryId         id,
    in Identifier           name
);

TypeCode create_component_tc (
    in RepositoryId         id,
    in Identifier           name
);

TypeCode create_home_tc (
    in RepositoryId         id,
    in Identifier           name
);

TypeCode create_event_tc (
    in RepositoryId         id,
    in Identifier           name,
    in ValueModifier        type_modifier,
    in TypeCode             concrete_base,
    in ValueMemberSeq       members
);
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        // Thread related operations

        boolean work_pending( );

        void perform_work();

        void run();

        void shutdown(
            in boolean          wait_for_completion
        );

        void destroy();

        // Policy related operations

        Policy create_policy(
            in PolicyType       type,
            in any              val
        ) raises (PolicyError);

        // Dynamic Any related operations deprecated and removed
        // from primary list of ORB operations

        // Value factory operations

        ValueFactory register_value_factory(
            in RepositoryId id,
            in ValueFactory_factory
        );

        void unregister_value_factory(in RepositoryId id);

        ValueFactory lookup_value_factory(in RepositoryId id);

        void register_initial_reference(
            in ObjectId id,
            in Object obj
        ) raises (InvalidName);
    };
};
```

All types defined in this clause are part of the CORBA module. When referenced in OMG IDL, the type names must be prefixed by "**CORBA::**".

The operations **object_to_string** and **string_to_object** are described in Converting Object References to Strings on page 101.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

For a description of the **create_list** and **create_operation_list** operations, see Polling on page 186. The **get_default_context** operation is described in get_default_context on page 102. The **send_multiple_requests_oneway** and **send_multiple_requests_deferred** operations are described in send_multiple_requests on page 185. The **poll_next_response** and **get_next_response** operations are described in get_next_response and poll_next_response on page 185.

The **list_intial_services** and **resolve_initial_references** operations are described in Obtaining Initial Object References on page 117.

The Type code creation operations with names of the form **create_<type>_tc** are described in Creating TypeCodes on page 143.

The **work_pending**, **perform_work**, **shutdown, destroy** and **run** operations are described in Thread-Related Operations on page 102.

The **create_policy** operations is described in Create_policy on page 128.

The **register_value_factory**, **unregister_value_factory** and **lookup_value_factory** operations are described in Language Specific Value Factory Requirements on page 163.

The **register_initial_reference** operation is described in register_initial_reference on page 409.

8.2.1 ORB Identity

8.2.1.1 id

```
ORBid id();
```

The **id** operation returns the identity of the ORB. The returned **ORBid** is the string that was passed to **ORB_init** (see ORB Initialization on page 115) as the **orb_identifier** parameter when the ORB was created. If that was the empty string, the returned string is the value associated with the **-ORBid** tag in the **arg_list** parameter. Calling **id** on the default ORB returns the empty string.

8.2.2 Converting Object References to Strings

8.2.2.1 object_to_string

```
string object_to_string (
    in Object           obj
);
```

8.2.2.2 string_to_object

```
Object string_to_object (
    in string           str
);
```

Because an object reference is opaque and may differ from ORB to ORB, the object reference itself is not a convenient value for storing references to objects in persistent storage or communicating references by means other than invocation. Two problems must be solved: allowing an object reference to be turned into a value that a client can store in some other medium, and ensuring that the value can subsequently be turned into the appropriate object reference.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

An object reference may be translated into a string by the operation **object_to_string**. The value may be stored or communicated in whatever ways strings may be manipulated. Subsequently, the **string_to_object** operation will accept a string produced by **object_to_string** and return the corresponding object reference.

To guarantee that an ORB will understand the string form of an object reference, that ORB's **object_to_string** operation must be used to produce the string. For all conforming ORBs, if **obj** is a valid reference to an object, then **string_to_object(object_to_string(obj))** will return a valid reference to the same object, if the two operations are performed on the same ORB. For all conforming ORB's supporting IOP, this remains true even if the two operations are performed on different ORBs.

8.2.3 Getting Service Information

8.2.3.1 get_service_information

```
boolean get_service_information (
    in ServiceType service_type;
    out ServiceInformation service_information;
);
```

The **get_service_information** operation is used to obtain information about CORBA facilities and services that are supported by this ORB. The service type for which information is being requested is passed in as the in parameter **service_type**, the values defined by constants in the **CORBA** module. If service information is available for that type, that is returned in the out parameter **service_information**, and the operation returns the value **TRUE**. If no information for the requested services type is available, the operation returns **FALSE** (i.e., the service is not supported by this ORB).

8.2.4 Creating a New Context

8.2.4.1 get_default_context

```
void get_default_context(       // PIDL
    out Context     ctx         // context object
);
```

This operation creates a new empty Context object every time it is called. The operation is defined in the **ORB** interface.

8.2.5 Thread-Related Operations

To support single-threaded ORBs, as well as multi-threaded ORBs that run multi-thread-unaware code, several operations are included in the **ORB** interface. These operations can be used by single-threaded and multi-threaded applications. An application that is a pure ORB client would not need to use these operations. Both the **ORB::run** and **ORB::shutdown** are useful in fully multi-threaded programs.

These operations are defined on the ORB rather than on an object adapter to allow the main thread to be used for all kinds of asynchronous processing by the ORB. Defining these operations on the ORB also allows the ORB to support multiple object adapters, without requiring the application main to know about all the object adapters. The interface between the ORB and an object adapter is not standardized.

8.2.5.1 work_pending

```
boolean work_pending( );
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This operation returns an indication of whether the ORB needs the main thread to perform some work.

A result of TRUE indicates that the ORB needs the main thread to perform some work and a result of FALSE indicates that the ORB does not need the main thread.

8.2.5.2 perform_work

void perform_work();

If called by the main thread, this operation performs an implementation-defined unit of work; otherwise, it does nothing.

It is platform-specific how the application and ORB arrange to use compatible threading primitives.

The **work_pending()** and **perform_work()** operations can be used to write a simple polling loop that multiplexes the main thread among the ORB and other activities. Such a loop would most likely be needed in a single-threaded server. A multi-threaded server would need a polling loop only if there were both ORB and other code that required use of the main thread.

Here is an example of such a polling loop:

```
// C++
   for (;;) {
         if (orb->work_pending()) {
            orb->perform_work();
         };
         // do other things
         // sleep?
   };
```

Once the ORB has shutdown, **work_pending** and **perform_work** will raise the BAD_INV_ORDER exception with minor code 4. An application can detect this exception to determine when to terminate a polling loop.

8.2.5.3 run

void run();

This operation provides execution resources to the ORB so that it can perform its internal functions. Single threaded ORB implementations, and some multi-threaded ORB implementations, need the use of the main thread in order to function properly. For maximum portability, an application should call either **run** or **perform_work** on its main thread. **run** may be called by multiple threads simultaneously.

This operation will block until the ORB has completed the shutdown process, initiated when some thread calls **shutdown**.

8.2.5.4 shutdown

```
void shutdown(
    in boolean          wait_for_completion
);
```

This operation instructs the ORB to shut down, that is, to stop processing in preparation for destruction.

Shutting down the ORB causes all object adapters to be destroyed, since they cannot exist in the absence of an ORB.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

In the case of the **POA**, all **POAManager**s are deactivated prior to destruction of all POAs. The deactivation that the ORB performs should be the equivalent of calling deactivate with the value **TRUE** for **etherealize_objects** and with the **wait_for_completion** parameter same as what **shutdown** was called with.

Shut down is complete when all ORB processing (including request processing and object deactivation or other operations associated with object adapters) has completed and the object adapters have been destroyed. In the case of the **POA**, this means that all object etherealizations have finished and root **POA** has been destroyed (implying that all descendent **POA**s have also been destroyed).

Shut down is complete when all **ORB** processing has completed and the object adapters have been destroyed. **ORB** processing is defined as including request processing and object deactivation or other operations associated with object adapters, and the forwarding of the responses from deferred synchronous invocations to their associated reply handlers. In the case of the **POA**, this means that all object etherealizations have finished and root POA has been destroyed (implying that all descendent **POA**s have also been destroyed)

If the **wait_for_completion** parameter is **TRUE**, this operation blocks until the shut down is complete. If an application does this in a thread that is currently servicing an invocation, the ORB will not shutdown, and the BAD_INV_ORDER system exception will be raised with the OMG minor code 3, and completion status COMPLETED_NO, since blocking would result in a deadlock.

If the **wait_for_completion** parameter is **FALSE**, then **shutdown** may not have completed upon return. An ORB implementation may require the application to call (or have a pending call to) **run** or **perform_work** after **shutdown** has been called with its parameter set to **FALSE**, in order to complete the shutdown process.

Additionally in systems that have Portable Object Adapters (see Clause 14) **shutdown** behaves as if **POA::destroy** is called on the Root **POA** with its first parameter set to TRUE and the second parameter set to the value of the **wait_for_completion** parameter that **shutdown** is invoked with.

While the ORB is in the process of shutting down, the ORB operates as normal, servicing incoming and outgoing requests until all requests have been completed. An implementation may impose a time limit for requests to complete while a **shutdown** is pending.

Once an ORB has shutdown, only object reference management operations(**duplicate**, **release** and **is_nil**) may be invoked on the ORB or any object reference obtained from it. An application may also invoke the destroy operation on the ORB itself. Invoking any other operation will raise the BAD_INV_ORDER system exception with the OMG minor code 4.

8.2.5.5 destroy

void destroy();

This operation destroys the ORB so that its resources can be reclaimed by the application. Any operation invoked on a destroyed ORB reference will raise the OBJECT_NOT_EXIST exception. Once an ORB has been destroyed, another call to **ORB_init** with the same **ORBid** will return a reference to a newly constructed ORB.

If **destroy** is called on an ORB that has not been shut down, it will start the shut down process and block until the ORB has shut down before it destroys the ORB. The behavior is similar to that achieved by calling **shutdown** with the **wait_for_completion** parameter set to **TRUE**. If an application calls **destroy** in a thread that is currently servicing an invocation, the BAD_INV_ORDER system exception will be raised with the OMG minor code 3, since blocking would result in a deadlock.

For maximum portability and to avoid resource leaks, an application should always call **shutdown** and **destroy** on all ORB instances before exiting.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.3 Object Reference Operations

There are some operations that can be done on any object. These are not operations in the normal sense, in that they are implemented directly by the ORB, not passed on to the object implementation. We will describe these as being operations on the object reference, although the interfaces actually depend on the language binding. As above, where we used interface **Object** to represent the object reference, we define an interface for **Object**:

```
module CORBA {

    interface DomainManager;          // forward declaration
    typedef sequence <DomainManager> DomainManagersList;

    interface Policy;                 // forward declaration
    typedef sequence <Policy> PolicyList;
    typedef sequence<PolicyType> PolicyTypeSeq;
    exception InvalidPolicies { sequence <unsigned short> indices; };

    interface Context;                // forward declaration

    typedef string Identifier;
    interface Request;                // forward declaration
    interface NVList;                 // forward declaration
    struct NamedValue{};              // an implicitly well known type
    typedef unsigned long Flags;
    interface InterfaceDef;

    enum SetOverrideType {SET_OVERRIDE, ADD_OVERRIDE};

    interface ORB;       // PIDL forward declaration

    interface Object {                                  // PIDL

        InterfaceDef get_interface ();

        boolean is_nil();

        Object duplicate ();

        void release ();

        boolean is_a (
            in RepositoryId          logical_type_id
        );

        boolean non_existent();

        boolean is_equivalent (
            in Object                other_object
        );
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        unsigned long hash(
            in unsigned long        maximum
        );

        void create_request (
            in Context              ctx
            in Identifier           operation,
            in NVList               arg_list,
            inout NamedValue        result,
            out Request             req,
            in Flags                req_flag
        );

        Policy get_policy (
            in PolicyType           policy_type
        );

        DomainManagersList get_domain_managers ();

        Object set_policy_overrides(
            in PolicyList           policies,
            in SetOverrideType      set_add
        ) raises (InvalidPolicies);

        Policy get_client_policy(
            in PolicyType type
        );

        PolicyList get_policy_overrides(
            in PolicyTypeSeq        types
        );

        boolean validate_connection(
            out PolicyList           inconsistent_policies
        );

        Object get_component ();

        string respository_id();

        ORB get_orb();
    };
};
```

The **create_request** operation is part of the Object interface because it creates a pseudo-object (a Request) for an object. It is described with the other Request operations in Request Operations on page 179.

Unless otherwise stated below, the operations in the IDL above do not require access to remote information.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.3.1 Determining the Object Interface

8.3.1.1 get_interface

```
InterfaceDef get_interface();
```

get_interface, returns an object in the Interface Repository that describes the most derived type of the object addressed by the reference. See the Interface Repository clause for a definition of operations on the Interface Repository. The implementation of this operation may involve contacting the ORB that implements the target object.

If the interface repository is not available, **get_interface** raises INTF_REPOS with standard minor code 1. If the interface repository does not contain an entry for the object's (most derived) interface, **get_interface** raises INTF_REPOS with standard minor code 2.

8.3.1.2 repository_id

repository_id returns the repository ID of an object (see Component Interface Repository Interfaces on page 262 for details of repository IDs). The implementation of this operation must contact the ORB that implements the target object.

8.3.2 Duplicating and Releasing Copies of Object References

8.3.2.1 duplicate

```
Object duplicate();
```

8.3.2.2 release

```
void release();
```

Because object references are opaque and ORB-dependent, it is not possible for clients or implementations to allocate storage for them. Therefore, there are operations defined to copy or release an object reference.

If more than one copy of an object reference is needed, the client may create a **duplicate**. Note that the object implementation is not involved in creating the duplicate, and that the implementation cannot distinguish whether the original or a duplicate was used in a particular request.

When an object reference is no longer needed by a program, its storage may be reclaimed by use of the **release** operation. Note that the object implementation is not involved, and that neither the object itself nor any other references to it are affected by the **release** operation.

8.3.3 Nil Object References

8.3.3.1 is_nil

```
boolean is_nil();
```

An object reference whose value is **OBJECT_NIL** denotes no object. An object reference can be tested for this value by the **is_nil** operation. The object implementation is not involved in the nil test.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.3.4 Equivalence Checking Operation

8.3.4.1 is_a

```
boolean is_a(
    in RepositoryId          logical_type_id
);
```

An operation is defined to facilitate maintaining type-safety for object references over the scope of an ORB.

The **logical_type_id** is a string denoting a shared type identifier (**RepositoryId**). The operation returns true if the object is really an instance of that type, including if that type is an ancestor of the "most derived" type of that object.

Determining whether an object's type is compatible with the **logical_type_id** may require contacting a remote ORB or interface repository. Such an attempt may fail at either the local or the remote end. If **is_a** cannot make a reliable determination of type compatibility due to failure, it raises an exception in the calling application code. This enables the application to distinguish among the **TRUE**, **FALSE**, and indeterminate cases.

This operation exposes to application programmers functionality that must already exist in ORBs that support "type safe narrow" and allows programmers working in environments that do not have compile time type checking to explicitly maintain type safety.

This operation always returns **TRUE** for the **logical_type_id IDL:omg.org/CORBA/Object:1.0**

8.3.5 Probing for Object Non-Existence

8.3.5.1 non_existent

```
boolean non_existent ();
```

The **non_existent** operation may be used to test whether an object (e.g., a proxy object) has been destroyed. It does this without invoking any application level operation on the object, and so will never affect the object itself. It returns true (rather than raising **CORBA::OBJECT_NOT_EXIST**) if the ORB knows authoritatively that the object does not exist; otherwise, it returns false.

Services that maintain state that includes object references, such as bridges, event channels, and base relationship services, might use this operation in their "idle time" to sift through object tables for objects that no longer exist, deleting them as they go, as a form of garbage collection. In the case of proxies, this kind of activity can cascade, such that cleaning up one table allows others then to be cleaned up.

Probing for object non-existence may require contacting the ORB that implements the target object. Such an attempt may fail at either the local or the remote end. If non-existent cannot make a reliable determination of object existence due to failure, it raises an exception in the calling application code. This enables the application to distinguish among the true, false, and indeterminate cases.

8.3.6 Object Reference Identity

In order to efficiently manage state that include large numbers of object references, services need to support a notion of object reference identity. Such services include not just bridges, but relationship services and other layered facilities.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Two identity-related operations are provided. One maps object references into disjoint groups of potentially equivalent references, and the other supports more expensive pairwise equivalence testing. Together, these operations support efficient maintenance and search of tables keyed by object references.

8.3.6.1 Hashing Object Identifiers

hash

```
unsigned long hash(
    in unsigned long        maximum
);
```

Object references are associated with ORB-internal identifiers that may indirectly be accessed by applications using the **hash** operation. The value of this identifier does not change during the lifetime of the object reference, and so neither will any hash function of that identifier.

The value of this operation is not guaranteed to be unique; that is, another object reference may return the same hash value. However, if two object references hash differently, applications can determine that the two object references are *not* identical.

The **maximum** parameter to the **hash** operation specifies an upper bound on the hash value returned by the ORB. The lower bound of that value is zero. Since a typical use of this feature is to construct and access a collision chained hash table of object references, the more randomly distributed the values are within that range, and the cheaper those values are to compute, the better.

For bridge construction, note that proxy objects are themselves objects, so there could be many proxy objects representing a given “real” object. Those proxies would not necessarily hash to the same value.

8.3.6.2 Equivalence Testing

is_equivalent

```
boolean is_equivalent(
    in Object               other_object
);
```

The **is_equivalent** operation is used to determine if two object references are equivalent, so far as the ORB can easily determine. It returns **TRUE** if the target object reference is known to be equivalent to the other object reference passed as its parameter, and **FALSE** otherwise.

If two object references are identical, they are equivalent. Two different object references that in fact refer to the same object are also equivalent.

ORBs are allowed, but not required, to attempt determination of whether two distinct object references refer to the same object. In general, the existence of reference translation and encapsulation, in the absence of an omniscient topology service, can make such determination impractically expensive. This means that a **FALSE** return from **is_equivalent** should be viewed as only indicating that the object references are distinct, and not necessarily an indication that the references indicate distinct objects. Setting of local policies on the object reference is not taken into consideration for the purposes of determining object reference equivalence.

A typical application use of this operation is to match object references in a hash table. Bridges could use it to shorten the lengths of chains of proxy object references. Externalization services could use it to “flatten” graphs that represent cyclical relationships between objects. Some might do this as they construct the table, others during idle time.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.3.7 Type Coercion Considerations

Many programming languages map **Object** to programming constructs that support inheritance. Mappings to languages (such as C++ and Java) typically provide a mechanism for narrowing (down-casting) an object reference from a base interface to a more derived interface. To do such down-casting in a type safe way, knowledge of the full inheritance hierarchy of the target interface may be required. The implementation of down-cast must either contact an interface repository or the target itself, to determine whether or not it is safe to down-cast the client's object reference. This requirement is not acceptable when a client is expecting only asynchronous communication with the target. Therefore, for the appropriate languages an unchecked down-cast operation (also referred to as unchecked narrow operation) shall be provided in the mapping of Object. This unchecked narrow always returns a stub of the requested type without checking that the target really implements that interface.

8.3.8 Getting Policy Associated with the Object

8.3.8.1 get_policy

The get_policy operation returns the policy object of the specified type (see Policy Object on page 126), which applies to this object. It returns the *effective* **Policy** for the object reference. The effective **Policy** is the one that would be used if a request were made.

This **Policy** is determined first by obtaining the effective override for the **PolicyType** as returned by **get_client_policy**. The effective override is then compared with the **Policy** as specified in the **IOR**. The effective **Policy** is determined by reconciling the effective override and the **IOR**-specified **Policy** (see Server Side Policy Management on page 132). If the two policies cannot be reconciled, the standard system exception INV_POLICY is raised with standard minor code 1. The absence of a **Policy** value in the **IOR** implies that any legal value may be used.

Invoking **non_existent** on an object reference prior to **get_policy** ensures the accuracy of the returned effective **Policy**. If **get_policy** is invoked prior to the object reference being bound, a compliant implementation shall attempt a binding and then return the effective **Policy**. If the binding attempt fails it shall pass through the system exception returned from the binding attempt. Note that if the effective **Policy** may change from invocation to invocation due to transparent rebinding.

```
Policy get_policy (
    in PolicyType       policy_type
);
```

Parameter(s)

- **policy_type**
  The type of policy to be obtained.

Return Value

A **Policy** object of the type specified by the **policy_type** parameter.

Exception(s)

- *CORBA::INV_POLICY*
  Raised when the value of policy type is not valid either because the specified type is not supported by this ORB or because a policy object of that type is not associated with this Object.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The implementation of this operation may involve remote invocation of an operation (e.g., **DomainManager::get_domain_policy** for some security policies) for some policy types.

8.3.8.2 get_client_policy

```
Policy get_client_policy(
    in PolicyType type
);
```

Returns the *effective overriding* Policy for the object reference. The effective override is obtained by first checking for an override of the given PolicyType at the Object scope, then at the Current scope, and finally at the ORB scope. If no override is present for the requested PolicyType, a system-dependent default value for that Policy Type may be returned. A nil Policy reference may also be returned to indicate that there is no default for the policy. Portable applications are expected to set the desired "defaults" at the ORB scope since default Policy values are not specified.

8.3.8.3 get_policy_overrides

```
PolicyList get_policy_overrides(
    in PolicyTypeSeq    types
);
```

Returns the list of Policy overrides (of the specified policy types) set at the Object scope. If the specified sequence is empty, all Policy overrides at this scope will be returned. If none of the requested PolicyTypes are overridden at the Object scope, an empty sequence is returned.

8.3.9 Overriding Associated Policies on an Object Reference

8.3.9.1 set_policy_overrides

The **set_policy_overrides** operation returns a new object reference with the new policies associated with it. It takes two input parameters. The first parameter **policies** is a sequence of references to **Policy** objects. The second parameter **set_add** of type **SetOverrideType** indicates whether these policies should be added onto any other overrides that already exist (**ADD_OVERRIDE**) in the object reference, or they should be added to a clean override free object reference (**SET_OVERRIDE**). This operation associates the policies passed in the first parameter with a newly created object reference that it returns. Only certain policies that pertain to the invocation of an operation at the client end can be overridden using this operation. Attempts to override any other policy will result in the raising of the CORBA::NO_PERMISSION exception.

```
enum SetOverrideType {SET_OVERRIDE, ADD_OVERRIDE};

Object set_policy_overrides(
    in PolicyList           policies,
    in SetOverrideType      set_add
) raises (InvalidPolicies);
```

Parameter(s)

- **policies**
  A sequence of **Policy** objects that are to be associated with the new copy of the object reference returned by this operation. If the sequence contains two or more **Policy** objects with the same **PolicyType** value, the operation raises the standard system exception BAD_PARAM with minor code 30.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- **set_add**
  Whether the association is in addition to (**ADD_OVERRIDE**) or as a replacement of (**SET_OVERRIDE**) any existing overrides already associated with the object reference. If the value of this parameter is **SET_OVERRIDE**, the supplied **policies** completely replace all existing overrides associated with the object reference. If the value of this parameter is **ADD_OVERRIDE**, the supplied **policies** are added to the existing overrides associated with the object reference, except that if a supplied **Policy** object has the same **PolicyType** value as an existing override, the supplied **Policy** object replaces the existing override.

Return Value

A copy of the object reference with the overrides from **policies** associated with it in accordance with the value of **set_add**.

Exception(s)

- InvalidPolicies
  Raised when an attempt is made to override any policy that cannot be overridden.

8.3.10 Validating Connection

8.3.10.1 validate_connection

```
boolean validate_connection(
    out PolicyList          inconsistent_policies
);
```

Returns the value TRUE if the current effective policies for the Object will allow an invocation to be made. If the object reference is not yet bound, a binding will occur as part of this operation. If the object reference is already bound, but current policy overrides have changed or for any other reason the binding is no longer valid, a rebind will be attempted regardless of the setting of any RebindPolicy override. The validate_connection operation is the only way to force such a rebind when implicit rebinds are disallowed by the current effective RebindPolicy. The attempt to bind or rebind may involve processing GIOP LocateRequests by the ORB.

If the RoutingPolicy **ROUTE_FORWARD** or **ROUTE_STORE_AND_FORWARD** are in effect when **validate_connection** is invoked then the client ORB shall attempt to open a connection for the first hop to the first target **Router** (applies to both **Router** and **PersistentRequestRouter**) as if it were the target **Object** and return success or failure based on success or failure to establish this connection.

Returns the value FALSE if the current effective policies would cause an invocation to raise the standard system exception INV_POLICY. If the current effective policies are incompatible, the out parameter inconsistent_policies contains those policies causing the incompatibility. This returned list of policies is not guaranteed to be exhaustive. If the binding fails due to some reason unrelated to policy overrides, the appropriate standard system exception is raised.

8.3.11 Getting the Domain Managers Associated with the Object

8.3.11.1 get_domain_managers

The get_domain_managers operation allows administration services (and applications) to retrieve the domain managers (see Management of Policies on page 131), and hence the security and other policies applicable to individual objects that are members of the domain.

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

typedef sequence <DomainManager> DomainManagersList;

DomainManagersList get_domain_managers ();

Return Value

The list of immediately enclosing domain managers of this object. At least one domain manager is always returned in the list since by default each object is associated with at least one domain manager at creation.

The implementation of this operation may involve contacting the ORB that implements the target object.

8.3.12 Getting Component Associated with the Object

8.3.12.1 get_component

Object get_component ();

If the target object reference is itself a component reference (i.e., it denotes the component itself), the **get_component** operation returns the same reference (or another equivalent reference). If the target object reference is a facet reference the **get_component** operation returns an object reference for the component. If the target reference is neither a component reference nor a provided reference, **get_component** returns a nil reference.

8.3.13 Getting the ORB

8.3.13.1 get_orb

ORB get_orb();

This operation returns the local ORB that is handling this particular Object Reference.

8.3.14 LocalObject Operations

Local interfaces are implemented by using **CORBA::LocalObject**, which derives from **CORBA::Object** and provides implementations of Object pseudo operations and any other ORB specific support mechanisms that are appropriate for such objects. Object implementation techniques are inherently language mapping specific. Therefore, the **LocalObject** type is not defined in IDL, but is specified by each language mapping.

- The **LocalObject** type provides implementations of the following **Object** pseudo-operations that raise the NO_IMPLEMENT system exception with standard minor code 8:
  - **get_interface**
  - **get_domain_managers**
  - **get_policy**
  - **get_client_policy**
  - **set_policy_overrides**
  - **get_policy_overrides**
  - **validate_connection**
  - **get_component**
  - **respository_id**

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- The **LocalObject** type provides implementations of the following pseudo-operations:
  - **non_existent** - always returns false.
  - **hash** - returns a hash value that is consistent for the lifetime of the object.
  - **is_equivalent** - returns true if the references refer to the same LocalObject implementation.
  - **is_a** - returns **TRUE** if the **LocalObject** derives from or is itself the type specified by the **logical_type_id** argument.
  - **get_orb** - The default behavior of this operation when invoked on a reference to a local object is to return the system exception NO_IMPLEMENT with standard minor code 8. Certain local objects that have close association with an ORB, like POAs, Current objects and certain portable interceptors related local objects override the default behavior and return a reference to the ORB that they are associated with. These are documented in the sub clauses where these local objects are specified
- Attempting to use a **LocalObject** to create a DII request shall result in a NO_IMPLEMENT system exception with standard minor code 4. Attempting to marshal or stringify a LocalObject shall result in a MARSHAL system exception with standard minor code 4. Narrowing and widening of references to **LocalObject**s must work as for regular object references.
- Local types cannot be marshaled and references to local objects cannot be converted to strings. Any attempt to marshal a local object, such as via an unconstrained base interface, as an **Object**, or as the contents of an **any**, or to pass a local object to **ORB::object_to_string**, shall result in a MARSHAL system exception with OMG minor code 4.
- The DII is not supported on local objects, nor are asynchronous invocation interfaces.
- Language mappings shall specify server side mechanisms, including base classes and/or skeletons if necessary, for implementing local objects, so that invocation overhead is minimized.
- The usage of client side language mappings for local types shall be identical to those of equivalent unconstrained types.
- Invocations on local objects are not ORB mediated. Specifically, parameter copy semantics are not honored, interceptors are not invoked, and the execution context of a local object does not have ORB service **Current** object contexts that are distinct from those of the caller. Implementations of local interfaces are responsible for providing the parameter copy semantics expected by clients.
- Local objects have no inherent identities beyond their implementations' identities as programming objects. The lifecycle of the implementation is the same as the lifecycle of the reference.
- Instances of local objects defined as part of OMG specifications to be supplied by ORB products or object service products shall be exposed through the **ORB::resolve_initial_references** operation or through some other local object obtained from **resolve_initial_references**.

8.4 ValueBase Operations

ValueBase serves a similar role for value types that **Object** serves for interfaces. Its mapping is language-specific and must be explicitly specified for each language.

Typically it is mapped to a concrete language type which serves as a base for all value types. Any operations that are required to be supported for all values are conceptually defined on **ValueBase**, although in reality their actual mapping depends upon the specifics of any particular language mapping.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Analogous to the definition of the **Object** interface for implicit operations of object references, the implicit operations of **ValueBase** are defined on a pseudo-**valuetype** as follows:

```
module CORBA {
    valuetype ValueBase{                                PIDL
        ValueDef get_value_def();
    };
};
```

The **get_value_def()** operation returns a description of the value's definition as described in the interface repository (ValueDef on page 257).

8.5 ORB and OA Initialization and Initial References

Before an application can enter the CORBA environment, it must first:

- Be initialized into the ORB and possibly the object adapter (POA) environments.
- Get references to ORB pseudo-object (for use in future ORB operations) and perhaps other objects (including the root POA or some Object Adapter objects).

The following operations are provided to initialize applications and obtain the appropriate object references:

- Operations providing access to the ORB. These operations reside in the CORBA module, but not in the ORB interface and are described in ORB Initialization on page 115.
- Operations providing access to Object Adapters, Interface Repository, Naming Service, and other Object Services. These operations reside in the ORB interface and are described in Obtaining Initial Object References on page 117.

8.5.1 ORB Initialization

When an application requires a CORBA environment it needs a mechanism to get the ORB pseudo-object reference and possibly an OA object reference (such as the root POA). This serves two purposes. First, it initializes an application into the ORB and OA environments. Second, it returns the ORB pseudo-object reference and the OA object reference to the application for use in future ORB and OA operations.

The ORB and OA initialization operations must be ordered with ORB occurring before OA: an application cannot call OA initialization routines until ORB initialization routines have been called for the given ORB. The operation to initialize an application in the ORB and get its pseudo-object reference is not performed on an object. This is because applications do not initially have an object on which to invoke operations. The ORB initialization operation is an application's bootstrap call into the CORBA world. The **ORB_init** call is part of the CORBA module but not part of the ORB interface.

Applications can be initialized in one or more ORBs. When an ORB initialization is complete, its pseudo reference is returned and can be used to obtain other references for that ORB.

In order to obtain an **ORB** pseudo-object reference, applications call the **ORB_init** operation. The parameters to the call comprise an identifier for the ORB for which the pseudo-object reference is required, and an **arg_list**, which is used to allow environment-specific data to be passed into the call. PIDL for the ORB initialization is as follows:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
// PIDL
module CORBA {
    typedef sequence <string> arg_list;
    ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);
};
```

The identifier for the ORB will be a name of type **CORBA::ORBid**. All **ORBid** strings other than the empty string are allocated by ORB administrators and are not managed by the OMG. ORB administration is the responsibility of each ORB supplier. ORB suppliers may optionally delegate this responsibility. **ORBid** strings other than the empty string are intended to be used to uniquely identify each ORB used within the same address space in a multi-ORB application. These special **ORBid** strings are specific to each ORB implementation and the ORB administrator is responsible for ensuring that the names are unambiguous.

If an empty **ORBid** string is passed to **ORB_init**, then the **arg_list** arguments shall be examined to determine if they indicate an ORB reference that should be returned. This is achieved by searching the **arg_list** parameters for one preceded by "**-ORBid**" for example, "**-ORBid example_orb**" (the white space after the "**-ORBid**" tag is ignored) or "**-ORBidMyFavoriteORB**" (with no white space following the "**-ORBid**" tag). Alternatively, two sequential parameters with the first being the string "**-ORBid**" indicates that the second is to be treated as an **ORBid** parameter. If an empty string is passed and no **arg_list** parameters indicate the ORB reference to be returned, the default ORB for the environment will be returned.

Other parameters of significance to the ORB can also be identified in **arg_list**, for example, "**Hostname**," "**SpawnedServer**," and so forth. To allow for other parameters to be specified without causing applications to be re-written, it is necessary to specify the parameter format that ORB parameters may take. In general, parameters shall be formatted as either one single **arg_list** parameter:

–ORB<suffix><optional white space> <value>

or as two sequential **arg_list** parameters:

-ORB<suffix>

<value>

Regardless of whether an empty or non-empty **ORBid** string is passed to **ORB_init**, the **arg_list** arguments are examined to determine if any ORB parameters are given. If a non-empty **ORBid** string is passed to **ORB_init**, all **ORBid** parameters in the **arg_list** are ignored. All other **-ORB<suffix>** parameters in the **arg_list** may be of significance during the ORB initialization process.

Before **ORB_init** returns, it will remove from the **arg_list** parameter all strings that match the **-ORB<suffix>** pattern described above and that are recognized by that ORB implementation, along with any associated sequential parameter strings. If any strings in **arg_list** that match this pattern are not recognized by the ORB implementation, **ORB_init** will raise the BAD_PARAM system exception instead.

The **ORB_init** operation may be called any number of times and shall return the same ORB reference when the same **ORBid** string is passed, either explicitly as an argument to **ORB_init** or through the **arg_list**. All other **-ORB<suffix>** parameters in the **arg_list** may be considered on subsequent calls to **ORB_init**.

NOTE: Whenever an **ORB_init** argument of the form **-ORBxxx** is specified, it is understood that the argument may be represented in different ways in different languages. For example, in Java **-ORBxxx** is equivalent to a property named **org.omg.CORBA.ORBxxx**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.5.1.1 Server ID

A Server ID must uniquely identify a server to an IMR. This specification only requires unique identification using a string of some kind. We do not intend to make more specific requirements for the structure of a server ID.

The server ID may be specified by an **ORB_init** argument of the form

-ORBServerId

The value assigned to this property is a **string**. All templates created in this **ORB** will return this server ID in the **server_id** attribute.

It is required that all ORBs in the same server share the same server ID. Specific environments may choose to implement **-ORBServerId** in ways that automatically enforce this requirement.

For example, the **org.omg.CORBA.ServerId** system property may be set to the server ID in Java when a Java server is activated. This system property is then picked up as part of the **`ORB_init`** call for every **ORB** created in the server.

8.5.1.2 Server Endpoint

The server endpoint information is passed into **ORB_init** by an argument of the form

-ORBListenEndpoints <endpoints>

The format of the <endpoints> argument is proprietary. All that is required by this specification is that each time **ORB_init** is called with the same value for this argument, the resulting **ORB** will listen for requests on the same set of endpoints, so that persistent object references for the **ORB** will continue to function correctly.

8.5.1.3 Starting Servers with No Proprietary Server Activation Support

Any server started with the flag:

-ORBNoProprietaryActivation

shall avoid the use of any proprietary activation framework.

8.5.2 Obtaining Initial Object References

Applications require a portable means by which to obtain their initial object references. References are required for the root POA, POA Current, Interface Repository and various Object Services instances. (The POA is described in the Portable Object Adapter clause; the Interface Repository is described in the Interface Repository clause; Object Services are described in the individual service specifications.) The functionality required by the application is similar to that provided by the Naming Service. However, the OMG does not want to mandate that the Naming Service be made available to all applications in order that they may be portably initialized. Consequently, the operations shown in this sub clause provide a simplified, local version of the Naming Service that applications can use to obtain a small, defined set of object references that are essential to its operation. Because only a small well-defined set of objects are expected with this mechanism, the naming context can be flattened to be a single-level name space. This simplification results in only two operations being defined to achieve the functionality required.

Initial references are not obtained via a new interface; instead two operations are provided in the ORB pseudo-object interface, providing facilities to list and resolve initial object references.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

list_initial_services

```
typedef string ObjectId;
typedef sequence <ObjectId> ObjectIdList;
ObjectIdList list_initial_services ();
```

resolve_initial_references

```
exception InvalidName {};

Object resolve_initial_references (
    in ObjectId identifier
) raises (InvalidName);
```

The **resolve_initial_references** operation is an operation on the ORB rather than the Naming Service's **NamingContext**. The interface differs from the Naming Service's resolve in that **ObjectId** (a string) replaces the more complex Naming Service construct (a sequence of structures containing string pairs for the components of the name). This simplification reduces the name space to one context.

ObjectIds are strings that identify the object whose reference is required. To maintain the simplicity of the interface for obtaining initial references, only a limited set of objects are expected to have their references found via this route. Unlike the ORB identifiers, the **ObjectId** name space requires careful management. To achieve this, the OMG may, in the future, define which services are required by applications through this interface and specify names for those services.

resolve_initial_references never returns a **nil** reference. Instead, the non-availability of a particular reference is indicated by throwing an **InvalidName** exception (even if a **nil** reference is explicitly configured for an **ObjectId**).

Currently, reserved **ObjectIds** are **RootPOA**, **POACurrent**, **InterfaceRepository, NameService**, **TradingService**, **SecurityCurrent**, **TransactionCurrent, DynAnyFactory, ORBPolicyManager, PolicyCurrent, NotificationService, TypedNotificationService, CodecFactory, PICurrent, ComponentHomeFinder** and **PSS**.

Table 8.1- ObjectIds for resolve_initial_references

ObjectId	Type of Object Reference	Reference
RootPOA	PortableServer::POA	POA Interface on page 328.
POACurrent	PortableServer::Current	POA Interface on page 328.
InterfaceRepository	CORBA::Repository CORBA::ComponentIR::Repository	Repository on page 238 and ComponentIR::Repository on page 264.
NameService	CosNaming:: NamingContext	Naming Service specification (formal/00-06-19), the CosNaming Module sub clause.
TradingService	CosTrading::Lookup	Trading Object Service specification (formal/00-06-27), the Functional Interfaces sub clause.
SecurityCurrent	SecurityLevel1::Current or SecurityLevel2::Current	Security Service specification (formal/00-06-25), the Security Operations on Current sub clause.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 8.1- ObjectIds for resolve_initial_references

ObjectId	Type of Object Reference	Reference
TransactionCurrent	CosTransaction::Current	Transaction Service specification (formal/00-06-28), the Transaction Service Interfaces sub clause.
DynAnyFactory	DynamicAny:: DynAnyFactory	Creating a DynAny Object on page 204.
ORBPolicyManager	CORBA::PolicyManager	Policy Management Interfaces on page 132.
PolicyCurrent	CORBA::PolicyCurrent	Policy Management Interfaces on page 132.
NotificationService	CosNotifyChannelAdmin:: EventChannelFactory	Notification Service specification (formal/00-06-20)
TypedNotificationService	CosTypedNotifyChannelAdmin::Typed EventChannelFactory	Notification Service specification (formal/00-06-20)
CodecFactory	IOP::CodecFactory	See *CORBA, Part 2*: ORB Interoperability Architecture clause.
PICurrent	PortableInterceptors::Current	Portable Interceptor Current Interface on page 387.
ComponentHomeFinder	Components::HomeFinder	Components specification (formal/02-06-65).
PSS	CosPersistentState::ConnectorRegistry	*Persistent State* specification (formal/02-09-06).

To allow an application to determine which objects have references available via the initial references mechanism, the **list_initial_services** operation (also a call on the ORB) is provided. It returns an **ObjectIdList,** which is a sequence of **ObjectIds**. **ObjectIds** are typed as strings. Each object, which may need to be made available at initialization time, is allocated a string value to represent it.

In addition to defining the id, the type of object being returned must be defined; that is, "**InterfaceRepository**" returns an object of type **Repository**, or **ComponentIR::Repository**, which is derived from **Repository**, depending on whether the ORB supports components or not, and "**NameService**" returns a **CosNaming::NamingContext** object.

The application is responsible for narrowing the object reference returned from **resolve_initial_references** to the type that was requested in the ObjectId. For example, for **InterfaceRepository** the object returned would be narrowed to **Repository** type or **ComponentIR::Repository** type, depending on whether the ORB supports components.

Specifications for Object Services (see individual service specifications) state whether it is expected that a service's initial reference be made available via the **resolve_initial_references** operation or not; that is, whether the service is necessary or desirable for bootstrap purposes.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

8.5.3 Configuring Initial Service References

8.5.3.1 ORB-specific Configuration

It is required that an ORB can be administratively configured to return an arbitrary object reference from **CORBA::ORB::resolve_initial_references** for non-locality-constrained objects.

In addition to this required implementation-specific configuration, two **CORBA::ORB_init** arguments are provided to override the ORB initial reference configuration.

8.5.3.2 ORBInitRef

The ORB initial reference argument, **-ORBInitRef**, allows specification of an arbitrary object reference for an initial service. The format is:

-ORBInitRef <ObjectID>=<ObjectURL>

Examples of use are:

-ORBInitRef NameService=IOR:00230021AB...

-ORBInitRef NotificationService=corbaloc::555objs.com/NotificationService

-ORBInitRef TradingService=corbaname::555objs.com#Dev/Trader

<ObjectID> represents the well-known **ObjectID** for a service defined in the CORBA specification, such as **NameService**. This mechanism allows an ORB to be configured with new initial service Object IDs that were not defined when the ORB was installed.

<ObjectURL> can be any of the URL schemes supported by **CORBA::ORB::string_to_object** (Part 2 - Clause 7, ORB Interoperability Architecture - 7.6.1, Object URLs), with the exception of the corbaloc URL scheme with the rir protocol (i.e., corbaloc:rir...). If a URL is syntactically malformed or can be determined to be invalid in an implementation defined manner, **ORB_init** raises a **BAD_PARAM** exception.

8.5.3.3 ORBDefaultInitRef

The ORB default initial reference argument, **-ORBDefaultInitRef**, assists in resolution of initial references not explicitly specified with **-ORBInitRef**. **-ORBDefaultInitRef** requires a URL that, after appending a slash '/' character and a stringified object key, forms a new URL to identify an initial object reference. For example:

-ORBDefaultInitRef corbaloc::555objs.com

A call to **resolve_initial_references (see the "NotificationService"**) with this argument results in a new URL:

corbaloc::555objs.com/NotificationService

That URL is passed to **CORBA::ORB::string_to_object** to obtain the initial reference for the service.

Another example is:

**-ORBDefaultInitRef **
corbaname::555ResolveRefs.com,:555Backup.com#Prod/Local

After calling **resolve_initial_references("NameService")**, one of the **corbaname** URLs

-`,,`-`-`,,`,,`,`,,`---
© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

corbaname::555ResolveRefs.com#Prod/Local/NameService

or

corbaname::555Backup411.com#Prod/Local/NameService

is used to obtain an object reference from **string_to_object**. (In this example, **Prod/Local/NameService** represents a stringified **CosNaming::Name**).

See *CORBA, Part 2* - ORB Interoperability Architecture for details of the **corbaloc** and **corbaname** URL schemes. The **-ORBDefaultInitRef** argument naturally extends to URL schemes that may be defined in the future, provided the final part of the URL is an object key.

8.5.3.4 Configuration Effect on resolve_initial_references

Default Resolution Order

The default order for processing a call to **CORBA::ORB::resolve_initial_references** for a given **<ObjectID>** is:

1. Resolve with **register_initial_reference** entry if possible.
2. Resolve with **-ORBInitRef** for this **<ObjectID>** if possible
3. Resolve with pre-configured ORB settings if possible.
4. Resolve with an **-ORBDefaultInitRef** entry if possible.

ORB Configured Resolution Order

There are cases where the default resolution order may not be appropriate for all services and use of **-ORBDefaultInitRef** may have unintended resolution side effects). For example, an ORB may use a proprietary service, such as **ImplementationRepository**, for internal purposes and may want to prevent a client from unknowingly diverting the ORB's reference to an implementation repository from another vendor. To prevent this, an ORB is allowed to ignore the **-ORBDefaultInitRef** argument for any or all **<ObjectID>**s for those services that are not OMG-specified services with a well-known service name as accepted by **resolve_initial_references**. An ORB can only ignore the **-ORBDefaultInitRef** argument but must always honor the **-ORBInitRef** argument.

8.5.3.5 Configuration Effect on list_initial_services

The **<ObjectID>**s of all **-ORBInitRef argument**s to **ORB_init** appear in the list of tokens returned by **list_initial_services** as well as all ORB-configured **<ObjectID>**s. Any other tokens that may appear are implementation-dependent.

The list of **<ObjectID>**s returned by **list_initial_services** can be a subset of the **<ObjectID>**s recognized as valid by **resolve_initial_references**.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.6 Context Object

8.6.1 Introduction

A context object contains a list of properties, each consisting of a name and a string value associated with that name. By convention, context properties represent information about the client, environment, or circumstances of a request that are passed as a single parameter representing that collection of information.

Context properties represent a portion of a client's or application's environment that is meant to be propagated to (and made available to) a server's environment (for example, a window identifier, or user preference information). Once an operation has been invoked in the server, the operation implementation may query its context object for these properties.

An operation definition may contain a context clause that specifies the context properties that may be of interest to a particular operation. These context properties (if present for the actual call) are propagated to the server. A client-side ORB may choose to pass more properties than are specified by an operation's context clause. An example of an operation with a context clause is

```
interface Example {
    void op() context("USER", "X*");
};
```

This context clause specifies that the "USER" property is to be made available to the server, as well as all properties with names beginning with "X." Note that there is no obligation on the client to actually pass values for these properties at run time; if the client omits one or more properties, the call proceeds normally and the operation implementation simply will not be able to retrieve the corresponding property values.

Property names are non-empty strings that cannot contain the character '*' - there are no other syntactic restrictions on property names. Property names that differ only in case are distinct names, so the following is a legal context clause that transmits two distinct properties:

```
interface Example2 {
    void op() context("FOO", "foo");
};
```

Context property values are strings. An empty string is a legal property value.

Property values are modified and accessed via the **Context** interface. A **Context** object represents a collection of property values. **Context** objects may be connected into hierarchies; properties defined in child **Context** objects lower in the hierarchy override properties in parent **Context** objects higher in the hierarchy.

8.6.2 Context Object Operations

Properties are represented as named value lists.

```
module CORBA {
    interface Context {                 // PIDL
        void set_one_value(
            in Identifier    prop_name,    // property name to set
            in string        value         // property value to set
        );
        void set_values(
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            in NVList       values          // property values to set
        );
         void get_values(
            in Identifier   start_scope,    // search scope
            in Flags        op_flags,       // operation flags
            in Identifie    prop_name,      // name of property(s) to retrieve
            out NVList      values          // requested property(s)
        );
        void delete_values(
            in Identifie    prop_name       // name of property(s) to delete
        );
         void create_child(
            in Identifier   ctx_name,       // name of context object
            out Context     child_ctx       // newly created context object
        );
        void delete(
             in Flags       del_flags       // flags controlling deletion
        );
    };
};
```

8.6.2.1 set_one_value

```
void set_one_value(
    in Identifier   prop_name,    // property name to set
    in string       value         // property value to set
);
```

This operation sets a single context object property. If **prop_name** is the empty string or contains the character '*,' the operation raises BAD_PARAM with minor code 35.

8.6.2.2 set_values

```
void set_values(
    in NVLis        values        // property values to set
);
```

This operation sets one or more property values in its context object. If a property name appears more than once in the **NVList**, the value with higher index (later in the list) overwrites the value with lower index.

The **flags** field of each passed **NVList** element must be zero. A non-zero flag in any of the **NVList** elements raises **INV_FLAGS**.

The property name of each **NVList** element must be a non-empty string not containing the character '*'. Otherwise the operation raises BAD_PARAM with minor code 35.

The value of each property of the passed **NVList** must be a (possibly empty) unbounded string. Property values other than unbounded strings raise BAD_TYPECODE with minor code 3.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.6.2.3 get_values

```
void get_values(
    in Identifie    start_scope,    // search scope
    in Flags        op_flags,       // operation flags
    in Identifier   prop_name,      // name of property(s) to retrieve
    out NVList      values          // requested property(s)
);
```

This operation returns an **NVList** with those properties that match the **prop_name** parameter. Legal values for **prop_name** are:

- A non-empty string that does not contain the character '*.'

  In this case, the **values** parameter returns the property with the name specified by **prop_name**.

- A string beginning with one or more characters other than '*,' followed by a single '*' at the end, such as "XYZ*."

  In this case, the **values** parameter contains the properties that have names beginning with "XYZ" (such as "XYZABC" or "XYZ").

If **prop_name** is the empty string, the string "*," contains more than one '*' character, or contains a '*' anywhere but at the end of the string, the operation raises **BAD_PARAM** with minor code 36.

The **start_scope** parameter controls the **context** object level at which to initiate the search for the specified properties as follows:

- The **start_scope** parameter specifies the name of the **context** object in which the search for properties is to start.
- If the context object on which **get_values** is invoked has a name equal to **start_scope**, that context object becomes the starting **context** object for the search.
- If **start_scope** is "" the context object on which **get_values** is invoked becomes the starting **context** object for the search.
  - If the **context** object on which **get_values** is invoked does not have a name equal to **start_scope** (and **start_scope** is not ""), the parent context object is retrieved and its name compared to **start_scope**; this process repeats until either a starting **context** object whose name equals **start_scope** is found, or the search terminates because it runs out of parent objects.

    The name of the root **context** object created by **get_default_context** is "RootContext."

    If no starting **context** object can be found, the operation raises **BAD_CONTEXT** with minor code 1.
  - Once a starting **context** object is found, **get_values** searches for properties in the matching **context** object.
  - If op_flags is **CORBA::CTX_RESTRICT_SCOPE**, **get_values** searches only the starting **context** object for properties that match **prop_name**. (The value of **CTX_RESTRICT_SCOPE** is 15.)
- If **op_flags** is zero, **get_values** searches the starting **context** and its parent **contexts** for properties that match **prop_name**. The property values that are returned are taken from the first **context** object in which they are found, so properties in child contexts override the values of properties in parent contexts.

In either case, if no property matches **prop_name**, the operation raises **BAD_CONTEXT** with minor code 2.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.6.2.4 delete_values

```
void delete_values(
    in Identifie        prop_name     // name of property(s) to delete
);
```

This operation deletes the properties that match **prop_name**. **prop_name** may have a trailing '*' character, in which case all properties whose name matches the specified prefix are deleted.

If **prop_name** is the empty string, the string "*", contains more than one '*' character, or contains a '*' anywhere but at the end of the string, the operation raises **BAD_PARAM** with minor code 36. The operation only affects the context object on which it is invoked (that is, parent contexts are never affected by **delete_values**).

If no property name matches **prop_name**, the operation raises **BAD_CONTEXT** with minor code 2.

8.6.2.5 create_child

```
void create_child(
    in Identifier       ctx_name,     // name of context object
   out Context          child_ctx     // newly created context object
);
```

This operation creates an empty child context object. The child context has the name **ctx_name**. **ctx_name** may not be the empty string or "RootContext;" otherwise, the operation raises **BAD_PARAM** with minor code 37. Calling **create_child** more than once with the same name on the same parent context is legal and results in the creation of a new, empty child context for each call.

8.6.2.6 delete

```
void delete(
    in Flags        del_flags       // flags controlling deletion
);
```

This operation deletes the context object on which it is invoked:

- If **del_flags** is zero, the context object is deleted only if it has no child contexts; otherwise, if **del_flags** is zero and the context object has child contexts, the operation raises **BAD_PARAM** with minor code 38.
- If **del_flags** is **CORBA::CTX_DELETE_DESCENDANTS**, the context object on which delete is invoked is destroyed, together with (recursively) its child contexts. The value of **CTX_DELETE_DESCENDANTS** is 1.

If **del_flags** has a value other than zero or **CTX_DELETE_DESCENDANTS**, the operation raises **INV_FLAGS**.

8.7 Current Object

ORB and CORBA services may wish to provide access to information (context) associated with the thread of execution in which they are running. This information is accessed in a structured manner using interfaces derived from the **Current** interface defined in the **CORBA** module.

Each ORB or CORBA service that needs its own context derives an interface from the **CORBA** module's **Current**. Users of the service can obtain an instance of the appropriate **Current** interface by invoking **ORB::resolve_initial_references**. For example the Security service obtains the **Current** relevant to it by invoking.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

ORB::resolve_initial_references(“SecurityCurrent”)

A CORBA service does not have to use this method of keeping context but may choose to do so.

```
module CORBA {
    // interface for the Current object
     local interface Current {
    };
};
```

Operations on interfaces derived from **Current** access state associated with the thread in which they are invoked, not state associated with the thread from which the **Current** was obtained. This prevents one thread from manipulating another thread’s state, and avoids the need to obtain and narrow a new **Current** in each method’s thread context.

Current objects must not be exported to other processes, or externalized with **ORB::object_to_string**. If any attempt is made to do so, the offending operation will raise a MARSHAL system exception. **Current**s are per-process singleton objects, so no destroy operation is needed.

8.8 Policy Object

8.8.1 Definition of Policy Object

An ORB or CORBA service may choose to allow access to certain choices that affect its operation. This information is accessed in a structured manner using interfaces derived from the **Policy** interface defined in the **CORBA** module. A CORBA service does not have to use this method of accessing operating options, but may choose to do so. The *Security Service* in particular uses this technique for associating *Security Policy* with objects in the system.

```
module CORBA {
    typedef unsigned long PolicyType;

    // Basic IDL definition
    interface Policy {
        readonly attribute PolicyType policy_type;
        Policy copy();
        void destroy();
    };

    typedef sequence <Policy> PolicyList;
    typedef sequence <PolicyType> PolicyTypeSeq;
};
```

PolicyType defines the type of **Policy** object. In general the constant values that are allocated are defined in conjunction with the definition of the corresponding **Policy** object. The values of **PolicyTypes** for policies that are standardized by OMG are allocated by OMG. Additionally, vendors may reserve blocks of 4096 PolicyType values identified by a 20 bit *Vendor PolicyType Valueset ID* (**VPVID**) for their own use.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

PolicyType which is an unsigned long consists of the 20-bit **VPVID** in the high order 20 bits, and the vendor assigned policy value in the low order 12 bits. The **VPVID**s *0 through \xf* are reserved for OMG. All values for the standard **PolicyTypes** are allocated within this range by OMG. Additionally, the **VPVID**s *\xfffff* is reserved for experimental use and **OMGVMCID** (8.12.3, Standard System Exception Definitions) is reserved for OMG use. These will not be allocated to anybody. Vendors can request allocation of **VPVID** by sending mail to tag-*request@omg.org*.

When a **VMCID** (Exceptions on page 148) is allocated to a vendor automatically the same value of **VPVID** is reserved for the vendor and vice versa. So once a vendor gets either a **VMCID** or a **VPVID** registered they can use that value for both their minor codes and their policy types.

8.8.1.1 Copy

Policy copy();

Return Value

This operation copies the policy object. The copy does not retain any relationships that the policy had with any domain, or object.

8.8.1.2 Destroy

void destroy();

This operation destroys the policy object. It is the responsibility of the policy object to determine whether it can be destroyed.

Exception(s)

- CORBA::NO_PERMISSION
  Raised when the policy object determines that it cannot be destroyed.

8.8.1.3 Policy_type

readonly attribute policy_type

Return Value

This readonly attribute returns the constant value of type **PolicyType** that corresponds to the type of the **Policy** object.

8.8.2 Creation of Policy Objects

A generic ORB operation for creating new instances of Policy objects is provided as described in this sub clause.

```
module CORBA {

    typedef short PolicyErrorCode;
    const PolicyErrorCode BAD_POLICY = 0;
    const PolicyErrorCode UNSUPPORTED_POLICY = 1;
    const PolicyErrorCode BAD_POLICY_TYPE = 2;
    const PolicyErrorCode BAD_POLICY_VALUE = 3;
    const PolicyErrorCode UNSUPPORTED_POLICY_VALUE = 4;
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    exception PolicyError {PolicyErrorCode reason;};

    interface ORB {

        .....

        Policy create_policy(
            in PolicyType type,
            in any val
        ) raises(PolicyError);
    };
};
```

8.8.2.1 PolicyErrorCode

A request to create a **Policy** may be invalid for the following reasons:

- BAD_POLICY - the requested **Policy** is not understood by the ORB.
- UNSUPPORTED_POLICY - the requested **Policy** is understood to be valid by the ORB, but is not currently supported.
- BAD_POLICY_TYPE - The type of the value requested for the **Policy** is not valid for that **PolicyType**.
- BAD_POLICY_VALUE - The value requested for the **Policy** is of a valid type but is not within the valid range for that type.
- UNSUPPORTED_POLICY_VALUE - The value requested for the **Policy** is of a valid type and within the valid range for that type, but this valid value is not currently supported.

8.8.2.2 PolicyError

```
    exception PolicyError {PolicyErrorCode reason;};
```

PolicyError exception is raised to indicate problems with parameter values passed to the **ORB::create_policy** operation. Possible reasons are described above.

8.8.2.3 Create_policy

The ORB operation **create_policy** can be invoked to create new instances of policy objects of a specific type with specified initial state. If **create_policy** fails to instantiate a new **Policy** object due to its inability to interpret the requested type and content of the policy, it raises the PolicyError exception with the appropriate reason as described in PolicyErrorCode on page 128.

```
    Policy create_policy(
        in PolicyType type,
        in any val
    ) raises(PolicyError);
```

Parameters

- **type**
  The **PolicyType** of the policy object to be created.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- **val**
  The value that will be used to set the initial state of the **Policy** object that is created.

Return Value

Reference to a newly created **Policy** object of type specified by the **type** parameter and initialized to a state specified by the **val** parameter.

Exception

- PolicyError
  Raised when the requested policy is not supported or a requested initial state for the policy is not support.

When new policy types are added to CORBA or CORBA Services specification, it is expected that the IDL type and the valid values that can be passed to **create_policy** also be specified.

8.8.3 Usages of Policy Objects

Policy Objects are used in general to encapsulate information about a specific policy, with an interface derived from the policy interface. The type of the Policy object determines how the policy information contained within it is used. Usually a Policy object is associated with another object to associate the contained policy with that object.

Objects with which policy objects are typically associated are Domain Managers, POA, the execution environment, both the process/capsule/ORB instance and thread of execution (Current object) and object references. Only certain types of policy object can be meaningfully associated with each of these types of objects.

These relationships are documented in sub clauses that pertain to these individual objects and their usages in various core facilities and object services. The use of Policy Objects with the POA are discussed in the *Portable Object Adapter* clause. The use of Policy objects in the context of the Security services, involving their association with Domain Managers as well as with the Execution Environment are discussed in the *Security Service* specification.

In the following sub clause the association of Policy objects with the Execution Environment is discussed. In Management of Policies on page 131 the use of Policy objects in association with Domain Managers is discussed.

8.8.4 Policy Associated with the Execution Environment

Certain policies that pertain to services like security (e.g., QOP, Mechanism, invocation credentials, etc.) are associated by default with the process/capsule(RM-ODP)/ORB instance (hereinafter referred to as “capsule”) when the application is instantiated together with the capsule. By default these policies are applicable whenever an invocation of an operation is attempted by any code executing in the said capsule. The Security service provides operations for modulating these policies on a per-execution thread basis using operations in the **Current** interface. Certain of these policies (e.g., invocation credentials, qop, mechanism, etc.) which pertain to the invocation of an operation through a specific object reference can be further modulated at the client end, using the **set_policy_overrides** operation of the **Object** reference. For a description of this operation see Overriding Associated Policies on an Object Reference on page 111. It associates a specified set of policies with a newly created object reference that it returns.

The association of these overridden policies with the object reference is a purely local phenomenon. These associations are never passed on in any IOR or any other marshaled form of the object reference. the associations last until the object reference in the capsule is destroyed or the capsule in which it exists is destroyed.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The policies thus overridden in this new object reference and all subsequent duplicates of this new object reference apply to all invocations that are done through these object references. The overridden policies apply even when the default policy associated with **Current** is changed. It is always possible that the effective policy on an object reference at any given time will fail to be successfully applied, in which case the invocation attempt using that object reference will fail and return a CORBA::NO_PERMISSION exception. Only certain policies that pertain to the invocation of an operation at the client end can be overridden using this operation. These are listed in the Security specification. Attempts to override any other policy will result in the raising of the CORBA::NO_PERMISSION exception.

In general the policy of a specific type that will be used in an invocation through a specific object reference using a specific thread of execution is determined first by determining if that policy type has been overridden in that object reference. if so then the overridden policy is used. if not then if the policy has been set in the thread of execution then that policy is used. If not, then the policy associated with the capsule is used. For policies that matter, the ORB ensures that there is a default policy object of each type that matters associated with each capsule (ORB instance). Hence, in a correctly implemented ORB there is no case when a required type policy is not available to use with an operation invocation.

8.8.5 Specification of New Policy Objects

When new **PolicyType**s are added to CORBA specifications, the following details must be defined. It must be clearly stated which particular uses of a new policy are legal and which are not:

- Specify the assigned **CORBA::PolicyType** and the policy's interface definition.
- If the **Policy** can be created through **CORBA::ORB::create_policy**, specify the allowable values for the any argument 'val' and how they correspond to the initial state/behavior of that **Policy** (such as initial values of attributes). For example, if a Policy has multiple attributes and operations, it is most likely that **create_policy** will receive some complex data for the implementation to initialize the state of the specific policy:

```
//IDL
struct MyPolicyRange {
    long low;
    long high;
};

const CORBA::PolicyType MY_POLICY_TYPE = 666;
interface MyPolicy : Policy {
    readonly attribute long low;
    readonly attribute long high;
};
```

  If this sample **MyPolicy** can be constructed via create_policy, the specification of **MyPolicy** will have a statement such as: "When instances of **MyPolicy** are created, a value of type **MyPolicyRange** is passed to **CORBA::ORB::create_policy** and the resulting MyPolicy's attribute 'low' has the same value as the **MyPolicyRange** member 'low' and attribute 'high' has the same value as the **MyPolicyRange** member 'high.'

- If the **Policy** can be passed as an argument to **POA::create_POA**, specify the effects of the new policy on that **POA**. Specifically define incompatibilities (or inter-dependencies) with other **POA** policies, effects on the behavior of invocations on objects activated with the **POA**, and whether or not presence of the POA policy implies some **IOR** profile/component contents for object references created with that **POA**. If the **POA** policy implies some addition/ modification to the object reference, it is marked as "client-exposed" and the exact details are specified including which profiles are affected and how the effects are represented.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- If the component that is used to carry this information can be set within a client to tune the client's behavior, specify the policy's effects on the client specifically with respect to (a) establishment of connections and reconnections for an object reference; (b) effects on marshaling of requests; (c) effects on insertion of service contexts into requests; (d) effects upon receipt of service contexts in replies. In addition, incompatibilities (or inter-dependencies) with other client-side policies are stated. For policies that cause service contexts to be added to requests, the exact details of this addition are given.
- If the **Policy** can be used with **POA** creation to tune **IOR** contents and can also be specified (overridden) in the client, specify how to reconcile the policy's presence from both the client and server. It is strongly recommended to avoid this case! As an exercise in completeness, most **POA** policies can probably be extended to have some meaning in the client and vice versa, but this does not help make usable systems, it just makes them more complicated without adding really useful features. There are very few cases where a policy is really appropriate to specify in both places, and for these policies the interaction between the two must be described.
- Pure client-side policies are assumed to be immutable. This allows efficient processing by the runtime that can avoid re-evaluating the policy upon every invocation and instead can perform updates only when new overrides are set (or policies change due to rebind). If the newly specified policy is mutable, it must be clearly stated what happens if non-readonly attributes are set or operations are invoked that have side-effects.
- For certain policy types, override operations may be disallowed. If this is the case, the policy specification must clearly state what happens if such overrides are attempted.

8.8.6 Standard Policies

NOTE: See Annex A for a list of the standard policy types that are defined by various parts of CORBA and CORBAservices in this version of CORBA.

8.9 Management of Policies

8.9.1 Client Side Policy Management

Client-side Policy management is performed through operations accessible in the following contexts:

- ORB-level Policies - A locality-constrained PolicyManager is accessible through the ORB interface. This PolicyManager has operations through which a set of Policies can be applied and the current overriding Policy settings can be obtained. Policies applied at the ORB level override any system defaults. The ORB's PolicyManager is obtained through an invocation of ORB::resolve_initial_references, specifying an identifier of "ORBPolicyManager."
- Thread-level Policies - A standard PolicyCurrent is defined with operations for the querying and applying of quality of service values specific to a thread. Policies applied at the thread level override any system defaults or values set at the ORB level. The locality-constrained PolicyCurrent is obtained through an invocation of ORB::resolve_initial_references, specifying an identifier of "PolicyCurrent." When accessed from a newly spawned thread, the PolicyCurrent initially has no overridden policies. The PolicyCurrent also has no overridden values when a POA with ThreadPolicy of ORB_CONTROL_MODEL dispatches an invocation to a servant. Each time an invocation is dispatched through a SINGLE_THREAD_MODEL POA, the thread-level overrides are reset to have no overridden values.
- Object-level Policies - Operations are defined on the base Object interface through which a set of Policies can be applied. Policies applied at the Object level override any system defaults or values set at the ORB or Thread levels. In addition, accessors are defined for querying the current *overriding* Policies set at the Object level, and for obtaining the

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

current *effective client-side* Policy of a given PolicyType. The e*ffective client-side* Policy is the value of a PolicyType that would be in effect if a request were made. This is determined by checking for overrides at the Object level, then at the Thread level, and finally at the ORB level. If no overriding policies are set at any level, the system-dependent default value is returned. Portable applications are expected to override the ORB-level policies since default values are not specified in most cases.

8.9.2 Server Side Policy Management

Server-side Policy management is handled by associating Policy objects with a POA. Since all policy objects are derived from interface Policy, those that are applicable to server-side behavior can be passed as arguments to POA::create_POA. Any such Policies that affect the behavior of requests (and therefore must be accessible to the ORB at the client side) are exported within the Object references that the POA creates. It is clearly noted in a POA Policy definition when that Policy is of interest to the Client. For those policies that can be exported within an Object reference, the absence of a value for that policy type implies that the target supports any legal value of that PolicyType.

Most Policies are appropriate only for management at either the Server or Client, but not both. For those Policies that can be established at the time of Object reference creation (through POA Policies) and overridden by the client (through overrides set at the ORB, thread, or Object reference scopes), reconciliation is done on a per-Policy basis. Such Policies are clearly noted in their definitions and describe the mechanism of reconciliation between the Policies that are set by the POA and overridden in the client. Furthermore, obtaining the effective Policy of some **PolicyType**s requires evaluating the effective Policy of other types of Policies. Such hierarchical Policy definitions are also noted clearly when used.

At the Thread and ORB scopes, the common operations for querying the current set of policies and for overriding these settings are encapsulated in the PolicyManager interface.

8.9.3 Policy Management Interfaces

```
module CORBA {

    local interface PolicyManager {

        PolicyList get_policy_overrides(in PolicyTypeSeq ts);

        void set_policy_overrides(
                in PolicyList           policies,
                in SetOverrideType      set_add
        ) raises (InvalidPolicies);
    };

    local interface PolicyCurrent : PolicyManager, Current {
    };
};
```

8.9.3.1 interface PolicyManager

The PolicyManager operations are used for setting and accessing Policy overrides at a particular scope. For example, an instance of the PolicyCurrent is used for specifying Policy overrides that apply to invocations from that thread (unless they are overridden at the Object scope as described in Client Side Policy Management on page 131).

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

get_policy_overrides

PolicyList get_policy_overrides(in PolicyTypeSeq ts);

Parameter

ts
A sequence of overridden policy types identifying the policies that are to be retrieved.

Return Value

Reference to a newly created **Policy** object of type specified by the **type** parameter and initialized to a state specified by the **val** parameter.

- policy list
  The list of overridden policies of the types specified by ts.

Exception
None

Returns a PolicyList containing the overridden Polices for the requested PolicyTypes. If the specified sequence is empty, all Policy overrides at this scope will be returned. If none of the requested **PolicyType**s are overridden at the target PolicyManager, an empty sequence is returned. This accessor returns only those Policy overrides that have been set at the specific scope corresponding to the target PolicyManager (no evaluation is done with respect to overrides at other scopes).

8.9.3.2 set_policy_overrides

```
void set_policy_overrides(
        in PolicyList       policies,
        in SetOverrideType  set_add
) raises (InvalidPolicies);
```

Parameters

- **policies**
  A sequence of **Policy** objects that are to be associated with the **PolicyManager** object. If the sequence contains two or more **Policy** objects with the same **PolicyType** value, the operation raises the standard system exception BAD_PARAM with standard minor code 30.
- **set_add**
  Whether the association is in addition to (**ADD_OVERRIDE**) or as a replacement of (**SET_OVERRIDE**) any existing overrides already associated with the **PolicyManager** object. If the value of this parameter is **SET_OVERRIDE**, the supplied **policies** completely replace all existing overrides associated with the **PolicyManager** object. If the value of this parameter is **ADD_OVERRIDE**, the supplied **policies** are added to the existing overrides associated with the **PolicyManager** object, except that if a supplied **Policy** object has the same **PolicyType** value as an existing override, the supplied **Policy** object replaces the existing override.

Return Value
None

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Exception

- InvalidPolicies
  A list of indices identifying the position in the input policies list that are occupied by invalid policies.

Modifies the current set of overrides with the requested list of **Policy** overrides. The first parameter policies is a sequence of references to **Policy** objects. The second parameter **set_add** of type **SetOverrideType** indicates whether these policies should be added onto any other overrides that already exist (**ADD_OVERRIDE**) in the **PolicyManager**, or they should be added to a clean **PolicyManager** free of any other overrides (**SET_OVERRIDE**). Invoking **set_policy_overrides** with an empty sequence of policies and a mode of **SET_OVERRIDE** removes all overrides from a **PolicyManager**. Only certain policies that pertain to the invocation of an operation at the client end can be overridden using this operation. Attempts to override any other policy will result in the raising of the CORBA::NO_PERMISSION exception. If the request would put the set of overriding policies for the target **PolicyManager** in an inconsistent state, no policies are changed or added, and the exception.

8.9.3.3 interface PolicyCurrent

This specific **PolicyManager** provides access to policies overridden at the Thread scope. A reference to a thread's **PolicyCurrent** is obtained through an invocation of CORBA::ORB::resolve_initial_references.

8.10 Management of Policy Domains

8.10.1 Basic Concepts

This sub clause describes how policies, such as security policies, are associated with objects that are managed by an ORB. The interfaces and operations that facilitate this aspect of management is described in this sub clause together with the sub clause describing **Policy** objects.

8.10.1.1 Policy Domain

A policy domain is a set of objects to which the policies associated with that domain apply. These objects are the domain members. The policies represent the rules and criteria that constrain activities of the objects that belong to the domain. On object reference creation, the ORB implicitly associates the object reference with one or more policy domains. Policy domains provide leverage for dealing with the problem of scale in policy management by allowing application of policy at a domain granularity rather than at an individual object instance granularity.

8.10.1.2 Policy Domain Manager

A policy domain includes a unique object, one per policy domain, called the domain manager, which has associated with it the policy objects for that domain. The domain manager also records the membership of the domain and provides the means to add and remove members. The domain manager is itself a member of a domain, possibly the domain it manages.

8.10.1.3 Policy Objects

A policy object encapsulates a policy of a specific type. The policy encapsulated in a policy object is associated with the domain by associating the policy object with the domain manager of the policy domain.

There may be several policies associated with a domain, with a policy object for each. There is at most one policy of each type associated with a policy domain. The policy objects are thus shared between objects in the domain, rather than being associated with individual objects. Consequently, if an object needs to have an individual policy, then it must be a singleton member of a domain.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.10.1.4 Object Membership of Policy Domains

Since the only way to access objects is through object references, associating object references with policy domains, implicitly associates the domain policies with the object associated with the object reference. Care should be taken by the application that is creating object references using **POA** operations to ensure that object references to the same object are not created by the server of that object with different domain associations. Henceforth whenever the concept of "object membership" is used, it actually means the membership of an object reference to the object in question.

An object can simultaneously be a member of more than one policy domain. In that case the object is governed by all policies of its enclosing domains. The reference model allows an object to be a member of multiple domains, which may overlap for the same type of policy (for example, be subject to overlapping access policies). This would require conflicts among policies defined by the multiple overlapping domains to be resolved. The specification does not include explicit support for such overlapping domains and, therefore, the use of policy composition rules required to resolve conflicts at policy enforcement time.

Policy domain managers and policy objects have two types of interfaces:

- The operational interfaces used when enforcing the policies. These are the interfaces used by the ORB during an object invocation. Some policy objects may also be used by applications, which enforce their own policies.

  The caller asks for the policy of a particular type (e.g., the delegation policy), and then uses the policy object returned to enforce the policy. The caller finding a policy and then enforcing it does not see the domain manager objects and the domain structure.

- The administrative interfaces used to set policies (e.g., specifying which events to audit or who can access objects of a specified type in this domain). The administrator sees and navigates the domain structure, so he is aware of the scope of what he is administering.

NOTE: This specification does not include any explicit interfaces for managing the policy domains themselves: creating and deleting them; moving objects between them; changing the domain structure and adding, changing, and removing policies applied to the domains.

8.10.1.5 Domains Association at Object Reference Creation

When a new object reference is created, the ORB implicitly associates the object reference (and hence the object that it is associated with) with the following elements forming its environment:

- One or more *Policy Domains*, defining all the policies to which the object associated with the object reference is subject.

- The *Technology Domains*, characterizing the particular variants of mechanisms (including security) available in the ORB.

The ORB will establish these associations when one of the object reference creation operations of the POA is called. Some or all of these associations may subsequently be explicitly referenced and modified by administrative or application activity, which might be specifically security-related but could also occur as a side-effect of some other activity, such as moving an object to another host machine.

In some cases, when a new object reference is created, it needs to be associated with a new domain. Within a given domain a construction policy can be associated with a specific object type thus causing a new domain; that is, a domain manager object to be created whenever an object reference of that type is created and the newly created object reference associated with the new domain manager. This construction policy is enforced at the same time as the domain membership; that is, by the POA when it creates an object reference.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.10.1.6 Implementor's View of Object Creation

For policy domains, the construction policy of the application or factory creating the object proceeds as follows. The application (which may be a generic factory) calls one of the object reference creation operations of the POA to create the new object reference. The ORB obtains the construction policy associated with the creating object, or the default domain absent a creating object.

By default, the new object reference that is created is made a member of the domain to which the parent belongs. Non-object applications on the client side are associated with a default, per-ORB instance policy domain by the ORB.

Each domain manager has a construction policy associated with it, which controls whether, in addition to creating the specified new object reference, a new domain manager is created with it. This object provides a single operation **make_domain_manager** which can be invoked with the **constr_policy** parameter set to **TRUE** to indicate to the ORB that new object references of the specified type are to be associated their own separate domains. Once such a construction policy is set, it can be reversed by invoking **make_domain_manager** again with the **constr_policy** parameter set to **FALSE**.

When creating an object reference of the type specified in the **make_domain_manager** call with **constr_policy** set to **TRUE**, the ORB must also create a new domain for the newly created object reference. If a new domain is needed, the ORB creates both the requested object reference and a domain manager object. A reference to this domain manager can be found by calling **get_domain_managers** on the newly created object reference.

While the management interface to the construction policy object is standardized, the interface from the ORB to the policy object is assumed to be a private one, which may be optimized for different implementations.

If a new domain is created, the policies initially applicable to it are the policies of the enclosing domain. The ORB will always arrange to provide a default enclosing domain with default ORB policies associated with it, in those cases where there would be no such domain as in the case of a non-object client invoking object creation operations.

The calling application, or an administrative application later, can change the domains to which this object belongs, using the domain management interfaces, which will be defined in the future.

Since the ORB has control only over domain associations with object references, it is the responsibility of the creator of new object to ensure that the object references that are created to the new object are associated meaningfully with domains.

8.10.2 Domain Management Operations

This sub clause defines the interfaces and operations needed to find domain managers and find the policies associated with these. However, it does not include operations to manage domain membership, structure of domains, or to manage which policies are associated with domains.

This sub clause also includes the interface to the construction policy object, as that is relevant to domains. The basic definitions of the interfaces and operations related to these are part of the **CORBA** module, since other definitions in the **CORBA** module depend on these.

```
module CORBA {
    interface DomainManager {
        Policy get_domain_policy (
            in PolicyType policy_type
        );
    };
```


© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    const PolicyType SecConstruction = 11;

    interface ConstructionPolicy: Policy{
        void make_domain_manager(
            in CORBA::InterfaceDef object_type,
            in boolean constr_policy
        );
    };

    typedef sequence <DomainManager> DomainManagersList;
};
```

8.10.2.1 Domain Manager

The domain manager provides mechanisms for:

- Establishing and navigating relationships to superior and subordinate domains.
- Creating and accessing policies.

There should be no unnecessary constraints on the ordering of these activities, for example, it must be possible to add new policies to a domain with a pre-existing membership. In this case, some means of determining the members that do not conform to a policy that may be imposed is required. It should be noted that interfaces for adding new policies to domains or for changing domain memberships have not currently been standardized.

All domain managers provide the **get_domain_policy** operation. By virtue of being an object, the Domain Managers also have the **get_policy** and **get_domain_managers** operations, which is available on all objects (see Getting Policy Associated with the Object on page 110 and Getting the Domain Managers Associated with the Object on page 112).

CORBA::DomainManager::get_domain_policy

This returns the policy of the specified type for objects in this domain.

```
Policy get_domain_policy (
    in PolicyType policy_type
);
```

Parameters

- **policy_type**
  The type of policy for objects in the domain which the application wants to administer. For security, the possible policy types are described in the *Security Service* specification, Security Policies Introduction sub clause.

Return Value

A reference to the policy object for the specified type of policy in this domain.

Exception

- CORBA::INV_POLICY
  Raised when the value of policy type is not valid either because the specified type is not supported by this ORB or because a policy object of that type is not associated with this Object.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.10.2.2 Construction Policy

The construction policy object allows callers to specify that when instances of a particular object reference are created, they should be automatically assigned membership in a newly created domain at creation time.

CORBA::ConstructionPolicy::make_domain_manager

This operation enables the invoker to set the construction policy that is to be in effect in the domain with which this **ConstructionPolicy** object is associated. Construction Policy can either be set so that when an object reference of the type specified by the input parameter is created, a new domain manager will be created and the newly created object reference will respond to **get_domain_managers** by returning a reference to this domain manager. Alternatively the policy can be set to associate the newly created object reference with the domain associated with the creator. This policy is implemented by the ORB during execution of any one of the object reference creation operations of the POA, and results in the construction of the application-specified object reference and a Domain Manager object if so dictated by the policy in effect at the time of the creation of the object reference.

```
void make_domain_manager (
    in InterfaceDef object_type,
    in boolean constr_policy
);
```

Parameter(s)

- **object_type**
  The type of the object references for which Domain Managers will be created. If this is nil, the policy applies to all object references in the domain.
- **constr_policy**
  If **TRUE** the construction policy is set to create a new domain manager associated with the newly created object reference of this type in this domain. If **FALSE** construction policy is set to associate the newly created object references with the domain of the creator or a default domain as described above.

8.11 TypeCodes

TypeCodes are values that represent invocation argument types and attribute types. They can be obtained from the Interface Repository or from IDL compilers.

TypeCodes have a number of uses. They are used in the dynamic invocation interface to indicate the types of the actual arguments. They are used by an Interface Repository to represent the type specifications that are part of many OMG IDL declarations. Finally, they are crucial to the semantics of the **any** type.

Abstractly, **TypeCode**s consist of a “kind” field, and a set of parameters appropriate for that kind. For example, the **TypeCode** describing OMG IDL type **long** has kind **tk_long** and no parameters. The **TypeCode** describing OMG IDL type **sequence<boolean,10>** has kind **tk_sequence** and two parameters: **10** and **boolean**.

8.11.1 The TypeCode Interface

The PIDL interface for **TypeCodes** is as follows:

```
module CORBA {
    enum TCKind {
        tk_null, tk_void,
```


© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        tk_short, tk_long, tk_ushort, tk_ulong,
        tk_float, tk_double, tk_boolean, tk_char,
        tk_octet, tk_any, tk_TypeCode, tk_Principal, tk_objref,
        tk_struct, tk_union, tk_enum, tk_string,
        tk_sequence, tk_array, tk_alias, tk_except,
        tk_longlong, tk_ulonglong, tk_longdouble,
        tk_wchar, tk_wstring, tk_fixed,
        tk_value, tk_value_box,
        tk_native,
        tk_abstract_interface,
        tk_local_interface
        tk_component, tk_home,
        tk_event
};

typedef short ValueModifier;
        const ValueModifier VM_NONE = 0;
        const ValueModifier VM_CUSTOM = 1;
        const ValueModifier VM_ABSTRACT = 2;
        const ValueModifier VM_TRUNCATABLE = 3;

interface TypeCode {
        exception       Bounds {};
        exception       BadKind {};

        // for all TypeCode kinds
        boolean equal (in TypeCode tc);

        boolean equivalent(in TypeCode tc);
        TypeCode get_compact_typecode();

        TCKind kind ();

        // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias,
        // tk_value, tk_value_box, tk_native, tk_abstract_interface
        // tk_local_interface, tk_except
        // tk_component, tk_home and tk_event
        RepositoryId id () raises (BadKind);

        // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias,
        // tk_value, tk_value_box, tk_native, tk_abstract_interface
        // tk_local_interface, tk_except
        // tk_component, tk_home and tk_event
        Identifier name () raises (BadKind);

        // for tk_struct, tk_union, tk_enum, tk_value,
        // tk_except and tk_event
        unsigned long member_count () raises (BadKind);
        Identifier member_name (in unsigned long index)
            raises(BadKind, Bounds);
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        // for tk_struct, tk_union, tk_value,
        // tk_except and tk_event
        TypeCode member_type (in unsigned long index)
            raises (BadKind, Bounds);

        // for tk_union
        any member_label (in unsigned long index)
            raises(BadKind, Bounds);
        TypeCode discriminator_type () raises (BadKind);
        long default_index () raises (BadKind);

        // for tk_string, tk_wstring, tk_sequence, and tk_array
        unsigned long length () raises (BadKind);

        // for tk_sequence, tk_array, tk_value_box and tk_alias
        TypeCode content_type () raises (BadKind);

        // for tk_fixed
        unsigned short fixed_digits() raises(BadKind);
        short fixed_scale() raises(BadKind);

        // for tk_value and tk_event
        Visibility member_visibility(in unsigned long index)
            raises(BadKind, Bounds);
        ValueModifier type_modifier() raises(BadKind);
        TypeCode concrete_base_type() raises(BadKind);
    };
};
```

With the above operations, any **TypeCode** can be decomposed into its constituent parts. The BadKind exception is raised if an operation is not appropriate for the **TypeCode** kind it invoked.

The **equal** operation can be invoked on any **TypeCode**. The **equal** operation returns **TRUE** if and only if for the target **TypeCode** and the **TypeCode** passed through the parameter **tc,** the set of legal operations is the same and invoking any operation from that set on the two **TypeCode**s return identical results.

The **equivalent** operation is used by the ORB when determining type equivalence for values stored in an IDL **any**. **TypeCodes** are considered equivalent based on the following semantics:

- If the result of the **kind** operation on either **TypeCode** is **tk_alias**, recursively replace the **TypeCode** with the result of calling **content_type**, until the kind is no longer **tk_alias**.
- If results of the **kind** operation on each typecode differ, **equivalent** returns false.
- If the **id** operation is valid for the **TypeCode kind**, **equivalent** returns **TRUE** if the results of **id** for both **TypeCodes** are non-empty strings and both strings are equal. If both ids are non-empty but are not equal, then **equivalent** returns **FALSE**. If either or both id is an empty string, or the **TypeCode kind** does not support the **id** operation, **equivalent** will perform a structural comparison of the **TypeCodes** by comparing the results of the other **TypeCode** operations in the following bullet items (ignoring aliases as described in the first bullet.). The structural comparison only calls operations that are valid for the given **TypeCode kind**. If any of these operations do not return equal results, then **equivalent** returns **FALSE**. If all comparisons are equal, **equivalent** returns true.
- The results of the **name** and **member_name** operations are ignored and not compared.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- The results of the **member_count**, **default_index**, **length**, **digits**, **scale,** and **type_modifier** operations are compared.
- The results of the **member_label** operation for each member index of a **union TypeCode** are compared for equality. Note that this means that **unions** whose members are not defined in the same order are not considered structurally equivalent.
- The results of the **discriminator_type, member_type,** and **concrete_base_type** operation and for each member index, and the result of the **content_type** operation are compared by recursively calling **equivalent**.
- The results of the **member_visibility** operation are compared for each member index.

Applications that need to distinguish between a type and different aliases of that type can supplement **equivalent** by directly invoking the **id** operation and comparing the results.

The **get_compact_typecode** operation strips out all optional **name** and **member name** fields, but it leaves all alias typecodes intact.

The **kind** operation can be invoked on any **TypeCode**. Its result determines what other operations can be invoked on the **TypeCode**.

The **id** operation can be invoked on object reference, valuetype, boxed valuetype, abstract interface, local interface, native, structure, union, enumeration, alias, exception, component, home, and event **TypeCode**s. It returns the **RepositoryId** globally identifying the type. Object reference, valuetype, boxed valuetype, native, exception, component, home, and event **TypeCode**s always have a **RepositoryId**. Structure, union, enumeration, and alias **TypeCode**s obtained from the Interface Repository or the **ORB::create_operation_list** operation also always have a **RepositoryId**. Otherwise, the **id** operation can return an empty string.

When the **id** operation is invoked on an object reference **TypeCode** that contains a base **Object**, the returned value is **IDL:omg.org/CORBA/Object:1.0**.

When it is invoked on a valuetype **TypeCode** that contains a **ValueBase**, the returned value is **IDL:omg.org/CORBA/ValueBase:1.0**.

When it is invoked on a component **TypeCode** that contains a **Components::CCMObject**, the returned value is **IDL:omg.org/Components/CCMObject:1.0**.

When it is invoked on a home **TypeCode** that contains a **Components::CCMHome,** the returned value is **IDL:omg.org/Components/CCMHome:1.0**.

When it is invoked on an eventtype **TypeCode** that contains a **Components::EventBase**, the returned value is **IDL:omg.org/Components/EventBase:1.0.**

The **name** operation can also be invoked on object reference, structure, union, enumeration, alias, abstract interface, local interface, value type, boxed valuetype, native, and exception **TypeCode**s. It returns the simple name identifying the type within its enclosing scope. Since names are local to a **Repository**, the name returned from a **TypeCode** may not match the name of the type in any particular **Repository**, and may even be an empty string.

The order in which members are presented in the interface repository is the same as the order in which they appeared in the IDL specification, and this ordering determines the index value for each member. The first member has index value 0. For example for a structure definition:

```
struct example {
    short    member1;
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        short   member2;
        long    member3;
    };
```

In this example **member1** has **index** = 0, **member2** has **index** = 1, and **member3** has **index** = 2. The value of **member_count** in this case is 3.

The **member_count** and **member_name** operations can be invoked on structure, union, non-boxed valuetype, non-boxed eventtype, exception, and enumeration **TypeCode**s. **Member_count** returns the number of members constituting the type. **Member_name** returns the simple name of the member identified by **index**. Since names are local to a **Repository**, the name returned from a **TypeCode** may not match the name of the member in any particular **Repository**, and may even be an empty string.

The **member_type** operation can be invoked on structure, non-boxed valuetype, non-boxed eventtype, exception and union **TypeCode**s. It returns the **TypeCode** describing the type of the member identified by **index**.

The **member_label**, **discriminator_type**, and **default_index** operations can only be invoked on union **TypeCode**s. **Member_label** returns the label of the union member identified by **index**. For the default member, the label is the zero octet. The **discriminator_type** operation returns the type of all non-default member labels. The **default_index** operation returns the index of the default member, or -1 if there is no default member.

The **member_visibility** operation can only be invoked on non-boxed valuetype and non-boxed eventtype, **TypeCodes**. It returns the **Visibility** of the valuetype/eventtype member identified by index.

The **member_name**, **member_type**, **member_label** and **member_visibility** operations raise **Bounds** if the index parameter is greater than or equal to the number of members constituting the type.

The **content_type** operation can be invoked on sequence, array, boxed valuetype and alias **TypeCode**s. For sequences and arrays, it returns the element type. For aliases, it returns the original type. For boxed valuetype, it returns the boxed type.

An array **TypeCode** only describes a single dimension of an OMG IDL array. Multi-dimensional arrays are represented by nesting **TypeCode**s, one per dimension. The outermost **tk_array Typecode** describes the leftmost array index of the array as defined in IDL. Its **content_type** describes the next index. The innermost nested **tk_array TypeCode** describes the rightmost index and the array element type.

The **type_modifier** and **concrete_base_type** operations can be invoked on non-boxed valuetype and non-boxed eventtype**TypeCode**s. The **type_modifier** operation returns the **ValueModifier** that applies to the valuetype/eventtype represented by the target **TypeCode**. If the valuetype/eventtype represented by the target **TypeCode** has a concrete base valuetype/eventtype, the **concrete_base_type** operation returns a **TypeCode** for the concrete base, otherwise it returns a nil **TypeCode** reference.

The **length** operation can be invoked on string, wide string, sequence, and array **TypeCode**s. For strings and sequences, it returns the bound, with zero indicating an unbounded string or sequence. For arrays, it returns the number of elements in the array. For wide strings, it returns the bound, or zero for unbounded wide strings.

8.11.2 TypeCode Constants

For IDL type declarations, the IDL compiler produces (if asked) a declaration of a **TypeCode** constant. See the language mapping rules for more information about the names of the generated **TypeCode** constants. **TypeCode** constants include tk_alias definitions wherever an IDL typedef is referenced. These constants can be used with the dynamic invocation interface and other routines that require **TypeCode**s.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The predefined **TypeCode** constants, named according to the C language mapping, are:

TC_null
TC_void
TC_short
TC_long
TC_longlong
TC_ushort
TC_ulong
TC_ulonglong
TC_float
TC_double
TC_longdouble
TC_boolean
TC_char
TC_wchar
TC_octet
TC_any
TC_TypeCode
TC_Object = tk_objref {Object}
TC_string= tk_string {0} // unbounded
TC_wstring = tk_wstring{0}/// unbounded
TC_ValueBase = tk_value {ValueBase}
TC_Component = tk_component {CCMObject}
TC_Home = tk_home {CCMHome}
TC_EventBase = tk_event {EventBase}

For the **TC_Object TypeCode** constant, calling **id** returns "**IDL:omg.org/CORBA/Object:1.0**" and calling **name** returns "**Object.**"

For the **TC_ValueBase TypeCode** constant, calling **id** returns "**IDL:omg.org/CORBA/ValueBase:1.0,**" calling **name** returns "**ValueBase,**" calling **member_count** returns **0**, calling **type_modifier** returns **CORBA::VM_NONE**, and calling **concrete_base_type** returns a **nil TypeCode**.

For the **TC_Component TypeCode** constant, calling **id** returns "**IDL:omg.org/Components/CCMObject:1.0**" and calling **name** returns "**CCMObject**."

For the **TC_Home TypeCode** constant, calling **id** returns "**IDL:omg.org/Components/CCMHome:1.0**" and calling **name** returns "**CCMHome**."

For the **TC_EventBase TypeCode** constant, calling **id** returns "**IDL:omg.org/Components/EventBase:1.0**," calling **name** returns "**EventBase**," calling **member_count** returns **0**, calling **type_modifier** returns **CORBA::VM_NONE**, and calling **concrete_base_type** returns a **nil TypeCode**.

8.11.3 Creating TypeCodes

When creating type definition objects in an Interface Repository, types are specified in terms of object references, and the **TypeCode**s describing them are generated automatically.

In some situations, such as bridges between ORBs, **TypeCode**s need to be constructed outside of any Interface Repository. This can be done using operations on the **ORB** pseudo-object.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
module CORBA {
    interface ORB {
        // other operations ...

        TypeCode create_struct_tc (
            in RepositoryId         id;
            in Identifier           name,
            in StructMemberSeq      members
        );

        TypeCode create_union_tc (
            in RepositoryId         id,
            in Identifier           name,
            in TypeCode             discriminator_type,
            in UnionMemberSeq       members
        );

        TypeCode create_enum_tc (
            in RepositoryId         id,
            in Identifier           name,
            in EnumMemberSeq        members
        );

        TypeCode create_alias_tc (
            in RepositoryId         id,
            in Identifier           name,
            in TypeCode             original_type
        );

        TypeCode create_exception_tc (
            in RepositoryId         id,
            in Identifier           name,
            in StructMemberSeq      members
        );

        TypeCode create_interface_tc (
            in RepositoryId         id,
            in Identifier           name
        );

        TypeCode create_string_tc (
            in unsigned long        bound
        );

        TypeCode create_wstring_tc (
            in unsigned long        bound
        );
```


© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
TypeCode create_fixed_tc (
    in unsigned short       digits,
    in unsigned short       scale
);

TypeCode create_sequence_tc (
    in unsigned long        bound,
    in TypeCode             element_type
);

TypeCode create_recursive_sequence_tc (// deprecated
    in unsigned long        bound,
    in unsigned long        offset
);

TypeCode create_array_tc (
    in unsigned long        length,
    in TypeCode             element_type
);

TypeCode create_value_tc (
    in RepositoryId         id,
    in Identifier           name,
    in ValueModifier        type_modifier,
    in TypeCode             concrete_base,
    in ValueMemberSeq       members
);

TypeCode create_value_box_tc (
    in RepositoryId         id,
    in Identifier           name,
    in TypeCode             boxed_type
);

TypeCode create_native_tc (
    in RepositoryId         id,
    in Identifier           name
);

TypeCode create_recursive_tc(
    in RepositoryId         id
);

TypeCode create_abstract_interface_tc(
    in RepositoryId         id,
    in Identifier           name
);
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        TypeCode create_local_interface_tc(
            in RepositoryId         id,
            in Identifier           name
        );

        TypeCode create_component_tc (
            in RepositoryId         id,
            in Identifier           name
        );

        TypeCode create_home_tc (
            in RepositoryId         id,
            in Identifier           name
        );

        TypeCode create_event_tc (
            in RepositoryId         id,
            in Identifier           name,
            in ValueModifier        type_modifier,
            in TypeCode             concrete_base,
            in ValueMemberSeq       members
        );
    };
};
```

Most of these operations are similar to corresponding IR operations for creating type definitions. **TypeCode**s are used here instead of **IDLType** object references to refer to other types. In the **StructMember**, **UnionMember**, and **ValueMember** structures, only the **type** is used, and the **type_def** should be set to nil.

Typecode creation operations that take **name** as an argument shall check that the name is a valid IDL name or is an empty string. If not, they shall raise the BAD_PARAM exception with standard minor code 15. Operations that take a **RepositoryId** argument shall check that the argument passed in is a string of the form **<format>:<string>** and if not, then raise a BAD_PARAM exception with standard minor code 16. Operations that take **content** or **member** types as arguments shall check that they are legitimate (i.e., that they don't have kinds **tk_null**, **tk_void**, or **tk_exception)**. If not, they shall raise the BAD_TYPECODE exception with standard minor code 2. Operations that take members shall check that the member names are valid IDL names and that they are unique within the member list, and if the name is found to be incorrect, they shall raise a BAD_PARAM with standard minor code 17.

The **create_union_tc** operation shall check that there are no duplicate label values. It shall also check that each label **TypeCode** compares equivalent to the discriminator **TypeCode**. If a duplicate label is found, raise BAD_PARAM with standard minor code 18. If the **TypeCode** of a label is not equivalent to the **TypeCode** of the discriminator (other than the **octet TypeCode** to indicate the default label), the operation shall raise **BAD_PARAM** with standard minor code 19. The **create_union_tc** operation shall also check that the supplied discriminator type is legitimate, and if the check fails, raise BAD_PARAM with standard minor code 20.

NOTE: The **create_recursive_sequence_tc** operation is deprecated. No new code should make use of this operation. Its functionality is subsumed by the new operation **create_recursive_tc**. The **create_recursive_sequence_tc** operation will be removed from a future revision of the standard.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The **create_recursive_sequence_tc** operation is used to create **TypeCodes** describing recursive sequences that are members of structs or unions. The result of this operation should be used as the typecode in the **StructMemberSeq** or **UnionMemberSeq** arguments of the **create_struct_tc** or **create_union_tc** operations. The **offset** parameter specifies which enclosing struct or union is the target of the recursion, with the value **1** indicating the most immediate enclosing struct or union, and larger values indicating successive enclosing struct or unions. For example, the offset would be **1** for the following IDL structure:

```
struct foo {
    long value;
    sequence <foo> chain;
};
```

Once the recursive sequence **TypeCode** has been properly embedded in its enclosing **TypeCodes**, it will function as a normal sequence **TypeCode**. Invoking operations on the recursive sequence **TypeCode** before it has been embedded in the required number of enclosing **TypeCodes** will result in undefined behavior. Attempt to marshal incomplete typecodes shall raise the BAD_TYPECODE exception with standard minor code 1. Attempt to use an incomplete **TypeCode** as a parameter of any operation when detected shall cause the BAD_PARAM exception to be raised with standard minor code 13.

For **create_value_tc** operation, the **concrete_base** parameter is a **TypeCode** for the immediate concrete valuetype base of the valuetype for which the **TypeCode** is being created. If the valuetype does not have a concrete base, the **concrete_base** parameter is a nil **TypeCode** reference.

The **create_recursive_tc** operation is used to create a recursive **TypeCode**, which serves as a place holder for a concrete **TypeCode** during the process of creating **TypeCode**s that contain recursion. The **id** parameter specifies the repository id of the type for which the recursive **TypeCode** is serving as a place holder. Once the recursive **TypeCode** has been properly embedded in the enclosing **TypeCode**, which corresponds to the specified repository id, it will function as a normal **TypeCode**. Invoking operations on the recursive **TypeCode** before it has been embedded in the enclosing **TypeCode** will result in undefined behavior. For example, the following IDL type declarations contain recursion:

```
struct foo {
    long value;
    sequence<foo> chain;
};

valuetype V {
    public V member;
};
```

To create a **TypeCode** for **valuetype V**, you would invoke the **TypeCode** creation operations as shown below:

```
// C++
TypeCode_var recursive_tc
   = orb->create_recursive_tc("IDL:V:1.0");

ValueMemberSeq v_seq;
v_seq.length(1);
v_seq[0].name = string_dup("member");
v_seq[0].type = recursive_tc;
v_seq[0].access = PUBLIC_MEMBER;
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
TypeCode_var v_val_tc
   = orb->create_value_tc("IDL:V:1.0",
                  "V",
                  VM_NONE,
                  TypeCode::_nil(),
   v_seq);
```

For **create_event_tc** operation, the **concrete_base** parameter is a **TypeCode** for the immediate concrete base of the eventtype for which the **TypeCode** is being created. If the eventtype does not have a concrete base, the **concrete_base** parameter is a nil **TypeCode** reference.

8.12 Exceptions

The terms "system" and "user" exception are defined in this sub clause. Further the terms "standard system exception" and "standard user exception" are defined, and then a list of "standard system exceptions" is provided.

8.12.1 Definition of Terms

In general the following terms should be used consistently in all OMG standards documents to refer to exceptions:

Standard Exception: Any exception that is defined in an OMG Standard.

System Exception: Clients must be prepared to handle these exceptions even though they are not declared in a raises clause. These exceptions cannot appear in a raises clause. These have the structure defined in Annex A and they are of type **SYSTEM_EXCEPTION** (see PIDL below).

Standard System Exception: A System Exception that is part of the CORBA Standard (e.g., BAD_PARAM). See Annex A for more details.

Non-Standard System Exceptions: System exceptions that are proprietary to a particular vendor/implementation.

User Exception: Exceptions that can be raised only by those operations that explicitly declare them in the raises clause of their signature. These exceptions are of type USER_EXCEPTION (see IDL below).

Standard User Exception: Any User Exception that is defined in a published OMG standard (e.g., WrongTransaction). These are documented in the documentation of individual interfaces.

Non-standard User Exception: User exceptions that are not defined in any published OMG specification.

8.12.2 System Exceptions

In order to bound the complexity in handling the standard exceptions, the set of standard exceptions should be kept to a tractable size. This constraint forces the definition of equivalence classes of exceptions rather than enumerating many similar exceptions. For example, an operation invocation can fail at many different points due to the inability to allocate dynamic memory. Rather than enumerate several different exceptions corresponding to the different ways that memory allocation failure causes the exception (during marshaling, unmarshaling, in the client, in the object implementation, allocating network packets), a single exception corresponding to dynamic memory allocation failure is defined.

```
module CORBA {
    const unsigned long OMGVMCID = 0x4f4d0000;
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
#define ex_body {unsigned long minor; completion_status completed;}

    enum completion_status {
        COMPLETED_YES,
        COMPLETED_NO,
        COMPLETED_MAYBE
    };

    enum exception_type {
        NO_EXCEPTION,
        USER_EXCEPTION,


        SYSTEM_EXCEPTION
    };
};
```

Each system exception includes a minor code to designate the subcategory of the exception.

Minor exception codes are of type **unsigned long** and consist of a 20-bit "Vendor Minor Codeset ID"(**VMCID**), which occupies the high order 20 bits, and the minor code that occupies the low order 12 bits.

The standard minor codes for the standard system exceptions are prefaced by the **VMCID** assigned to OMG, defined as the unsigned long constant **CORBA::OMGVMCID**, which has the VMCID allocated to OMG occupying the high order 20 bits. The minor exception codes associated with the standard exceptions that are found in Annex A, "Exception Codes" are or-ed with **OMGVMCID** to get the minor code value that is returned in the **ex_body** structure (see Standard System Exception Definitions on page 150 and Standard Minor Exception Codes on page 156).

Within a vendor assigned space, the assignment of values to minor codes is left to the vendor. Vendors may request allocation of **VMCID**s by sending email to tag-*request@omg.org*.

The **VMCID** *0* and *0xfffff* are reserved for experimental use. The **VMCID OMGVMCID** (8.12.3, Standard System Exception Definitions) and *1 through 0xf* are reserved for OMG use.

Each standard system exception also includes a **completion_status** code that takes one of the values {COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE}. These have the following meanings:

COMPLETED_YES	The object implementation has completed processing prior to the exception being raised.
COMPLETED_NO	The object implementation was never initiated prior to the exception being raised.
COMPLETED_MAYBE	The status of implementation completion is indeterminate.

Client applications must be prepared to handle system exceptions other than the standard system exception defined below in Standard System Exception Definitions on page 150, both because future versions of this specification may define additional standard system exceptions, and because ORB implementations may raise non-standard system exceptions.

Vendors may define non-standard system exceptions, but these exceptions are discouraged because they are non-portable. A non-standard system exception, when passed to an ORB that does not recognize it, shall be presented by that ORB as an **UNKNOWN** standard system exception. The completion status shall be preserved in the **UNKNOWN** exception, and the minor code shall be set to standard value 2 for system exception and standard value 1 for user exception.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Non-standard system exceptions shall have the same structure as of standard standard system exceptions as specified in Standard System Exception Definitions on page 150 (i.e., they have the same ex_body). They also shall follow the same language mappings as standard system exceptions. Although they are PIDL, vendors should ensure that their names do not clash with any other names following the normal naming and scoping rules as they apply to regular IDL exceptions.

8.12.3 Standard System Exception Definitions

The standard system exceptions are defined in this sub clause.

```
module CORBA {                  // PIDL

    exception UNKNOWN ex_body;
                                // the unknown exception
    exception BAD_PARAM ex_body;
                                // an invalid parameter was passed
    exception NO_MEMORY ex_body;
                                // dynamic memory allocation failure
    exception IMP_LIMIT ex_body;
                                // violated implementation limit
    exception COMM_FAILURE ex_body;
                                // communication failure
    exception INV_OBJREF ex_body;
                                // invalid object reference
    exception NO_PERMISSION ex_body;
                                // no permission for attempted op.
    exception INTERNAL ex_body;
                                // ORB internal error
    exception MARSHAL ex_body;
                                // error marshaling param/result
    exception INITIALIZE ex_body;
                                // ORB initialization failure
    exception NO_IMPLEMENT ex_body;
                                // operation implementation unavailable
    exception BAD_TYPECODE ex_body;
                                // bad typecode
    exception BAD_OPERATION ex_body;
                                // invalid operation
    exception NO_RESOURCES ex_body;
                                // insufficient resources for req.
    exception NO_RESPONSE ex_body;
                                // response to req. not yet available
    exception PERSIST_STORE ex_body;
                                // persistent storage failure
    exception BAD_INV_ORDER ex_body;
                                // routine invocations out of order
    exception TRANSIENT ex_body;
                                // transient failure - reissue request
    exception FREE_MEM ex_body;
                                // cannot free memory
    exception INV_IDENT ex_body;
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
                                    // invalid identifier syntax
    exception INV_FLAG ex_body;
                                    // invalid flag was specified
    exception INTF_REPOS ex_body;
                                    // error accessing interface repository
    exception BAD_CONTEXT ex_body;
                                    // error processing context object
    exception OBJ_ADAPTER ex_body;
                                    // failure detected by object adapter
    exception DATA_CONVERSION ex_body;
                                    // data conversion error
    exception OBJECT_NOT_EXIST ex_body;
                                    // non-existent object, delete reference
    exception TRANSACTION_REQUIRED ex_body;
                                    // transaction required
    exception TRANSACTION_ROLLEDBACK x_body;
                                    // transaction rolled back
    exception INVALID_TRANSACTION ex_body;
                                    // invalid transaction
    exception INV_POLICY ex_body;
                                    // invalid policy
    exception CODESET_INCOMPATIBLE ex_body
                                        // incompatible code set
    exception REBIND ex_body;
                                        // rebind needed
    exception TIMEOUT ex_body;
                                        // operation timed out
    exception TRANSACTION_UNAVAILABLE ex_body;
                                        // no transaction
    exception TRANSACTION_MODE ex_body;
                                        // invalid transaction mode
    exception BAD_QOS ex_body;
                                        // bad quality of service
    exception INVALID_ACTIVITY ex_body;
                                        // bad quality of service
    exception ACTIVITY_COMPLETED ex_body;
                                        // bad quality of service
    exception ACTIVITY_REQUIRED ex_body;
                                        // bad quality of service
};
```

8.12.3.1 UNKNOWN

This exception is raised if an operation implementation throws a non-CORBA exception (such as an exception specific to the implementation's programming language), or if an operation raises a user exception that does not appear in the operation's raises expression. UNKNOWN is also raised if the server returns a system exception that is unknown to the client. (This can happen if the server uses a later version of CORBA than the client and new system exceptions have been added to the later version.)

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.12.3.2 BAD_PARAM

A parameter passed to a call is out of range or otherwise considered illegal. An ORB may raise this exception if null values or null pointers are passed to an operation (for language mappings where the concept of a null pointers or null values applies). BAD_PARAM can also be raised as a result of client generating requests with incorrect parameters using the DII.

8.12.3.3 NO_MEMORY

The ORB run time has run out of memory.

8.12.3.4 IMP_LIMIT

This exception indicates that an implementation limit was exceeded in the ORB run time. For example, an ORB may reach the maximum number of references it can hold simultaneously in an address space, the size of a parameter may have exceeded the allowed maximum, or an ORB may impose a maximum on the number of clients or servers that can run simultaneously.

8.12.3.5 COMM_FAILURE

This exception is raised if communication is lost while an operation is in progress, after the request was sent by the client, but before the reply from the server has been returned to the client.

8.12.3.6 INV_OBJREF

This exception indicates that an object reference is internally malformed. For example, the repository ID may have incorrect syntax or the addressing information may be invalid.

An ORB may choose to detect calls via nil references (but is not obliged to detect them). INV_OBJREF is used to indicate this.

If the client invokes an operation that results in an attempt by the client ORB to marshal wchar or wstring data for an in parameter (or to unmarshal wchar or wstring data for an in/out parameter, out parameter or the return value), and the associated object reference does not contain a codeset component, the INV_OBJREF standard system exception is raised.

8.12.3.7 NO_PERMISSION

An invocation failed because the caller has insufficient privileges.

8.12.3.8 INTERNAL

This exception indicates an internal failure in an ORB, for example, if an ORB has detected corruption of its internal data structures.

8.12.3.9 MARSHAL

A request or reply from the network is structurally invalid. This error typically indicates a bug in either the client-side or server-side run time. For example, if a reply from the server indicates that the message contains 1000 bytes, but the actual message is shorter or longer than 1000 bytes, the ORB raises this exception. MARSHAL can also be caused by using the DII or DSI incorrectly, for example, if the type of the actual parameters sent does not agree with IDL signature of an operation.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.12.3.10 INITIALIZE

An ORB has encountered a failure during its initialization, such as failure to acquire networking resources or detecting a configuration error.

8.12.3.11 NO_IMPLEMENT

This exception indicates that even though the operation that was invoked exists (it has an IDL definition), no implementation for that operation exists. NO_IMPLEMENT can, for example, be raised by an ORB if a client asks for an object's type definition from the interface repository, but no interface repository is provided by the ORB.

8.12.3.12 BAD_TYPECODE

The ORB has encountered a malformed type code (for example, a type code with an invalid **TCKind** value).

8.12.3.13 BAD_OPERATION

This indicates that an object reference denotes an existing object, but that the object does not support the operation that was invoked.

8.12.3.14 NO_RESOURCES

The ORB has encountered some general resource limitation. For example, the run time may have reached the maximum permissible number of open connections.

8.12.3.15 NO_RESPONSE

This exception is raised if a client attempts to retrieve the result of a deferred synchronous call, but the response for the request is not yet available.

8.12.3.16 PERSIST_STORE

This exception indicates a persistent storage failure, for example, failure to establish a database connection or corruption of a database.

8.12.3.17 BAD_INV_ORDER

This exception indicates that the caller has invoked operations in the wrong order. For example, it can be raised by an ORB if an application makes an ORB-related call without having correctly initialized the ORB first.

8.12.3.18 TRANSIENT

TRANSIENT indicates that the ORB attempted to reach an object and failed. It is not an indication that an object does not exist. Instead, it simply means that no further determination of an object's status was possible because it could not be reached. This exception is raised if an attempt to establish a connection fails, for example, because the server or the implementation repository is down.

8.12.3.19 FREE_MEM

The ORB failed in an attempt to free dynamic memory, for example because of heap corruption or memory segments being locked.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.12.3.20 INV_IDENT

This exception indicates that an IDL identifier is syntactically invalid. It may be raised if, for example, an identifier passed to the interface repository does not conform to IDL identifier syntax, or if an illegal operation name is used with the DII.

8.12.3.21 INV_FLAG

An invalid flag was passed to an operation (for example, when creating a DII request).

8.12.3.22 INTF_REPOS

An ORB raises this exception if it cannot reach the interface repository, or some other failure relating to the interface repository is detected.

8.12.3.23 BAD_CONTEXT

An operation may raise this exception if a client invokes the operation but the passed context does not contain the context values required by the operation.

8.12.3.24 OBJ_ADAPTER

This exception typically indicates an administrative mismatch. For example, a server may have made an attempt to register itself with an implementation repository under a name that is already in use, or is unknown to the repository. OBJ_ADAPTER is also raised by the POA to indicate problems with application-supplied servant managers.

8.12.3.25 DATA_CONVERSION

This exception is raised if an ORB cannot convert the representation of data as marshaled into its native representation or vice-versa. For example, DATA_CONVERSION can be raised if wide character codeset conversion fails, or if an ORB cannot convert floating point values between different representations.

8.12.3.26 OBJECT_NOT_EXIST

The OBJECT_NOT_EXIST exception is raised whenever an invocation on a deleted object was performed. It is an authoritative “hard” fault report. Anyone receiving it is allowed (even expected) to delete all copies of this object reference and to perform other appropriate “final recovery” style procedures.

Bridges forward this exception to clients, also destroying any records they may hold (for example, proxy objects used in reference translation). The clients could in turn purge any of their own data structures.

8.12.3.27 TRANSACTION_REQUIRED

The TRANSACTION_REQUIRED exception indicates that the request carried a null transaction context, but an active transaction is required.

8.12.3.28 TRANSACTION_ROLLEDBACK

The TRANSACTION_ROLLEDBACK exception indicates that the transaction associated with the request has already been rolled back or marked to roll back. Thus, the requested operation either could not be performed or was not performed because further computation on behalf of the transaction would be fruitless.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.12.3.29 INVALID_TRANSACTION

The INVALID_TRANSACTION indicates that the request carried an invalid transaction context. For example, this exception could be raised if an error occurred when trying to register a resource.

8.12.3.30 INV_POLICY

INV_POLICY is raised when an invocation cannot be made due to an incompatibility between Policy overrides that apply to the particular invocation.

8.12.3.31 CODESET_INCOMPATIBLE

This exception is raised whenever meaningful communication is not possible between client and server native code sets. See *CORBA, Part II* - ORB Interoperability Architecture.

8.12.3.32 REBIND

REBIND is raised when the current effective **RebindPolicy**, as described in interface RebindPolicy on page 420, has a value of **NO_REBIND** or **NO_RECONNECT** and an invocation on a bound object reference results in a LocateReply message with status **OBJECT_FORWARD** or a Reply message with status **LOCATION_FORWARD**. This exception is also raised if the current effective RebindPolicy has a value of **NO_RECONNECT** and a connection must be re-opened. The invocation can be retried once the effective **RebindPolicy** is changed to **TRANSPARENT** or binding is re-established through an invocation of **CORBA::Object::validate_connection**.

REBIND is raised when there is a problem in carrying out a requested or implied attempt to rebind an object reference (interface RebindPolicy on page 420).

8.12.3.33 TIMEOUT

TIMEOUT is raised when no delivery has been made and the specified time-to-live period has been exceeded. It is a standard system exception because time-to-live QoS can be applied to any invocation.

8.12.3.34 TRANSACTION_UNAVAILABLE

TRANSACTION_UNAVAILABLE exception is raised by the ORB when it cannot process a transaction service context because its connection to the Transaction Service has been abnormally terminated.

8.12.3.35 TRANSACTION_MODE

TRANSACTION_MODE exception is raised by the ORB when it detects a mismatch between the **TransactionPolicy** in the IOR and the current transaction mode.

8.12.3.36 BAD_QOS

The BAD_QOS exception is raised whenever an object cannot support the quality of service required by an invocation parameter that has a quality of service semantics associated with it.

8.12.3.37 INVALID_ACTIVITY

The INVALID_ACTIVITY system exception may be raised on the Activity or Transaction services' resume methods if a transaction or Activity is resumed in a context different to that from which it was suspended. It is also raised when an attempted invocation is made that is incompatible with the Activity's current state.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.12.3.38 ACTIVITY_COMPLETED

The ACTIVITY_COMPLETED system exception may be raised on any method for which Activity context is accessed. It indicates that the Activity context in which the method call was made has been completed due to a timeout of either the Activity itself or a transaction that encompasses the Activity, or that the Activity completed in a manner other than that originally requested.

8.12.3.39 ACTIVITY_REQUIRED

The ACTIVITY_REQUIRED system exception may be raised on any method for which an Activity context is required. It indicates that an Activity context was necessary to perform the invoked operation, but one was not found associated with the calling thread.

8.12.4 Standard Minor Exception Codes

Please refer to Annex A for a table that specifies standard minor exception codes that have been assigned for the standard system exceptions.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9 Value Type Semantics

9.1 Overview

Objects, more specifically, interface types that objects support, are defined by an IDL interface, allowing arbitrary implementations. There is great value, which is described in great detail elsewhere, in having a distributed object system that places almost no constraints on implementations.

However there are many occasions in which it is desirable to be able to pass an object by value, rather than by reference. This may be particularly useful when an object's primary "purpose" is to encapsulate data, or an application explicitly wishes to make a "copy" of an object.

The semantics of passing an object by value are similar to that of standard programming languages. The receiving side of a parameter passed by value receives a description of the "state" of the object. It then instantiates a new instance with that state but having a separate identity from that of the sending side. Once the parameter passing operation is complete, no relationship is assumed to exist between the two instances.

Because it is necessary for the receiving side to instantiate an instance, it must necessarily know something about the object's state and implementation.

Value types provide semantics that bridge between CORBA structs and CORBA interfaces:

- They support description of complex state (i.e., arbitrary graphs, with recursion and cycles).
- Their instances are always local to the context in which they are used (because they are always copied when passed as a parameter to a remote call).
- They support both public and private (to the implementation) data members.
- They can be used to specify the state of an object implementation (i.e., they can support an interface).
- They support single inheritance (of **valuetype**) and can support an **interface**.
- They may be also be **abstract**.

9.2 Architecture

The basic notion is relatively simple. A **value type** is, in some sense, half way between a "regular" IDL interface type and a struct. The use of a value type is a signal from the designer that some additional properties (state) and implementation details be specified beyond that of an interface type. Specification of this information puts some additional constraints on the implementation choices beyond that of interface types. This is reflected in both the semantics specified herein, and in the language mappings.

An essential property of value types is that their implementations are always local. That is, the explicit use of value type in a concrete programming language is always guaranteed to use a local implementation, and will not require a remote call. They have no identity (their value is their identity) and they are not "registered" with the ORB.

There are two kinds of value types, concrete (or stateful) value types, and abstract (stateless) ones. As explained below the essential characteristics of both are the same. The differences between them result from the differences in the way they are mapped in the language mappings. In this specification the semantics of value types apply to both kinds, unless specifically stated otherwise.

Concrete (stateful) values add to the expressive power of (IDL) structs by supporting:

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- Single derivation (from other value types).
- Supports a single non-abstract interface.
- Arbitrary recursive value type definitions, with sharing semantics providing the ability to define lists, trees, lattices, and more generally arbitrary graphs using value types.
- Null value semantics.

When an instance of such a type is passed as a parameter, the sending context marshals the state (data) and passes it to the receiving context. The receiving context instantiates a new instance using the information in the GIOP request and unmarshals the state. It is assumed that the receiving context has available to it an implementation that is consistent with the sender's (i.e., only needs the state information), or that it can somehow download a usable implementation. Provision is made in the on-the-wire format to support the carrying of an optional call back object (**CodeBase**) to the sending context, which enables such downloading when it is appropriate.

It should be noted that it is possible to define a concrete value type with an empty state as a degenerate case.

9.2.1 Abstract Values

Value types may also be abstract. They are called abstract because an abstract value type may not be instantiated. Only concrete types derived from them may be actually instantiated and implemented. Their implementation, of course, is still local. However, because no state information may be specified (only local operations are allowed), abstract value types are not subject to the single inheritance restrictions placed upon concrete value types. Essentially they are a bundle of operation signatures with a purely local implementation. This distinction is made clear in the language mappings for abstract values.

Note that a concrete value type with an empty state is not an abstract value type. They are considered to be stateful, may be instantiated, marshaled, and passed as actual parameters. Consider them to be a degenerate case of stateful values.

9.2.2 Operations

Operations defined on a value type specify signatures whose implementation can only be local. Because these operations are local, they must be directly implemented by a body of code in the language mapping (no proxy or indirection is involved).

The language mappings of such operations require that instances of value types passed into and returned by such local methods are passed by reference (programming language reference semantics, not CORBA object reference semantics) and that a copy is not made. Note, such a (local) invocation is not a CORBA invocation. Hence it is not mediated by the ORB, although the API to be used is specified in the language mapping.

The (copy) semantics for instances of value type are only guaranteed when instances of these value types are passed as a parameter to an operation defined on a CORBA interface, and hence mediated by the ORB. If an instance of a value type is passed as a parameter to a method of another value type in an invocation, then this call is a "normal" programming language call. In this case both of the instances are local programming language constructs. No CORBA style copy semantics are used and programming language reference semantics apply.

Operations on the value type are supported in order to guarantee the portability of the client code for these value types. They have no representation on the wire and hence no impact on interoperability.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.2.3 Value Type vs. Interfaces

By default value types are not CORBA Objects. In particular, instances of value types do not inherit from **CORBA::Object** and do not support normal object reference semantics. However it is always possible to explicitly declare that a given value type supports an interface type. In this case instances of the type may support CORBA object reference semantics (if they are registered with the ORB using an object adapter).

9.2.4 Parameter Passing

This sub clause describes semantics when a value instance is passed as parameter in a CORBA invocation. It does not deal with the case of calling another non-CORBA (i.e., local) programming method, which happens to have a parameter of the same type.

9.2.4.1 Value vs. Reference Semantics

Determination of whether a parameter is to be passed by value or reference is made by examining the parameter's formal type (i.e., the signature of the operation it is being passed to). If it is a value type, then it is passed by value. If it is an ordinary interface, then it is passed by reference (the case today for all CORBA objects). This rule is simple and consistent with the handling of the same situation in recursive state definitions or in structs.

In the case of abstract interfaces, the determination is made at runtime. See Semantics of Abstract Interfaces on page 173 for a description of the rules.

9.2.4.2 Sharing Semantics

In order to be expressive enough to describe arbitrary graphs, lattice, trees, etc., value types support sharing and null semantics. Instances of a value type can be shared by others across or within other instances. They can also be null. This is unlike other IDL data types such as structs, unions, and sequences that can never be shared. The sharing of values within and between the parameters to an operation is preserved across an invocation; that is, the graph that is reconstructed in the receiving context is structurally isomorphic to the sending context's.

9.2.4.3 Identity Semantics

When an instance of the value type is passed as a parameter to an operation of a non-local interface, the effect in all cases shall be as if an independent copy of the instance is instantiated in the receiving context. While certain implementation optimizations are possible the net effect shall be as if the copy is a separate independent entity and there is no explicit or implicit sharing of state. This applies to all valuetypes involved in the invocation, including those embedded in other IDL datatypes or in an any. This notional copying occurs twice, once for in and inout parameters when the invocation is initiated, and once again for inout, out, and return parameters when the invocation completes. Optimization techniques such as copy on write, etc. must make sure that the semantics of copying as described above is preserved.

9.2.4.4 Any parameter type

When an instance of a value type is passed to an **any**, as with all cases of passing instances to an **any**, it is the responsibility of the implementor to insert and extract the value according to the language mapping specification.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.2.5 Substitutability Issues

The substitutability requirements for CORBA require the definition of what happens when an instance of a derived value type is passed as a parameter that is declared to be a base value type or an instance of a value type that supports an interface is passed as a parameter that is declared as the interface type.

There are three cases to consider: the parameter type is a regular interface, the parameter type is an abstract interface, and the parameter type is a value type.

9.2.5.1 Value instance -> Interface type

A value type that supports a regular interface is not a subtype of that interface, and hence cannot be substituted for that interface in an invocation parameter. In this case an object reference corresponding to the value type instance that has been registered with the ORB must be obtained and this object reference must be used as the actual parameter. Different language mappings provide different facilities to aid in such parameter passing.

9.2.5.2 Value Instance -> Abstract interface type

A value type that supports an abstract interface is a subtype of that interface, and can be substituted for that interface in an invocation parameter.

9.2.5.3 Value instance -> Value type

In this case the receiving context is expecting to receive a value type. If the receiving context currently has the appropriate implementation class, then there is no problem.

If the receiving context does not currently hold an implementation with which to reconstruct the original type, then the following algorithm is used to find such an implementation:

1. ***Load*** - Attempt to load (locally in C/C++, possibly remotely in Java and other “portable” languages) the real type of the object (with its methods). If this succeeds, OK.

2. ***Truncate*** - Truncate the type of the object to the base type (if specified as **truncatable** in the IDL). Truncation can never lead to faulty programs because, from a structural point view base types structurally subsume a derived type and an object created in the receiving context bears no relationship with the original one. However, it might be semantically puzzling, as the derived type may completely re-interpret the meaning of the state of the base. For that reason a derived value needs to indicate if it is safe to truncate to its immediate non-abstract parent.

3. ***Raise Exception*** - If none of these work or are possible, then raise the NO_IMPLEMENT exception with standard minor code 1.

Truncatability is a transitive property.

Example

```
valuetype EmployeeRecord {  // note this is not a CORBA::Object
    // state definition
    private string name;
    private string email;
    private string SSN;
    // initializer
    factory init(in string name, in string SSN);
};
```


© ISO/IEC 2008 - All rights reserved
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
valuetype ManagerRecord: truncatable EmployeeRecord {
    // state definition
    private sequence<EmployeeRecord> direct_reports;
};
```

9.2.6 Widening/Narrowing

As has been described above, value type instances may be widened/narrowed to other value types. Each language mapping is responsible for specifying how these operations are made available to the programmer.

Narrowing from an interface type instance to a value type instance is not allowed. If the interface designer wants to allow the receiving context to create a local implementation of the value type (i.e., a value representing the interface), an operation that returns the appropriate value type may be defined.

9.2.7 Value Base Type

All value types have a conventional base type called **ValueBase**. This is a type, which fulfills a role that is similar to that played by **Object**. Conceptually it supports the common operations available on all value types. See ValueBase Operations on page 114 for a description of those operations. In each language mapping **ValueBase** will be mapped to an appropriate base type that supports the marshaling/unmarshaling protocol as well as the model for custom marshaling.

The mapping for other operations, which all value types must support, such as getting meta information about the type, may be found in the specifics for each language mapping.

9.2.8 Life Cycle issues

Value type instances are always local to their creating context. For example, in a given language mapping an instance of a value type is always created as a local “language” object with no POA semantics attached to it initially.

When passed using a CORBA invocation, a copy of the value is made in the receiving context and that copy starts its life as a local programming language entity with no POA semantics attached to it.

If a value type supports an ordinary interface type, its instances may also be passed by reference when the formal parameter type is an interface type (see Parameter Passing on page 159). In this case they behave like ordinary object implementations and must be associated with a POA policy and also be registered with the ORB (e.g., **POA::activate_object()** before they can be passed by reference. Not registering the value as a CORBA object and/or not associating an appropriate policy with it results in an exception when trying to use it as a remote object, the “normal” behavior. The exception raised shall be OBJECT_NOT_EXIST with standard minor code 1.

9.2.8.1 Creation and Factories

When an instance of a value type is received by the ORB, it must be unmarshaled and an appropriate factory for its actual type found in order for the new instance to be created. The type is encoded by the RepositoryID, which is passed over the wire as part of an invocation. The mapping between the type (as specified by the RepositoryID) and the factory is language specific. In certain languages it may be possible to specify default policies that are used to find the factory, without requiring that specific routines be called. In others the runtime and/or generated code may have to explicitly specify the mapping on a per type basis. In others a combination may be used. In any event the ORB implementation is responsible for maintaining this mapping See Language Specific Value Factory Requirements on page 163 for more details on the requirements for each language mapping. Value box types do not need or use factories.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.2.9 Security Considerations

The addition of value types has few impacts on the CORBA security model. In essence, the security implications in defining and using value types are similar to those involved with the use of IDL structs. Instances of value types are mapped to local, concrete programming language constructs. Except for providing the marshaling mechanisms, the ORB is not directly involved with accessing value type implementations. This specification is mostly about two things: how value types manifest themselves as concrete programming language constructs and how they are transmitted.

To see this consider how value types are actually used. The IDL definition of a value type in conjunction with a programming language mapping is used to generate the concrete programming language definitions for that type.

Let us consider its life cycle. In order to use it, the programmer uses the mechanisms in the programming language to instantiate an instance. This is instance is a local programming language construct. It is not “registered” with the ORB, object adapter, etc. The programmer may manipulate this programming construct just like any other programming language construct. So far there are no security implications. As long as no ORB-mediated invocations are made, the programmer may manipulate the construct. Note, this includes making “local,” non ORB-mediated calls to any locally implemented operations. Any assignments to the construct are the responsibility of the programmer and have no special security implications.

Things get interesting when the program attempts to pass one of these constructs through an orb-mediated invocation (i.e., calls a stub that uses it as a parameter type, or uses the DII). There are two cases to consider: 1) Value as Value and 2) Value as Object Reference.

9.2.9.1 Value as Value

The formal type of the parameter is a value. This case is no different from using any other kind of a value (long, string, struct) in a CORBA invocation, with respect to security. The value (data) is marshaled and delivered to the receiving context. On the receiving context, the knowledge of the type is used (at least implicitly) to find the factory to create the correct local programming language construct. The data is then unmarshaled to fill in the newly created construct. This is similar to using other values (longs, strings, structs) except that the knowledge of the factory is not “built-in” to the ORB’s skeleton/DSI engine.

9.2.9.2 Value as Object Reference

The formal type of the parameter is an interface type that is supported by a value. The program must have “registered” the value with an object adapter and is really using the returned object reference (see for the specific rules.) Thus this case “reduces” to a regular CORBA invocation, using a regular object reference. An IOR is passed to the receiving context. All the “normal” security considerations apply. From the point of view of the receiving context, the IOR is a “normal” object reference. No “special” rules, with respect to security or otherwise, apply to it. The fact that it is ultimately a reference to an implementation that was created from instantiating and registering a value type implementation is not relevant.

In both of these cases, security considerations are involved with the decision to allow the ORB-mediated invocation to proceed. The fact that a value type is involved is not material.

9.3 Standard Value Box Definitions

For some CORBA-defined types for which preservation of sharing and transmission of nulls are likely to be important, the following value box type definitions are added to the CORBA module.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
module CORBA {
    valuetype StringValue string;
    valuetype WStringValue wstring;
};
```

9.4 Language Mappings

9.4.1 General Requirements

A concrete value is mapped to a concrete usable “class” construct in each programming language, plus possibly some helper classes where appropriate. In Java, C++, and Smalltalk this is a real concrete class. In C it is a struct.

An abstract value is mapped to some sort of an abstract construct--an interface in Java, and an abstract class with pure virtual function members in C++.

Tools that implement the language mapping are free to “extend” the implementation classes with “extra” data members and methods. When an instance of such a class is used as a parameter, only the portions that correspond directly to the IDL declaration, are marshaled and delivered to the receiving context. This allows freedom of implementations while preserving the notion of contract and type safety in IDL.

9.4.2 Language Specific Marshaling

Each language mapping defines an appropriate marshaling/unmarshaling API and the entry point for custom marshaling/unmarshaling.

9.4.3 Language Specific Value Factory Requirements

Each language mapping specifies the algorithm and means by which RepositoryIDs are used to find the appropriate factory for an instance of a value type so that it may be created as it is unmarshaled “off the wire.”

It is desirable, where it makes sense, to specify a “default” policy for automatically using RepositoryIDs that are in common formats to find the appropriate factory. Such a policy can be thought of as an implicit registration.

Each language mapping specifies how and when the registration occurs, both explicit and implicit. The registration must occur before an attempt is made to unmarshal an instance of a value type. If the ORB is unable to locate and use the appropriate factory, then a MARSHAL exception with standard minor code 1 is raised.

Because the type of the factory is programming language specific and each programming language platform has different policies, the factory type is specified as **native**. It is the responsibility of each language mapping to specify the actual programming language type of the factory.

```
module CORBA {

    // IDL
    native ValueFactory;
};
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.4.4 Value Method Implementation

The mapped class must support method bodies (i.e., code) that implement the required IDL operations. The means by which this association is accomplished is a language mapping “detail” in much the same way that an IDL compiler is.

9.5 Custom Marshaling

Value types can override the default marshaling/unmarshaling model and provide their own way to encode/decode their state. Custom marshaling is intended to be used to facilitate integration of existing “class libraries” and other legacy systems. It is explicitly not intended to be a standard practice, nor used in other OMG specifications to avoid “standard ORB” marshaling.

The fact that a value type has some custom marshaling code is declared explicitly in the IDL. This explicit declaration has two goals:

- ***Type safety*** - stubs and skeleton can know statically that a given type is custom marshaled and can then do a sanity check on what is coming over the wire.
- ***efficiency*** - for value types that are not custom marshaled no run time test is necessary in the marshaling code.

If a custom marshaled value type has a state definition, the state definition is treated the same as that of a non custom value type for mapping purposes (i.e., the fields show up in the same fashion in the concrete programming language). It is provided to help with application portability.

A custom marshaled value type is always a stateful value type.

```
// Example IDL

custom valuetype T {
    // optional state definition
        ...
};
```

Custom value types can never be safely truncated to base (i.e., they always require an exact match for their RepositoryId in the receiving context).

Once a value type has been marked as custom, it needs to provide an implementation that marshals and unmarshals the valuetype. The marshaling code encapsulates the application code that can marshal and unmarshal instances of the value type over a stream using the CDR encoding. It is the responsibility of the implementation to marshal the state of all of its base types.

The following sub clauses define the operations and streams that are used for custom marshaling.

9.5.1 Implementation of Custom Marshaling

Once a value type has been marked as custom, an implementation of the custom marshaling code must be provided. This is specified by providing a concrete implementation of an abstract value type, **CustomMarshal**, as part of the implementation of the value type. **CustomMarshal** encapsulates the application code that can marshal and unmarshal instances of the value type over a stream using the CDR encoding.

The following IDL defines the interfaces that are used to support the definition and use of custom marshaling.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
module CORBA {
    abstract valuetype CustomMarshal {
        void marshal (in DataOutputStream os);
        void unmarshal (in DataInputStream is);
    };
};
```

CustomMarshal is an abstract value type that is meant to be used by the ORB, not the user. Semantically it is treated as a custom valuetype's implicit base class, although the custom valuetype does not actually inherit it in IDL. The implementor of a custom value type provides an implementation of the **CustomMarshal** operations. The manner in which this is done is specified for each language mapping. Each custom marshaled value type has its own implementation. The interface is exposed in the CORBA module so that the implementor can use the skeletons generated by the IDL compiler as the basis for the implementation. Hence there is no need for the application to acquire a reference to a Stream.

Note that while nothing prevents a user from writing IDL that inherits from **CustomMarshal**, doing so will not make the type custom, nor will it cause the ORB to treat it as custom.

The implementation requirements of the streaming mechanism require that the implementations must be local since local memory addresses (i.e., the marshal buffers) have to be manipulated.

9.5.2 Marshaling Streams

The streams used for marshaling are defined below. They are responsible for marshaling and demarshaling the data that makes up a custom value in CDR format.

```
module CORBA {

    typedef sequence<any> AnySeq;
    typedef sequence<boolean> BooleanSeq;
    typedef sequence<char> CharSeq;
    typedef sequence<wchar> WCharSeq;
    typedef sequence<octet> OctetSeq;
    typedef sequence<short> ShortSeq;
    typedef sequence<unsigned short> UShortSeq;
    typedef sequence<long> LongSeq;
    typedef sequence<unsigned long> ULongSeq;
    typedef sequence<long long> LongLongSeq;
    typedef sequence<unsigned long long> ULongLongSeq;
    typedef sequence<float> FloatSeq;
    typedef sequence<double> DoubleSeq;
    typedef sequence<long double> LongDoubleSeq;
    typedef sequence<string> StringSeq;
    typedef sequence<wstring> WStringSeq;

    exception BadFixedValue {
        unsigned long offset;
    };

    abstract valuetype DataOutputStream {
        void write_any(in any value);
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
void write_boolean(in boolean value);
void write_char(in char value);
void write_wchar(in wchar value);
void write_octet(in octet value);
void write_short(in short value);
void write_ushort(in unsigned short value);
void write_long(in long value);
void write_ulong(in unsigned long value);
void write_longlong(in long long value);
void write_ulonglong(in unsigned long long value);
void write_float(in float value);
void write_double(in double value);
void write_longdouble(in long double value);
void write_string(in string value);
void write_wstring(in wstring value);
void write_Object(in Object value);
void write_Abstract(in AbstractBase value);
void write_Value(in ValueBase value);
void write_TypeCode(in TypeCode value);

void write_any_array(
    in AnySeq seq,
    in unsigned long offset,
    in unsigned long length
);
void write_boolean_array(
    in BooleanSeq seq,
    in unsigned long offset,
    in unsigned long length
);
void write_char_array(
    in CharSeq seq,
    in unsigned long offset,
    in unsigned long length
);
void write_wchar_array(
    in WCharSeq seq,
    in unsigned long offset,
    in unsigned long length
);
void write_octet_array(
    in OctetSeq seq,
    in unsigned long offset,
    in unsigned long length
);
void write_short_array(
    in ShortSeq seq,
    in unsigned long offset,
    in unsigned long length
);
void write_ushort_array(
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        in UShortSeq seq,
        in unsigned long offset,
        in unsigned long length
    );
    void write_long_array(
        in LongSeq seq,
        in unsigned long offset,
        in unsigned long length
    );
    void write_ulong_array(
        in ULongSeq seq,
        in unsigned long offset,
        in unsigned long length
    );
    void write_ulonglong_array(
        in ULongLongSeq seq,
        in unsigned long offset,
        in unsigned long length
    );
    void write_longlong_array(
        in LongLongSeq seq,
        in unsigned long offset,
        in unsigned long length
    );
    void write_float_array(
        in FloatSeq seq,
        in unsigned long offset,
        in unsigned long length
    );
    void write_double_array(
        in DoubleSeq seq,
        in unsigned long offset,
        in unsigned long length
    );

    void write_long_double_array(
        in LongDoubleSeq seq,
        in unsigned long offset,
        in unsigned long length
    );

    void write_fixed(
        in any fixed_value
    ) raises (BadFixedValue);
    void write_fixed_array(
        in AnySeq seq,
        in unsigned long offset,
        in unsigned long length
    ) raises (BadFixedValue);
};
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
abstract valuetype DataInputStream {
    any read_any();
    boolean read_boolean();
    char read_char();
    wchar read_wchar();
    octet read_octet();
    short read_short();
    unsigned short read_ushort();
    long read_long();
    unsigned long read_ulong();
    long long read_longlong();
    unsigned long long read_ulonglong();
    float read_float();
    double read_double();
    long double read_longdouble();
    string read_string();
    wstring read_wstring();
    Object read_Object();
    AbstractBase read_Abstract();
    ValueBase read_Value();
    TypeCode read_TypeCode();

    void read_any_array(
        inout AnySeq seq,
        in unsigned long offset,
        in unsigned long length
    );
    void read_boolean_array(
        inout BooleanSeq seq,
        in unsigned long offset,
        in unsigned long length
    );
    void read_char_array(
        inout CharSeq seq,
        in unsigned long offset,
        in unsigned long length
    );
    void read_wchar_array(
        inout WCharSeq seq,
        in unsigned long offset,
        in unsigned long length
    );
    void read_octet_array(
        inout OctetSeq seq,
        in unsigned long offset,
        in unsigned long length
    );
    void read_short_array(
        inout ShortSeq seq,
        in unsigned long offset,
        in unsigned long length
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
);
void read_ushort_array(
      inout UShortSeq seq,
      in unsigned long offset,
      in unsigned long length
);
void read_long_array(
      inout LongSeq seq,
      in unsigned long offset,
      in unsigned long length
);
void read_ulong_array(
      inout ULongSeq seq,
      in unsigned long offset,
      in unsigned long length
);
void read_ulonglong_array(
      inout ULongLongSeq seq,
      in unsigned long offset,
      in unsigned long length
);
void read_longlong_array(
      inout LongLongSeq seq,
      in unsigned long offset,
      in unsigned long length
);
void read_float_array(
      inout FloatSeq seq,
      in unsigned long offset,
      in unsigned long length
);
void read_double_array(
      inout DoubleSeq seq,
      in unsigned long offset,
      in unsigned long length
);

void read_long_double_array(
      inout DoubleSeq seq,
      in unsigned long offset,
      in unsigned long length
);
any read_fixed(
      in unsigned short digits,
      in short scale
) raises (BadFixedValue);
void read_fixed_array(
      inout AnySeq seq,
      in unsigned long offset,
      in unsigned long length,
      in unsigned short digits,
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            in short scale
        ) raises (BadFixedValue);
    };
};
```

Note that the Data streams are abstract value types. This ensures that their implementation will be local, which is required in order for them to properly flatten and encode nested value types.

The **read**_ operations that have an inout parameter named seq are expected to extend the sequence to fit the read value.

The ORB (i.e., the CDR encoding engine) is responsible for actually constructing the value's encoding. The application marshaling code merely calls the above operations. The details of writing the value tag, header information, end tag(s) are specifically not exposed to the application code. In particular the size of the custom data is not written by the application. This guarantees that the custom marshaling (and unmarshaling code) cannot corrupt the other parameters of the call.

If an inconsistency is detected, then the standard system exception MARSHAL is raised.

A possible implementation might have the engine determine that a custom marshal parameter is "next." It would then write the value tag and other header information and then return control back to the application defined marshaling policy, which would do the marshaling by calling the **DataOutputStream** operations to write the data as appropriate. (Note the stream takes care of breaking the data into chunks, if necessary.) When control was returned back to the engine, it performs any other cleanup activities to complete the value type, and then proceeds onto the next parameter. How this is actually accomplished is an implementation detail of the ORB.

The Data Streams shall test for possible shared or null values and place appropriate indirections or null encodings (even when used from the custom streaming policy).

There are no explicit operations for creating the streams. It is assumed that the ORB implicitly acts as a factory. In a sense they are always available.

For **write_fixed**, the **fixed_value** parameter must be an "any" containing a fixed value. If the "any" passed in does not contain a fixed value, then a BadFixedValue exception is raised with the offset field set to 0.

For **write_fixed_array**, the elements of the **seq** parameter that are specified by the offset and length parameters must be a sequence of "any"s each of which contains a fixed value. If any of these "any"s do not contain a fixed value, or if any of them contain a fixed value whose **digits** and **scale** (as specified by the **TypeCode** in the "any") differ from those of the first of these "any"s (as specified by its **TypeCode**), then a BadFixedValue exception is raised with the offset field set to a zero-origin ordinal number indicating the position of the first incorrect "any" within the subsequence of fixed values written to the stream.

For both **write_fixed** and **write_fixed_array**, the **TypeCode** within each "any" being written specifies the **digits** and **scale** to be used to write the fixed value contained in the "any." The **TypeCode** itself is not written to the **DataOutputStream**.

The **read_fixed** operation returns an "any" containing the fixed value that was read from the **DataInputStream**. The digits and scale in the **TypeCode** of the returned "any" are set to the **digits** and **scale** parameters passed to **read_fixed**. If the fixed value read from the **DataInputStream** is incompatible with the **digits** and **scale** parameters passed to **read_fixed**, then a BadFixedValue exception is raised with the offset field set to 0.

The **read_fixed_array** operation sets the elements of the **seq** parameter that are specified by the **offset** and **length** parameters. These elements are set to "any"s with **TypeCodes** specifying a fixed value whose **digits** and **scale** are the same as the **digits** and **scale** parameters, and fixed values that were read from the **DataInputStream**. The previous contents of these "any"s, including their **TypeCodes**, are destroyed by the **read_fixed_array** operation. Other "any"s in

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

the **seq** parameter (if any) are left unchanged. No **TypeCode** information is read from the **DataInputStream**. If any of the fixed values read from the **DataInputStream** are incompatible with the **digits** and **scale** parameters, then a BadFixedValue exception is raised with the **offset** field set to a zero-origin ordinal number indicating the position of the first incorrect “any” within the subsequence of fixed values read from the stream.

The stream representation of a fixed value is considered incompatible if its **digit** and **scale** values do not match the **digits** and **scale** values being used to read it from the stream.

9.6 Access to the Sending Context Run Time

There are two cases where a receiving context might want to access the run time environment of the sending context:

- To attempt the downloading of some missing implementation for the value.
- To access some meta information about the version of the value just received.

In order to provide that kind of service a call back object interface is defined. It may optionally be supported by the sending context (it can be seen as a service). If such a callback object is supported, its IOR may be added to an optional service context in the GIOP header passed from the sending context to the receiving context.

A service context tagged with the ServiceID **SendingContextRunTime** (see *CORBA, Part 2* - ORB Interoperability Architecture) contains an encapsulation of the IOR for a **SendingContext::RunTime** object. Because ORBs are always free to skip a service context they don’t understand, this addition does not impact IIOP interoperability.

```
module SendingContext {
    interface RunTime {}; // so that we can provide more
                          // sending context run time
                          // services in the future

    interface CodeBase: RunTime {
        typedef string URL; // blank-separated list of one or more URLs
        typedef sequence<URL> URLSeq;
        typedef sequence
            <CORBA::ValueDef::FullValueDescription> ValueDescSeq;

        // Operation to obtain the IR from the sending context
        CORBA::Repository get_ir();

        // Operations to obtain a location of the implementation code
        URL implementation(in CORBA::RepositoryId x);
        URLSeq implementations(in CORBA::RepositoryIdSeq x);

        // Operations to obtain complete meta information about a Value
        // This is just a performance optimization the IR can provide
        // the same information
        CORBA::FullValueDescription meta(in CORBA::RepositoryId x);
        ValueDescSeq metas(in CORBA::RepositoryIdSeq x);

        // To obtain a type graph for a value type
        // same comment as before the IR can provide similar
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
        // information
        CORBA::RepositoryIdSeq bases(in CORBA::RepositoryId x);
    };
};
```

Supporting the **CodeBase** interface for a given ORB run time is an issue of quality of service. The point here is that if the sending context does not support a **CodeBase,** then the receiving context will simply raise an exception with which the sending context had to be prepared to deal. There will always be cases where a receiving context will get a value type and won't be able to interpret it because:

- It can't get a legal implementation for it (even if it knows where it is, possibly due to security and/or resource access issues).
- Its local version is so radically different that it cannot make sense out of the piece of state being provided.

These two failure modes will be represented by the CORBA system exception NO_IMPLEMENT with identified minor codes, for a missing local value implementation and for incompatible versions (see Standard Minor Exception Codes on page 156).

Under certain conditions it is possible that when several values of the same CORBA type (same repository id) are sent in either a request or reply, that the reality is that they have distinct implementations. In this case, in addition to the codebase URL(s) sent in the service context, each value that has a different codebase may have codebase URL(s) associated with it. This is encoded by using a different tag to encode the value on the wire.

The sending context does not need to resend the same value for this service context on subsequent requests over the same underlying connection. Resending a different value for this service context is only necessary if the callback object reference in use is changed by the sending context within the lifetime of the underlying connection.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

10 Abstract Interface Semantics

10.1 Overview

In many cases it may be useful to defer the determination of whether an object is passed by reference or by value until runtime. An IDL abstract interface provides this capability. See Example on page 174 for an example of when this might be useful.

10.2 Semantics of Abstract Interfaces

Abstract interfaces differ from regular IDL interfaces in the following ways:

1. When used in an operation signature, they do not determine whether actual parameters are passed as an object reference or by value. Instead, the type of the actual parameter (regular interface or value) is used to make this determination using the following rules:
   - The actual parameter is passed as an object reference if it is a regular interface type (or a subtype of a regular interface type), and that regular interface type is a subtype of the signature abstract interface type, and the object is already registered with the ORB/OA.
   - The actual parameter is passed as a value if it cannot be passed as an object reference but can be passed as a value. Otherwise, a BAD_PARAM exception is raised.
2. Abstract interfaces do not implicitly inherit from **CORBA::Object.** This is because they can represent either value types or CORBA object references, and value types do not necessarily support the object reference operations (see Object Reference Operations on page 105). If an IDL abstract interface type can be successfully narrowed to an object reference type (a regular IDL interface), then the **CORBA::Object** operations can be invoked on the narrowed object reference.
3. Abstract interfaces implicitly inherit from **CORBA::AbstractBase**. This type is defined as native. It is the responsibility of each language mapping to specify the actual programming language type that is used for this type.

```
module CORBA {
// IDL
    native AbstractBase;
};
```

4. Abstract interfaces do not imply copy semantics for value types passed as arguments to their operations. This is because their operations may be either CORBA invocations (for abstract interfaces that represent CORBA object references) or local programming language calls (for abstract interfaces that represent CORBA value types). See Operations on page 158 and Parameter Passing on page 159 for details of these differences.
5. Special inheritance rules that apply to abstract interfaces are described in Abstract Interface on page 49.
6. See the General Inter-orb Protocol clause in CORBA, Part 2 - for special consideration when transmitting an abstract interface using GIOP.

In other respects, abstract interfaces are identical to regular IDL interfaces. For example, consider the following operation **m1()** in abstract interface **foo**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
abstract interface foo {
        void m1(in AnInterfaceType x, in AnAbstractInterfaceType y,
            in AValueType z);
};
```

x's are always passed by reference.

z's are passed as:

- copied values if **foo** refers to an ordinary interface.
- non-copied values if **foo** refers to a value type.

y's are passed as:

- reference if their concrete type is an ordinary interface subtype of **AnAbstractInterfaceType** (registered with the ORB), no matter what **foo**'s concrete type is.
- copied values if their concrete type is value and **foo**'s concrete type is ordinary interface.
- non-copied values if their concrete type is value and **foo**'s concrete type is value.

10.3 Usage Guidelines

Abstract interfaces are intended for situations where it cannot be known at compile time whether an object reference or a value will be passed. In other cases, a regular interface or value type should be used. Abstract interfaces are not intended to replace regular CORBA interfaces in situations where there is no clear need to provide runtime flexibility to pass either an object reference or a value. If reference semantics are intended, regular interfaces should be used.

10.4 Example

For example, in a business application it is extremely common to need to display a list of objects of a given type, with some identifying attribute like account number and a translated text description such as "Savings Account." A developer might define an interface such as **Describable** whose methods provide this information, and implement this interface on a wide range of types. This allows the method that displays items to take an argument of type **Describable** and query it for the necessary information. The **Describable** objects passed in to the **display** method may be either CORBA interface types (passed in as object references) or CORBA value types (passed in by value).

In this example, **Describable** is used as a polymorphic abstract type. No instances of type **Describable** exist, but many different instances have interfaces that support the **Describable** type abstraction. In C++, **Describable** would be an abstract base class; in Java, an interface. In statically typed languages, the compiler can check that the actual parameter type passed by callers of **display** is a valid subtype of **Describable** and therefore supports the methods defined by **Describable**. The **display** method can simply invoke the methods of **Describable** on the objects that it receives, without concern for any details of their implementation.

Describable could not be declared as a regular IDL interface. This is because arguments of declared interface type are always passed as object references (see Parameter Passing on page 159) and we also want the **display** method to be able to accept value type objects that can only be passed by value. Similarly we cannot define **Describable** as a value type because then the **display** method would not be able to accept actual parameter objects that only support passing as an object reference. Abstract interfaces are needed to cover such cases.

The **Describable** abstract interface could be defined and used by the following IDL:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
abstract interface Describable {
        string get_description();
};

interface Example {
        void display (in Describable anObject);
};

interface Account : Describable {// passed by reference
   // add Account methods here
};

valuetype Currency supports Describable {// passed by value
   // add Currency methods here
};
```

If **Describable** was defined as a regular interface instead of an abstract interface, then it would not be possible to pass a **Currency** value to the display method, even though the **Currency** IDL type supports the **Describable** interface.

10.5 Security Considerations

Security considerations for abstract interfaces are similar to those for regular interfaces and values (see Security Considerations on page 162). This is because an abstract interface formal parameter type allows either a regular interface (IOR) or a value to be passed. Likewise, an operation defined in an abstract interface can be implemented by either a regular interface (with “normal” security considerations) or by a value type (in which case it is a local call, not mediated by the ORB). The security implication of making the choice between these alternatives a runtime determination is that the programmer must ensure that for both alternatives, no security violations can occur. For example, a technique similar to that described in “Passing Values to Trusted Domains” could be used to avoid inadvertently passing values outside a domain of trust.

10.5.1 Passing Values to Trusted Domains

When a server passes an object reference, it can be sure that access control policies will apply to any attempt to access anything through that object reference. When the underlying object is passed as a value, the granularity and level/ semantics of access control are different. In the “by value” case, all the data for the object is passed, and method invocations on the passed object are local calls that are not mediated by the ORB. Whether the server wants to use the (potentially more permissive) pass by value access control or not could depend on the security domain, which is receiving the said object or object reference.

Consider the case where the server S has an object O that it is willing to pass only in the form of an object reference Or' to a domain Du that it does not trust, but is willing to pass the object by value Ow to another domain Ot that it trusts.

This flexibility is not possible without abstract interfaces. Signatures would have to be written to either always pass references or always pass values, irrespective of the level of trust of the invocation target domain. However, abstract interfaces provide the necessary flexibility. The formal parameter type **MyType** can be declared as an abstract interface and the method invocation can be coded along the lines of

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
myExample->foo(security_check(myExample,mydata));
```

where the **security_check** function determines the level of trust of **myExample**'s domain and returns a regular interface subtype of **MyType** for untrusted domains and a value subtype of **MyType** for trusted domains. The rules for abstract interfaces will then pass the correct thing in both these cases.

© ISO/IEC 2008 - All rights reserved
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11 Dynamic Invocation Interface

The Dynamic Invocation Interface (DII) describes the client's side of the interface that allows dynamic creation and invocation of request to objects. All types defined in this clause are part of the CORBA module.

11.1 Overview

The Dynamic Invocation Interface (DII) allows dynamic creation and invocation of requests to objects. A client using this interface to send a request to an object obtains the same semantics as a client using the operation stub generated from the type specification.

A request consists of an object reference, an operation, and a list of parameters. The ORB applies the implementation-hiding (encapsulation) principle to requests.

In the Dynamic Invocation Interface, parameters in a request are supplied as elements of a list. Each element is an instance of a **NamedValue** (see Common Data Structures on page 177). Each parameter is passed in its native data form.

Parameters supplied to a request may be subject to run-time type checking upon request invocation. Parameters must be supplied in the same order as the parameters defined for the operation in the Interface Repository.

The standard user exception WrongTransaction is defined in the CORBA module, prior to the definitions of the ORB and Request interfaces, as follows:

```
exception WrongTransaction {};
```

This exception can be raised only if the request is implicitly associated with a transaction (the current transaction at the time that the request was issued).

11.1.1 Common Data Structures

The type **NamedValue** is a well known data type in OMG IDL. It can be used either as a parameter type directly or as a mechanism for describing arguments to a request. The types are described in OMG IDL as:

```
module CORBA {

    typedef unsigned long Flags;
    struct NamedValue {                                    PIDL
        Identifier      name;       // argument name
        any             argument;   // argument
        long            len;        // length/count of argument value
        Flags           arg_modes;// argument mode flags
    };
};
```

For out parameters, applications can set the **argument** member of the **NamedValue** structure to a value that includes either a NULL or a non-NULL storage pointer. If a non-null storage pointer is provided for an out parameter, the ORB will attempt to use the storage pointed to for holding the value of the out parameter. If the storage pointed to is not sufficient to hold the value of the out parameter, the behavior is undefined.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A named value includes an argument name, argument value (as an **any**), length of the argument, and a set of argument mode flags. When named value structures are used to describe arguments to a request, the names are the argument identifiers specified in the OMG IDL definition for a specific operation.

As described in *CORBA* (Mapping: COM and CORBA) an **any** consists of a **TypeCode** and a pointer to the data value. The **TypeCode** is a well known opaque type that can encode a description of any type specifiable in OMG IDL. See this sub clause for a full description of **TypeCode**s.

For most data types, **len** is the actual number of bytes that the value occupies. For object references, **len** is 1. Table 11.1 shows the length of data values for the C language binding. The behavior of a **NamedValue** is undefined if the **len** value is inconsistent with the TypeCode.

Table 11.1 - C Language Binding Data Values

Data type: X	Length (X)
short	sizeof (CORBA_short)
unsigned short	sizeof (CORBA_unsigned_short)
long	sizeof (CORBA_long)
unsigned long	sizeof (CORBA_unsigned_long)
long long	sizeof (CORBA_long_long)
unsigned long long	sizeof (CORBA_unsigned_long_long)
float	sizeof (CORBA_float)
double	sizeof (CORBA_double)
long double	sizeof (CORBA_long_double)
fixed<d,s>	sizeof (CORBA_fixed_d_s)
char	sizeof (CORBA_char)
wchar	sizeof (CORBA_wchar)
boolean	sizeof (char)
octet	sizeof (CORBA_octet)
string	strlen (string) /* does NOT include '\0' byte! */
wstring	number of wide characters in string, not including wide null terminator
enum E {};	sizeof (CORBA_enum)
union U { };	sizeof (U)
struct S { };	sizeof (S)
Object	1
array N of type T1	Length (T1) * N
sequence V of type T2	Length (T2) * V /* V is the actual, dynamic, number of elements */

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

The **arg_mode** field is of type **Flags** which is an **unsigned long**. This field is used as follows in this structure. It should be noted that **Flags** type is used as parameter type in many operations and the meaning of the constants passed in those cases are specific to those operations. Those values should not be confused with the specific use of this type in the context of the **NamedValue** structure. These values are reserved, as are the high order 16 bits of the **unsigned long**.:

CORBA::ARG_IN	1	The associated value is an input only argument.
CORBA::ARG_OUT	2	The associated value is an output only argument.
CORBA::ARG_INOUT	3	The associated value is an in/out argument.

The specific usage of **Flags** in other contexts are described as part of the description of the operation that uses this type of parameters.

11.1.2 Memory Usage

The values for output argument data types that are unbounded strings or unbounded sequences are returned as pointers to dynamically allocated memory. In order to facilitate the freeing of all "out-arg memory," the request routines provide a mechanism for grouping, or keeping track of, this memory. If so specified, out-arg memory is associated with the argument list passed to the create request routine. When the list is deleted, the associated out-arg memory will automatically be freed.

If the programmer chooses not to associate out-arg memory with an argument list, the programmer is responsible for freeing each out parameter using **`CORBA_free()`**, which is discussed in the *C Language Mapping* specification (*Mapping for Structure Types* sub clause).

11.1.3 Return Status and Exceptions

In the Dynamic Invocation interface, routines typically indicate errors or exceptional conditions either via programming language exception mechanisms, or via an Environment parameter for those languages that do not support exceptions. Thus, the return type of these routines is void.

11.2 Request Operations

The request operations (except **create_request**) are defined in terms of the **Request** pseudo-object. The **Request** routines use the **`NVList`** definition defined in the preceding sub clause.

```
module CORBA {

    native OpaqueValue;

    interface Request {                                    // PIDL

        void add_arg (
            in Identifier     name,     // argument name
            in TypeCode       arg_type, // argument datatype
            in OpaqueValue    value,    // argument value to be added
            in long           len,      // length/count of argument value
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            in Flags            arg_flags   // argument flags
        );

        void invoke (
            in Flags            invoke_flags    // invocation flags
        );

        void delete ();

        void send (
            in Flags            invoke_flags    // invocation flags
        );

        void get_response () raises (WrongTransaction);

        boolean poll_response();

        Object sendp( );

        void prepare(in Object p);

        void sendc(in Object handler);
    };
};
```

In IDL, The **native** type **OpaqueValue** is used to identify the type of the implementation language representation of the value that is to be passed as a parameter. For example in the C language this is the C language type **`(void *)`**. Each language mapping specifies what **OpaqueValue** maps to in that specific language.

For each **Request** pseudo-object instance, only one call to either the invoke or the **send** operations is legal during the lifetime of the **Request** object. In addition, once a **Request** object was passed to one of the **send_multiple_requests_*** operations, neither invoke nor send can be called, nor can it be passed in another invocation of **send_multiple_request_*** operation.Violations raise BAD_INV_ORDER with standard minor code 5 or 10.

11.2.1 create_request

Because it creates a pseudo-object, this operation is defined in the **Object** interface (see Object Reference Operations on page 105 for the complete interface definition). The **create_request** operation is performed on the **Object** that is to be invoked.

```
module CORBA{

    interface Object{                                           // PIDL
        . . . . .

        void create_request (
            in Context          ctx,            // context object for operation
            in Identifier       operation,      // intended operation on object
            in NVList           arg_list,       // args to operation
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            inout NamedValue result,      // operation result
            out Request      request,     // newly created request
            in Flags         req_flags    // request flags
        );
    };
};
```

This operation creates an ORB request. The actual invocation occurs by calling **invoke** or by using the **send / get_response** calls.

The operation name specified on **create_request** is the same operation identifier that is specified in the OMG IDL definition for this operation. In the case of attributes, it is the name as constructed following the rules specified in the **ServerRequest** interface as described in the DSI in ServerRequestPseudo-Object on page 194.

The **arg_list**, if specified, contains a list of arguments (input, output, and/or input/output) that become associated with the request. If **arg_list** is omitted (specified as **NULL**), the arguments (if any) must be specified using the **add_arg** call below.

Arguments may be associated with a request by passing in an argument list or by using repetitive calls to **add_arg**. One mechanism or the other may be used for supplying arguments to a given request; a mixture of the two approaches is not supported.

If specified, the **arg_list** becomes associated with the request; until the **invoke** call has completed (or the request has been deleted), the ORB assumes that **arg_list** (and any values it points to) remains unchanged.

When specifying an argument list, the **value** and **len** for each argument must be specified. An argument's datatype, name, and usage flags (i.e., in, out, inout) may also be specified; if so indicated, arguments are validated for data type, order, name, and usage correctness against the set of arguments expected for the indicated operation.

An implementation of the request services may relax the order constraint (and allow arguments to be specified out of order) by doing ordering based upon argument name.

The context properties associated with the operation are passed to the object implementation. The object implementation may not modify the context information passed to it.

The operation result is placed in the **result** argument after the invocation completes.

The **req_flags** argument is defined as a bitmask (**long**) that may contain the following flag values:

CORBA::OUT_LIST_MEMORY indicates that any out-arg memory is associated with the argument list (**NVList**).

Setting the **OUT_LIST_MEMORY** flag controls the memory allocation mechanism for out-arg memory (output arguments, for which memory is dynamically allocated). If **OUT_LIST_MEMORY** is specified, an argument list must also have been specified on the **create_request** call. When output arguments of this type are allocated, they are associated with the list structure. When the list structure is freed (see below), any associated out-arg memory is also freed.

If **OUT_LIST_MEMORY** is *not* specified, then each piece of out-arg memory remains available until the programmer explicitly frees it with procedures provided by the language mappings (see the *C Language Mapping* specification, *Argument Passing Considerations* sub clause; *C++ Language Mapping* specification, *NVList* sub clause; and the *COBOL Language Mapping* specification, *Argument Passing Considerations* sub clause).

The implicit object reference operations **non_existent**, **is_a, repository_id and get_interface** may be invoked using DII. No other implicit object reference operations may be invoked via DII.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

To create a request for any one of these allowed implicit object reference operations, **create_request** must be passed the name of the operation with a "_" prepended, in the parameter "operation." For example to create a DII request for "**is_a**", the name passed to **create_request** must be "**_is_a**." If the name of an implicit operation that is not invocable through DII is passed to **create_request** with a "_" prepended, **create_request** shall raise a BAD_PARAM standard system exception with the standard minor code 32. For example, if "**_is_equivalent**" is passed to **create_request** as the "**operation**" parameter will cause **create_request** to raise the BAD_PARAM standard system exception with the standard minor code 32.

11.2.2 add_arg

```
void add_arg (                                                    // PIDL
    in Identifier       name,       // argument name
    in TypeCode         arg_type,   // argument datatype
    in OpaqueValue      value,      // argument value to be added
    in long             len,        // length/count of argument value
    in Flags            arg_flags   // argument flags
);
```

add_arg incrementally adds arguments to the request.

For each argument, minimally its **value** and **len** must be specified. **len** is the length in octets, of the thing that the **value** parameter refers to. An argument's data type, name, and usage flags (i.e., in, out, inout) may also be specified. If so indicated, arguments are validated for data type, order, name, and usage correctness against the set of arguments expected for the indicated operation.

An implementation of the request services may relax the order constraint (and allow arguments to be specified out of order) by doing ordering based upon argument name.

The arguments added to the request become associated with the request and are assumed to be unchanged until the invoke has completed (or the request has been deleted).

Arguments may be associated with a request by specifying them on the **Object::create_request** call or by adding them via calls to **add_arg**. Using both methods for specifying arguments for the same request is not supported.

In addition to the argument modes defined in Common Data Structures on page 177, **arg_flags** may also take the flag value **IN_COPY_VALUE**. The argument passing flags defined in "Common Data Structures" may be used here to indicate the intended parameter passing mode of an argument.

If the **IN_COPY_VALUE** flag is set, a copy of the argument value is made and used instead. This flag is ignored for inout and out arguments.

11.2.3 invoke

```
void invoke (                                                     // PIDL
    in Flags            invoke_flags        // invocation flags
);
```

This operation calls the ORB, which performs method resolution and invokes an appropriate method. If the method returns successfully, its result is placed in the **result** argument specified on **create_request**. Calling **invoke** on a **Request** after **invoke**, **send**, or **ORB::send_multiple_requests** for that **Request** was called raises BAD_INV_ORDER with standard minor code 5 or 10.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.2.4 delete

```
void delete ( );                                    // PIDL
```

This operation deletes the request. Any memory associated with the request (i.e., by using the **IN_COPY_VALUE** flag) is also freed.

11.2.5 send

```
void send (                                         // PIDL
    in Flags        invoke_flags        // invocation flags
);
```

Send initiates an operation according to the information in the **Request**. Unlike **invoke, send** returns control to the caller without waiting for the operation to finish. To determine when the operation is done, the caller must use the **get_response** or **ORB::get_next_response** operations described below. The out parameters and return value must not be used until the operation is done.

Although it is possible for some standard system exceptions to be raised by the **send** operation, there is no guarantee that all possible errors will be detected. For example, if the object reference is not valid, **send** might detect it and raise an exception, or might return before the object reference is validated, in which case the exception will be raised when **get_response** is called.

If the operation is defined to be oneway or if **INV_NO_RESPONSE** is specified, and the effective **SyncScopePolicy** does not have a value of **WITH_SERVER** or **WITH_TARGET**, then **get_response** does not need to be called. In such cases, some errors might go unreported, since if they are not detected before **send** returns there is no way to inform the caller of the error.

The following invocation flags are currently defined for **send**:

CORBA::INV_NO_RESPONSE indicates that the invoker wishes the request to be subject to the effective **SyncScopePolicy**. If the **SyncScopePolicy** has a value of **NONE** or **WITH_TRANSPORT**, the invoker will not receive a response, nor does it expect any of the output arguments (in/out and out) to be updated. This option may be specified even if the operation has not been defined to be **oneway**.

11.2.6 poll_response

```
// PIDL
boolean poll_response ();
```

poll_response determines whether the request has completed. A **TRUE** return indicates that it has; **FALSE** indicates it has not.

Return is immediate, whether the response has completed or not. Values in the request are not changed.

11.2.7 get_response

```
//PIDL
void get_response () raises (WrongTransaction);
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

get_response returns the result of a request. If **get_response** is called before the request has completed, it blocks until the request has completed. Upon return, the out parameters and return values defined in the **Request** are set appropriately and they may be treated as if the **Request** invoke operation had been used to perform the request.

A request has an associated transaction context if the thread originating the request had a non-null transaction context and the target object is a transactional object. The **get_response** operation may raise the WrongTransaction exception if the request has an associated transaction context, and the thread invoking **get_response** either has a null transaction context or a non-null transaction context that differs from that of the request. If a BAD_INV_ORDER exception with standard minor code X3599 is received, it shall be trapped and a WrongTransaction shall be returned to the caller.

11.2.8 sendp

sendp initiates an operation according to the information in the Request and returns a reference to a **MessageRouting::PersistentRequest** as a **CORBA::Object**. As with **send**, the results of invocations made with **sendp** will be available once the caller uses **get_response** or **get_next_response**. The out parameters and return value must not be used before the operation is done. A new **CORBA::Request** may be constructed (in this same or a different process) and used to poll for the response to this request by calling **create_request**, properly associating the out arguments and return value with that request and then passing the **PersistentRequest** reference to the new Request's **prepare** (described below). The caller can then invoke **get_response** or **get_next_response** to obtain the operation results.

As with **send**, **sendc** may raise a standard system exception if a failure is detected before control is returned to the client, but this is not guaranteed. All other exceptions will be raised when **get_response** is called.

11.2.9 prepare

prepare is called to associate an initialized **CORBA::Request** with a previous operation that was initiated via **sendp**. The Request must be created and associated with the operation's out arguments and return value prior to calling **prepare**. Once **prepare** has been called, it is as if that prepared Request was the one that actually had **sendp** used. Each Request is subject only to one of these operations, which puts it in a valid state for an invocation of **get_response**: **send**, **sendp**, **sendc**, or **prepare**. Invoking **prepare** on a Request that had previously been used for a send (or one of its variants) raises the standard system exception BAD_INV_ORDER. Invoking prepare with an object reference that was not previously returned from an invocation of **sendp** raises the standard system exception BAD_PARAM.

11.2.10 sendc

sendc initiates an operation according to the information in the Request. Unlike **send**, the results of invocations made with **sendc** will be available through the callback **Messaging::ReplyHandler** passed into **sendc** as a base **CORBA::Object**. For an invocation of operation "foo," the "foo" or "foo_excep" methods of the ReplyHandler is invoked to receive the reply. See Type-Specific ReplyHandler Mapping on page 434 for details of how the names of the operations to be invoked to return the reply are constructed, as well as the form of the argument lists for the reply invocations. A truly dynamic client can implement this **ReplyHandler** using the DSI. Specifying a nil **ReplyHandler** is equivalent to invoking **send** with a flag of **CORBA::INV_NO_RESPONSE**.

As with **send**, **sendc** may raise a standard system exception if a failure is detected before control is returned to the client, but this is not guaranteed. All other exceptions will be passed to the **ReplyHandler**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.3 ORB Operations

11.3.1 send_multiple_requests

```
module CORBA {

    interface Request;       // forward declaration
    typedef sequence <Request> RequestSeq;

    interface ORB {
        . . . . . .

        void send_multiple_requests_oneway(
            in RequestSeq req
        );

        void send_multiple_requests_deferred(
            in RequestSeq req
        );
    };
};
```

send_multiple_requests initiates more than one request in parallel. Like **send, send_multiple_requests** returns to the caller without waiting for the operations to finish. To determine when each operation is done, the caller must use the **Request::get_response** or **get_next_response** operations.

Calling **send** on a request after **invoke**, **send**, or **send_multiple_requests** for that request was called raises BAD_INV_ORDER with standard minor code 10.

Calling **send_multiple_requests** for a request after **invoke**, **send**, or **send_multiple_requests** for that request was called raises BAD_INV_ORDER with standard minor code 10. If **send_multiple_requests** raises BAD_INV_ORDER, the actual number of requests that were sent is implementation dependent.

11.3.2 get_next_response and poll_next_response

```
module CORBA {

    interface Request;       // forward declaration
    typedef sequence <Request> RequestSeq;

    interface ORB {
        . . . . . .

        boolean poll_next_response();

        void get_next_response(
            out Request req
        ) raises (WrongTransaction);
    };
};
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Poll_next_response determines whether any request has completed. A **TRUE** return indicates that at least one has; **FALSE** indicates that none have completed. Return is immediate, whether any response has completed or not.

Get_next_response returns the next request that completes. Despite the name, there is no guaranteed ordering among the completed requests, so the order in which they are returned from successive **get_next_response** calls is not necessarily related to the order in which they finish.

A request has an associated transaction context if the thread originating the request had a non-null transaction context and the target object is a transactional object. The **get_next_response** operation may raise the WrongTransaction exception if the request has an associated transaction context, and the thread invoking **get_next_response** has a non-null transaction context that differs from that of the request. If a BAD_INV_ORDER exception with standard minor code X3599 is received, it shall be trapped and a WrongTransaction shall be returned to the caller.

Calling **poll_response** before send or **send_multiple_requests** for that request raises BAD_INV_ORDER with standard minor code 11. Calling **poll_response** after calling **invoke** raises BAD_INV_ORDER with standard minor code 13. Calling **poll_response** after calling **get_response** raises BAD_INV_ORDER with standard minor code 12. Calling **poll_response** after that request was returned by **get_next_response** raises BAD_INV_ORDER with standard minor code 12.

Calling **get_next_response** or **poll_next_response** at a time when no requests are outstanding raises BAD_INV_ORDER with standard minor code 11. If concurrent calls to **get_next_response** or **poll_next_response** are in progress, the exact outcome is implementation dependent; however, **get_next_response** is guaranteed not to return the same completed request to more than one caller.

11.4 Polling

There are two types of Polling model invocations that allow a client to proceed before the request finishes: The DII's **send** (which supports deferred synchronous invocations) and the typed **sendp** variants of the interface stubs (which support both deferred synchronous and asynchronous invocations). This sub clause describes a single mechanism that allows a client to query or block on the completion of outstanding requests.

- For the typed polling model (**sendp**), a client invokes the request's type-specific **Poller** to receive the response. This poll can either block (wait for the completion) or return immediately if the request isn't finished yet, depending on the value of the first parameter. Alternately, a client can simply query whether the request has completed by using the generic non-blocking **CORBA::Pollable::is_ready( )** operation defined on the base interface that is inherited by all type-specific pollers. For the sake of efficiency, it must be possible to query or block on multiple async pollers in a single operation. To do this, it is necessary to identify precisely, which such pollers are to be polled.
- A client might want to mix deferred typed and dynamic operations. Deferred DII (in some unholy combination of language mappings) has operations somewhat similar to those of the typed **Poller: ORB::poll_next_response** and **ORB::get_next_response**. It should be possible to mix the two kinds of polling: typed and dynamic.
- Other potential happenings might occur that are susceptible to polling in current or future CORBA. This mechanism is designed for extensibility so that other ORB services can perform a poll as a part of the single poll operation described here.

The mechanism for generalized polling on multiple types of occurrences uses the **CORBA::PollableSet** interface.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
module CORBA {

    local interface PollableSet;

    abstract valuetype Pollable {
        boolean is_ready(
            in unsigned long timeout
        );

        PollableSet create_pollable_set( );
    };

    abstract valuetype DIIPollable : Pollable { };

    local interface PollableSet {

        exception NoPossiblePollable { };
        exception UnknownPollable { };

        DIIPollable create_dii_pollable();

        void add_pollable(
            in Pollable potential
        );

        Pollable get_ready_pollable(
            in unsigned long timeout
        ) raises( NoPossiblePollable );

        void remove(
            in Pollable potential
        ) raises( UnknownPollable );

        unsigned short number_left( );
    };
};
```

11.4.1 Abstract Valuetype Pollable

A **Pollable** supports queries to see if it is ready to be used, and can be registered with a pollable set to allow a single client thread to block on multiple potential happenings at the same time.

11.4.1.1 is_ready

```
boolean is_ready(
    in unsigned long timeout
);
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Returns the value **TRUE** if and only if the specific happening represented by the pollable is ready to be consumed. Returns the value **FALSE** if the pollable is not yet ready to be consumed. If the **timeout** argument is the maximum value for **unsigned long**, the operation will block until it can return the value **TRUE** indicating that its happening is ready to be consumed. If the **timeout** argument is the value 0, the operation returns immediately.

11.4.1.2 create_pollable_set

```
PollableSet create_pollable_set( );
```

Once there is a **Pollable**, it is possible to create a set of such pollables, which can be queried or upon which a client can block. The **create_pollable_set** operation creates a **PollableSet** object reference for an object with an empty set of pollable entities.

11.4.2 Abstract Valuetype DIIPollable

The specific **Pollable** that indicates interest in DII requests. A **DIIPollable** can be used in conjunction with a pollable set to allow a client to block or poll for the completion of DII requests, similar to the use of **CORBA::ORB::get_next_response**. When the **DIIPollable** is returned from **PollableSet::poll**, the reply to some DII request must be ready for processing.

11.4.3 interface PollableSet

The pollable set contains potential happenings for which a poll can be performed. The client adds potential happenings to the set and later queries the set to see if any have occurred. **PollableSet** is a locality constrained object.

NOTE: There is a factory for **PollableSet** on the generic **Pollable** interface. Some implementation of this interface, such as a type-specific poller value, must first be accessible before a client can create a **PollableSet**.

11.4.3.1 create_dii_pollable

```
DIIPollable create_dii_pollable();
```

Returns an instance of **DIIPollable** that can subsequently be registered to indicate interest in replies to DII requests.

11.4.3.2 add_pollable

```
void add_pollable(
    in Pollable potential
);
```

The **add_pollable** operation adds a potential happening to the **PollableSet**. The supplied **Pollable** parameter is some implementation that can be polled for readiness. To register interest in DII requests, an instance of **DIIPollable** is added to the pollable set.

If the supplied **Pollable** has already been added to another **PollableSet**, this operation raises the standard **BAD_PARAM** system exception with minor code 43.

11.4.3.3 get_ready_pollable

```
Pollable get_ready_pollable(
    in unsigned long timeout
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

) raises(NoPossiblePollable);

The **get_ready_pollable** operation asks the **PollableSet** if any of its potential happenings have occurred. The **timeout** parameter indicates how many milliseconds this call should wait until the response becomes available. If this timeout expires before a reply is available, the operation raises the standard system exception TIMEOUT. Any delegated invocations used by the implementation of this polling operation are subject to the single **timeout** parameter, which supersedes any ORB or thread-level timeout quality of service. Two specific values are of interest:

- 0 - the call is a non-blocking query that raises the standard system exception NO_RESPONSE if the reply is not immediately available.
- 2^{32}-1 - the maximum value for **unsigned long** indicates no timeout should be used. The query will not return until the reply is available.

If the **PollableSet** contains no potential happenings, the NoPossiblePollable exception is raised. If an actual happening is returned, the **PollableSet** removes that happening from the set. For the typed **Poller**, removing the happening is necessary since its usefulness ends once the **Poller** completes. In the case of a DII happening, there may still be deferred requests outstanding; if this is the case, the client application must add the **DIIPollable** again to the **PollableSet**.

When the **get_ready_pollable** operation blocks, the ORB has control of the thread and can process any work it has (such as receiving and dispatching requests through its Object Adapter). The **get_ready_pollable** operation can be used in an “event-style main loop” using **ORB::work_pending** and **ORB::perform_work**.

If the ORB supports multiple threads, one thread may be blocking on a **PollableSet** while another is adding and removing potential happenings from the set. It is valid for the **PollableSet** to change dynamically while a **poll** is in progress. If another thread’s **PollableSet::remove** operation leaves the **PollableSet** empty, any blocked threads raise the NoPossiblePollable exception.

11.4.3.4 remove

```
void remove(
    in Pollable potential
) raises( UnknownPollable );
```

The **remove** operation deletes the potential happening identified by the **potential** parameter from the **PollableSet**. If it was not a member of the set, the UnknownPollable exception is raised.

11.4.3.5 number_left

```
unsigned short number_left( );
```

The **number_left** operation returns the number of potential happenings in the pollable set. A returned value of zero means that there are no potential happenings in the set, in which case a query on the set would raise the NoPossibleHappening exception. A return value of 65535 indicates that there are at least 65535 remaining number of potential happenings.

11.5 List Operations

NVList is a pseudo-interface that facilitates manipulation of list of name value pairs. The operations that create **NVList** objects are defined in the **ORB** interface Clause, but are described in this sub clause. The **NVList** pseudo-interface is shown below.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
interface NVList {                                          // PIDL
    void add_item (
        in Identifier       item_name,    // name of item
        in TypeCode         item_type,    // item datatype
        in OpaqueValue      value,        // item value
        in long             value_len,    // length of item value
        in Flags            item_flags    // item flags
    );
    void free ( );
    void free_memory ( );
    void get_count (
        out long             count        // number of entries in the list
    );
};
```

Interface **NVList** is defined in the CORBA module.

11.5.1 create_list

This operation, which creates a pseudo-object, is defined in the ORB interface and excerpted below.

```
void create_list (                                                    //PIDL
    in long         count,      // number of items to allocate for list
    out NVList      new_list    // newly created list
);
```

This operation allocates a list and clears it for initial use. The specified count is a "hint" to help with the storage allocation. List items may be added to the list using the **add_item** routine. Items are added starting with the "**slot()**," in the next available slot.

An **NVList** is a partially opaque structure. It may only be allocated via a call to **create_list**.

11.5.2 add_item

```
void add_item (                                             // PIDL
    in Identifier       item_name,          // name of item
    in TypeCode         item_type,          // item datatype
    in OpaqueValue      value,              // item value
    in long             value_len,          // length of item value
    in Flags            item_flags          // item flags
);
```

This operation adds a new item to the indicated list. The item is added after the previously added item.

In addition to the argument modes defined in Common Data Structures on page 177, **item_flags** may also take the following flag values: **IN_COPY_VALUE**, **DEPENDENT_LIST**. The argument passing flags defined in Common Data Structures on page 177 may be used here to indicate the intended parameter passing mode of an argument.

If the **IN_COPY_VALUE** flag is set, a copy of the argument value is made and used instead.

If a list structure is added as an item (e.g., a "sublist"), the **DEPENDENT_LIST** flag may be specified to indicate that the sublist should be freed when the parent list is freed.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.5.3 free

```
void free ( );                                  // PIDL
```

This operation frees the list structure and any associated memory (an implicit call to the list **free_memory** operation is done).

11.5.4 free_memory

```
void free_memory ( );                           // PIDL
```

This operation frees any dynamically allocated out-arg memory associated with the list. The list structure itself is not freed.

11.5.5 get_count

```
void get_count (                                                    // PIDL
    out long            count       // number of entries in the list
);
```

This operation returns the total number of items added to the list.

11.5.6 create_operation_list

This operation, which creates a pseudo-object, is defined in the ORB interface.

```
void create_operation_list (                                // PIDL
    in OperationDef     oper,       // operation
    out NVList          new_list    // argument definitions
);
```

This operation returns an **NVList** initialized with the argument descriptions for a given operation. The information is returned in a form that may be used in *Dynamic Invocation* requests. The arguments are returned in the same order as they were defined for the operation.

The list **free** operation is used to free the returned information.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12 Dynamic Skeleton Interface

The Dynamic Skeleton Interface (DSI) allows dynamic handling of object invocations. That is, rather than being accessed through a skeleton that is specific to a particular operation, an object's implementation is reached through an interface that provides access to the operation name and parameters in a manner analogous to the client side's Dynamic Invocation Interface. Purely static knowledge of those parameters may be used, or dynamic knowledge (perhaps determined through an Interface Repository) may also be used, to determine the parameters.

12.1 Introduction

The Dynamic Skeleton Interface is a way to deliver requests from an ORB to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. This contrasts with the type-specific, OMG IDL-based skeletons, but serves the same architectural role.

DSI is the server side's analogue to the client side's Dynamic Invocation Interface (DII). Just as the implementation of an object cannot distinguish whether its client is using type-specific stubs or the DII, the client who invokes an object cannot determine whether the implementation is using a type-specific skeleton or the DSI to connect the implementation to the ORB.

.

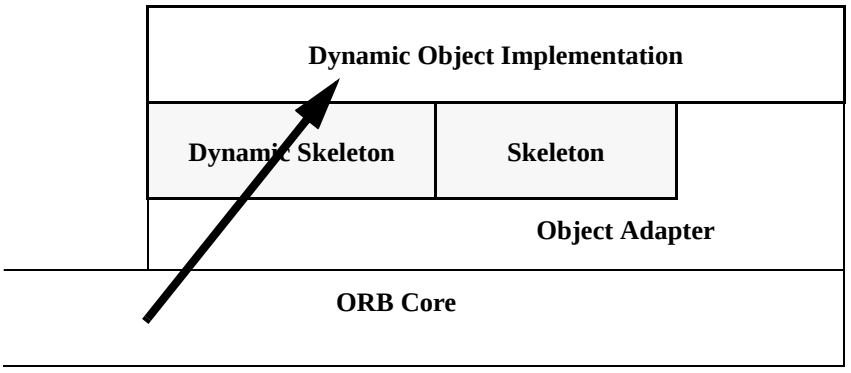


Figure 12.1 - Requests are delivered through skeletons, including dynamic ones

DSI, like DII, has many applications beyond interoperability solutions. Uses include interactive software development tools based on interpreters, debuggers, and monitors that want to dynamically interpose on objects, and support for dynamically-typed languages such as LISP.

12.2 Overview

The basic idea of the DSI is to implement all requests on a particular object by having the ORB invoke the same upcall routine, a Dynamic Implementation Routine (DIR). Since in any language binding all DIRs have the same signature, a single DIR could be used as the implementation for many objects, with different interfaces.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The DIR is passed all the explicit operation parameters, and an indication of the object that was invoked and the operation that was requested. The information is encoded in the request parameters. The DIR can use the invoked object, its object adapter, and the Interface Repository to learn more about the particular object and invocation. It can access and operate on individual parameters. It can make the same use of an object adapter as other object implementations.

This chapter describes the elements of the DSI that are common to all object adapters that provide a DSI. See Single Servant, Many Objects and Types, Using DSI on page 355 for the specification of the DSI for the Portable Object Adapter.

12.3 ServerRequestPseudo-Object

12.3.1 ExplicitRequest State: ServerRequestPseudo-Object

The ServerRequest pseudo-object captures the explicit state of a request for the DSI, analogous to the Request pseudo-object in the DII. The object adapter dispatches an invocation to a DSI-based object implementation by passing an instance of ServerRequest to the DIR associated with the object implementation. The following shows how it provides access to the request information:

```
module CORBA {
    ...
    interface ServerRequest { //                          PIDL
        readonly attribute Identifier    operation;
        void          arguments(inout NVList nv);
        Context       ctx();
        void          set_result(in Any val);
        void          set_exception(in Any val);
    };
};
```

The identity and/or reference of the target object of the invocation is provided by the object adapter and its language mapping. In the context of a bridge, the target object will typically be a proxy for an object in some other ORB.

The **operation** attribute provides the identifier naming the operation being invoked; according to OMG IDL's rules, these names must be unique among all operations supported by the object's "most-derived" interface. Note that the operation names for getting and setting attributes are **_get_<attribute_name>** and **_set_<attribute_name>**, respectively. The operation attribute can be accessed by the DIR at any time.

Operation parameter types will be specified, and "in" and "inout" argument values will be retrieved, with **arguments**. Unless it calls **set_exception**, the DIR must call **arguments** exactly once, even if the operation signature contains no parameters. Once **arguments** or **set_exception** has been called, calling **arguments** on the same **ServerRequest** will result in a BAD_INV_ORDER system exception with standard minor code 7. The DIR must pass in to **arguments** an **NVList** initialized with **TypeCodes** and **Flags** describing the parameter types for the operation, in the order in which they appear in the IDL specification (left to right). A potentially-different **NVList** will be returned from **arguments**, with the "in" and "inout" argument values supplied. If it does not call **set_exception**, the DIR must supply the returned **NVList** with return values for any "out" arguments before returning, and may also change the return values for any "inout" arguments.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

When the operation is not an attribute access, and the operation's IDL definition contains a context expression, **ctx** will return the context information specified in IDL for the operation. Otherwise it will return a nil **Context** reference. Calling **ctx** before **arguments** has been called or after **ctx**, **set_result,** or **set_exception** has been called will result in a BAD_INV_ORDER system exception with standard minor code 8.

The **set_result** operation is used to specify any return value for the call. Unless **set_exception** is called, if the invoked operation has a non-void result type, **set_result** must be called exactly once before the DIR returns. If the operation has a void result type, **set_result** may optionally be called once with an **Any** whose type is **tk_void**. Calling **set_result** before **arguments** has been called or after **set_result** or **set_exception** has been called will result in a BAD_INV_ORDER system exception with standard minor code 8. Calling **set_result** without having previously called ctx when the operation IDL contains a context expression will result in a MARSHAL system exception with standard minor code 2. If the **NVList** passed to arguments did not describe all parameters passed by the client, it may result in a MARSHAL system exception with standard minor code 3.

The DIR may call **set_exception** at any time to return an exception to the client. The **Any** passed to **set_exception** must contain either a system exception or one of the user exceptions specified in the **raises** expression of the invoked operation's IDL definition. Passing in an **Any** that does not contain an exception will result in a BAD_PARAM system exception with standard minor code 21. Passing in an unlisted user exception will result in either the DIR receiving a BAD_PARAM system exception with standard minor code 22 or in the client receiving an UNKNOWN system exception with standard minor code 1.

See each language mapping for a description of the memory management aspects of the parameters to the **ServerRequest** operations.

12.4 DSI: Language Mapping

Because DSI is defined in terms of a pseudo-object, special attention must be paid to it in the language mapping. This section provides general information about mapping the Dynamic Skeleton Interface to programming languages. Each language provides its own mapping for DSI.

12.4.1 ServerRequest's Handling of Operation Parameters

There is no requirement that a **ServerRequest** pseudo-object be usable as a general argument in OMG IDL operations, or listed in "orb.idl."

The client-side memory management rules normally applied to pseudo-objects do not strictly apply to a ServerRequest's handling of operation parameters. Instead, the memory associated with parameters follows the memory management rules applied to data passed from skeletons into statically typed implementation routines, and vice versa.

12.4.2 Registering Dynamic Implementation Routines

In an ORB implementation, the Dynamic Skeleton Interface is supported entirely through the Object Adapter. An Object Adapter does not have to support the Dynamic Skeleton Interface but, if it does, the Object Adapter is responsible for the details.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

13 Dynamic Management of Any Values

An **any** can be passed to a program that doesn't have any static information for the type of the **any** (code generated for the type by an IDL compiler has not been compiled with the object implementation). As a result, the object receiving the **any** does not have a portable method of using it.

The facility presented here enables traversal of the data value associated with an **any** at runtime and extraction of the primitive constituents of the data value. This is especially helpful for writing powerful generic servers (bridges, event channels supporting filtering).

Similarly, this facility enables the construction of an **any** at runtime, without having static knowledge of its type. This is especially helpful for writing generic clients (bridges, browsers, debuggers, user interface tools).

13.1 Overview

Unless explicitly stated otherwise, all IDL presented in *Overview* through *Usage in C++ Language* is part of the **DynamicAny** module.

Any values can be dynamically interpreted (traversed) and constructed through **DynAny** objects. A **DynAny** object is associated with a data value, which corresponds to a copy of the value inserted into an **any**.

A **DynAny** object may be viewed as an ordered collection of component **DynAny**s. For **DynAny**s representing a basic type, such as **long**, or a type without components, such as an empty exception, the ordered collection of components is empty. Each **DynAny** object maintains the notion of a current position into its collection of component **DynAny**s. The current position is identified by an index value that runs from 0 to n–1, where n is the number of components. The special index value –1 indicates a current position that points nowhere. For values that cannot have a current position (such as an empty exception), the index value is fixed at –1. If a **DynAny** is initialized with a value that has components, the index is initialized to 0. After creation of an uninitialized **DynAny** (that is, a **DynAny** that has no value but a **TypeCode** that permits components), the current position depends on the type of value represented by the **DynAny**. (The current position is set to 0 or –1, depending on whether the new **DynAny** gets default values for its components.)

The iteration operations **rewind**, **seek**, and **next** can be used to change the current position and the **current_component** operation returns the component at the current position. The **component_count** operation returns the number of components of a **DynAny**. Collectively, these operations enable iteration over the components of a **DynAny**, for example, to (recursively) examine its contents.

A constructed **DynAny** object is a **DynAny** object associated with a constructed type. There is a different interface, inheriting from the **DynAny** interface, associated with each kind of constructed type in IDL (fixed, enum, struct, sequence, union, array, exception, and valuetype).

A constructed **DynAny** object exports operations that enable the creation of new **DynAny** objects, each of them associated with a component of the constructed data value.

As an example, a **DynStruct** is associated with a struct value. This means that the **DynStruct** may be seen as owning an ordered collection of components, one for each structure member. The **DynStruct** object exports operations that enable the creation of new **DynAny** objects, each of them associated with a member of the struct.

If a **DynAny** object has been obtained from another (constructed) **DynAny** object, such as a **DynAny** representing a structure member that was created from a **DynStruct**, the member **DynAny** is logically contained in the **DynStruct**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Destroying a top-level **DynAny** object (one that was not obtained as a component of another **DynAny**) also destroys any component **DynAny** objects obtained from it. Destroying a non-top level **DynAny** object does nothing. Invoking operations on a destroyed top-level **DynAny** or any of its descendants raises OBJECT_NOT_EXIST. Note that simply releasing all references to a **DynAny** object does not delete the **DynAny** or components; each **DynAny** created with one of the create operations or with the **copy** operation must be explicitly destroyed to avoid memory leaks.

If the programmer wants to destroy a **DynAny** object but still wants to manipulate some component of the data value associated with it, then he or she should first create a **DynAny** for the component and, after that, make a copy of the created **DynAny** object.

The behavior of **DynAny** objects has been defined in order to enable efficient implementations in terms of allocated memory space and speed of access. **DynAny** objects are intended to be used for traversing values extracted from **any**s or constructing values of **any**s at runtime. Their use for other purposes is not recommended.

13.2 DynAny API

The **DynAny** API comprises the following IDL definitions, located in the **DynamicAny** module:

```
// IDL
// File: DynamicAny.idl
#ifndef _DYNAMIC_ANY_IDL_
#define _DYNAMIC_ANY_IDL_

import ::CORBA;

module DynamicAny {
    typeprefix DynamicAny "omg.org";

    local interface DynAny {
        exception InvalidValue {};
        exception TypeMismatch {};

        CORBA::TypeCode type();

        void assign(in DynAny dyn_any) raises(TypeMismatch);
        void from_any(in any value) raises(TypeMismatch, InvalidValue);
        any to_any();

        boolean equal(in DynAny dyn_any);

        void destroy();
        DynAny copy();

        void insert_boolean(in boolean value)
            raises(TypeMismatch, InvalidValue);
        void insert_octet(in octet value)
            raises(TypeMismatch, InvalidValue);
        void insert_char(in char value)
            raises(TypeMismatch, InvalidValue);
        void insert_short(in short value)
```


© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        raises(TypeMismatch, InvalidValue);
    void insert_ushort(in unsigned short value)
        raises(TypeMismatch, InvalidValue);
    void insert_long(in long value)
        raises(TypeMismatch, InvalidValue);
    void insert_ulong(in unsigned long value)
        raises(TypeMismatch, InvalidValue);
    void insert_float(in float value)
        raises(TypeMismatch, InvalidValue);
    void insert_double(in double value)
        raises(TypeMismatch, InvalidValue);
    void insert_string(in string value)
        raises(TypeMismatch, InvalidValue);
    void insert_reference(in Object value)
        raises(TypeMismatch, InvalidValue);
    void insert_typecode(in CORBA::TypeCode value)
        raises(TypeMismatch, InvalidValue);
    void insert_longlong(in long long value)
        raises(TypeMismatch, InvalidValue);
    void insert_ulonglong(in unsigned long long value)
        raises(TypeMismatch, InvalidValue);
    void insert_longdouble(in long double value)
        raises(TypeMismatch, InvalidValue);
    void insert_wchar(in wchar value)
        raises(TypeMismatch, InvalidValue);
    void insert_wstring(in wstring value)
        raises(TypeMismatch, InvalidValue);
    void insert_any(in any value)
        raises(TypeMismatch, InvalidValue);
    void insert_dyn_any(in DynAny value)
        raises(TypeMismatch, InvalidValue);
    void insert_val(in ValueBase value)
        raises(TypeMismatch, InvalidValue);

    boolean get_boolean()
        raises(TypeMismatch, InvalidValue);
    octet get_octet()
        raises(TypeMismatch, InvalidValue);
    char get_char()
        raises(TypeMismatch, InvalidValue);
    short get_short()
        raises(TypeMismatch, InvalidValue);
    unsigned short get_ushort()
        raises(TypeMismatch, InvalidValue);
    long get_long()
        raises(TypeMismatch, InvalidValue);
    unsigned long get_ulong()
        raises(TypeMismatch, InvalidValue);
    float get_float()
        raises(TypeMismatch, InvalidValue);
    double get_double()
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        raises(TypeMismatch, InvalidValue);
    string get_string()
        raises(TypeMismatch, InvalidValue);
    Object get_reference()
        raises(TypeMismatch, InvalidValue);
    CORBA::TypeCode get_typecode()
        raises(TypeMismatch, InvalidValue);
    long long get_longlong()
        raises(TypeMismatch, InvalidValue);
    unsigned long long get_ulonglong()
        raises(TypeMismatch, InvalidValue);
    long double get_longdouble()
        raises(TypeMismatch, InvalidValue);
    wchar get_wchar()
        raises(TypeMismatch, InvalidValue);
    wstring get_wstring()
        raises(TypeMismatch, InvalidValue);
    any get_any()
        raises(TypeMismatch, InvalidValue);
    DynAny get_dyn_any()
        raises(TypeMismatch, InvalidValue);
    ValueBase get_val()
        raises(TypeMismatch, InvalidValue);

    boolean seek(in long index);
    void rewind();
    boolean next();
    unsigned long component_count();
    DynAny current_component() raises(TypeMismatch);

    void insert_abstract(in CORBA::AbstractBase value)
        raises(TypeMismatch, InvalidValue);
    CORBA::AbstractBase get_abstract()
        raises(TypeMismatch, InvalidValue);

    void insert_boolean_seq(in CORBA::BooleanSeq value)
        raises(TypeMismatch, InvalidValue);
    void insert_octet_seq(in CORBA::OctetSeq value)
        raises(TypeMismatch, InvalidValue);
    void insert_char_seq(in CORBA::CharSeq value)
        raises(TypeMismatch, InvalidValue);
    void insert_short_seq(in CORBA::ShortSeq value)
        raises(TypeMismatch, InvalidValue);
    void insert_ushort_seq(in CORBA::UShortSeq value)
        raises(TypeMismatch, InvalidValue);
    void insert_long_seq(in CORBA::LongSeq value)
        raises(TypeMismatch, InvalidValue);
    void insert_ulong_seq(in CORBA::ULongSeq value)
        raises(TypeMismatch, InvalidValue);
    void insert_float_seq(in CORBA::FloatSeq value)
        raises(TypeMismatch, InvalidValue);
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    void insert_double_seq(in CORBA::DoubleSeq value)
        raises(TypeMismatch, InvalidValue);
    void insert_longlong_seq(in CORBA::LongLongSeq value)
        raises(TypeMismatch, InvalidValue);
    void insert_ulonglong_seq(in CORBA::ULongLongSeq value)
        raises(TypeMismatch, InvalidValue);
    void insert_longdouble_seq(in CORBA::LongDoubleSeq value)
        raises(TypeMismatch, InvalidValue);
    void insert_wchar_seq(in CORBA::WCharSeq value)
        raises(TypeMismatch, InvalidValue);
    CORBA::BooleanSeq get_boolean_seq()
        raises(TypeMismatch, InvalidValue);
    CORBA::OctetSeq get_octet_seq()
        raises(TypeMismatch, InvalidValue);
    CORBA::CharSeq get_char_seq()
        raises(TypeMismatch, InvalidValue);
    CORBA::ShortSeq get_short_seq()
        raises(TypeMismatch, InvalidValue);
    CORBA::UShortSeq get_ushort_seq()
        raises(TypeMismatch, InvalidValue);
    CORBA::LongSeq get_long_seq()
        raises(TypeMismatch, InvalidValue);
    CORBA::ULongSeq get_ulong_seq()
        raises(TypeMismatch, InvalidValue);
    CORBA::FloatSeq get_float_seq()
        raises(TypeMismatch, InvalidValue);
    CORBA::DoubleSeq get_double_seq()
        raises(TypeMismatch, InvalidValue);
    CORBA::LongLongSeq get_longlong_seq()
        raises(TypeMismatch, InvalidValue);
    CORBA::ULongLongSeq get_ulonglong_seq()
        raises(TypeMismatch, InvalidValue);
    CORBA::LongDoubleSeq get_longdouble_seq()
        raises(TypeMismatch, InvalidValue);
    CORBA::WCharSeq get_wchar_seq()
        raises(TypeMismatch, InvalidValue);
};

local interface DynFixed : DynAny {
    string get_value();
    boolean set_value(in string val) raises(TypeMismatch, InvalidValue);
};

local interface DynEnum : DynAny {
    string get_as_string();
    void set_as_string(in string value) raises(InvalidValue);
    unsigned long get_as_ulong();
    void set_as_ulong(in unsigned long value) raises(InvalidValue);
};
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
typedef string FieldName;

struct NameValuePair {
    FieldName id;
    any value;
};

typedef sequence<NameValuePair> NameValuePairSeq;

struct NameDynAnyPair {
    FieldName id;
    DynAny value;
};

typedef sequence<NameDynAnyPair> NameDynAnyPairSeq;

local interface DynStruct : DynAny {
    FieldName current_member_name()
        raises(TypeMismatch, InvalidValue);
    CORBA::TCKind current_member_kind()
        raises(TypeMismatch, InvalidValue);
    NameValuePairSeq get_members();
    void set_members(in NameValuePairSeq value)
        raises(TypeMismatch, InvalidValue);
    NameDynAnyPairSeq get_members_as_dyn_any();
    void set_members_as_dyn_any(in NameDynAnyPairSeq value)
        raises(TypeMismatch, InvalidValue);
};

local interface DynUnion : DynAny {
    DynAny get_discriminator();
    void set_discriminator(in DynAny d) raises(TypeMismatch);
    void set_to_default_member() raises(TypeMismatch);
    void set_to_no_active_member() raises(TypeMismatch);
    boolean has_no_active_member();
    CORBA::TCKind discriminator_kind();
    DynAny member() raises(InvalidValue);
    FieldName member_name() raises(InvalidValue);
    CORBA::TCKind member_kind() raises(InvalidValue);
};

typedef sequence<any> AnySeq;
typedef sequence<DynAny> DynAnySeq;

local interface DynSequence : DynAny {
    unsigned long get_length();
    void set_length(in unsigned long len) raises(InvalidValue);
    AnySeq get_elements();
    void set_elements(in AnySeq value)
        raises(TypeMismatch, InvalidValue);
    DynAnySeq get_elements_as_dyn_any();
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    void set_elements_as_dyn_any(in DynAnySeq value)
        raises(TypeMismatch, InvalidValue);
};

local interface DynArray : DynAny {
    AnySeq get_elements();
    void set_elements(in AnySeq value)
        raises(TypeMismatch, InvalidValue);
    DynAnySeq get_elements_as_dyn_any();
    void set_elements_as_dyn_any(in DynAnySeq value)
        raises(TypeMismatch, InvalidValue);
};

local interface DynValueCommon : DynAny {
    boolean is_null();
    void set_to_null();
    void set_to_value();
};

local interface DynValue : DynValueCommon {
    FieldName current_member_name()
        raises(TypeMismatch, InvalidValue);
    CORBA::TCKind current_member_kind()
        raises(TypeMismatch, InvalidValue);
    NameValuePairSeq get_members()
        raises(InvalidValue);
    void set_members(in NameValuePairSeq value)
        raises(TypeMismatch, InvalidValue);
    NameDynAnyPairSeq get_members_as_dyn_any()
        raises(InvalidValue);
    void set_members_as_dyn_any(in NameDynAnyPairSeq value)
    raises(TypeMismatch, InvalidValue);
};

local interface DynValueBox : DynValueCommon {
    any get_boxed_value()
        raises(InvalidValue);
    void set_boxed_value(in any boxed)
        raises(TypeMismatch, InvalidValue);
    DynAny get_boxed_value_as_dyn_any()
        raises(InvalidValue);
    void set_boxed_value_as_dyn_any(in DynAny boxed)
        raises(TypeMismatch);
};

exception MustTruncate { };

local interface DynAnyFactory {
    exception InconsistentTypeCode {};
    DynAny create_dyn_any(in any value)
        raises(InconsistentTypeCode);
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        DynAny
            create_dyn_any_from_type_code(in CORBA::TypeCode type)
            raises(InconsistentTypeCode);

        DynAny create_dyn_any_without_truncation(in any value)
             raises(InconsistentTypeCode, MustTruncate);
        DynAnySeq create_multiple_dyn_anys(
            in AnySeq values,
            in boolean allow_truncate)
            raises(InconsistentTypeCode, MustTruncate);

        AnySeq create_multiple_anys(in DynAnySeq values);
    };
}; // module DynamicAny
#endif // _DYNAMIC_ANY_IDL_
```

13.2.1 Creating a DynAny Object

A **DynAny** object can be created as a result of:

- invoking an operation on an existing **DynAny** object.
- invoking an operation on a **DynAnyFactory** object.

A constructed **DynAny** object supports operations that enable the creation of new **DynAny** objects encapsulating access to the value of some constituent. **DynAny** objects also support the **copy** operation for creating new **DynAny** objects.

In addition, **DynAny** objects can be created by invoking operations on the **DynAnyFactory** object. A reference to the **DynAnyFactory** object is obtained by calling **CORBA::ORB::resolve_initial_references** with the **identifier** parameter set to **"DynAnyFactory."**

```
local interface DynAnyFactory {
    exception InconsistentTypeCode {};
    DynAny create_dyn_any(in any value)
        raises(InconsistentTypeCode);
    DynAny create_dyn_any_from_type_code(in CORBA::TypeCode type)
        raises(InconsistentTypeCode);
};
```

The **create_dyn_any** operation creates a new **DynAny** object from an **any** value. A copy of the **TypeCode** associated with the **any** value is assigned to the resulting **DynAny** object. The value associated with the **DynAny** object is a copy of the value in the original any. The **create_dyn_any** operation sets the current position of the created **DynAny** to zero if the passed value has components; otherwise, the current position is set to −1. The operation raises **InconsistentTypeCode** if **value** has a **TypeCode** with a **TCKind** of **tk_Principal** or **tk_native**.

The **create_dyn_any_from_type_code** operation creates a **DynAny** from a **TypeCode**. Depending on the **TypeCode**, the created object may be of type **DynAny**, or one of its derived types, such as **DynStruct**. The returned reference can be narrowed to the derived type.

For both **create_dyn_any** and **create_dyn_any_from_type_code**, the source type code is copied into the **DynAny** object unchanged. This means that, after creation of a **DynAny** object, the source type code and the type code inside the **DynAny** must compare equal as determined by **TypeCode::equal**. The same is true for type codes extracted from a

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

DynAny with the type operation and for type codes that are part of any values that are constructed from a **DynAny**: such type codes compare equal to the type code that was originally used to create the **DynAny**. For a given parent **DynAny** with its associated **TypeCode**, the **TypeCode** of a component **DynAny** also compares equal to the corresponding results of the **member_type** or **component_type** operation on the parent **TypeCode**.

The **create_dyn_any_without_truncation** operation has the same semantics as **create_dyn_any**, but will raise the MustTruncate exception if it cannot avoid truncating a valuetype.

The **create_multiple_dyn_anys** operation converts a sequence of anys into a sequence of **DynAnys**, ensuring that each reference to a valuetype instance is converted consistently to the same **DynValue** or **DynValueBox** instance. If the **allow_truncate** parameter is false, the operation will raise the MustTruncate exception if it cannot avoid truncating a valuetype.

The **create_multiple_anys** operation converts a sequence of **DynAnys** into a sequence of **anys**, ensuring that each **DynValue** or **DynValueBox** instance is consistently converted to the same valuetype instance.

Creation of **DynAnys** with **TCKind tk_null** and **tk_void** is legal and results in the creation of a **DynAny** without a value and with zero components.

In all cases, a **DynAny** constructed from a **TypeCode** has an initial default value. The default values of basic types are:

- **FALSE** for **Boolean**
- zero for numeric types
- zero for types **octet**, **char**, and **wchar**
- the empty string for **string** and **wstring**
- nil for object references
- a type code with a **TCKind** value of **tk_null** for type codes
- for **any** values, an **any** containing a type code with a **TCKind** value of **tk_null** type and no value

For complex types, creation of the corresponding **DynAny** assigns a default value as follows:

- For **DynSequence**, the operation sets the current position to –1 and creates an empty sequence.
- For **DynEnum**, the operation sets the current position to –1 and sets the value of the enumerator to the first enumerator value indicated by the **TypeCode**.
- For **DynFixed**, operations set the current position to –1 and sets the value zero.
- For **DynStruct**, the operation sets the current position to –1 for empty exceptions and to zero for all other **TypeCode**s. The members (if any) are (recursively) initialized to their default values.
- For **DynArray**, the operation sets the current position to zero and (recursively) initializes elements to their default value.
- For **DynUnion**, the operation sets the current position to zero. The discriminator value is set to a value consistent with the first named member of the union. That member is activated and (recursively) initialized to its default value.
- **DynValue** and **DynValueBox** are initialized to a null value.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Dynamic interpretation of an **any** usually involves creating a **DynAny** object using **DynAnyFactory::create_dyn_any** as the first step. Depending on the type of the **any**, the resulting **DynAny** object reference can be narrowed to a **DynFixed**, **DynStruct**, **DynSequence**, **DynArray**, **DynUnion**, **DynEnum**, or **DynValue** object reference.

Dynamic creation of an **any** involves creating a **DynAny** object using **DynAnyFactory::create_dyn_any_from_type_code**, passing the **TypeCode** associated with the value to be created. The returned reference is narrowed to one of the complex types, such as **DynStruct**, if appropriate. Then, the value can be initialized by means of invoking operations on the resulting object. Finally, the **to_any** operation can be invoked to create an **any** value from the constructed **DynAny**.

13.2.2 The DynAny Interface

The following operations can be applied to a **DynAny** object:

- Obtaining the **TypeCode** associated with the **DynAny** object.
- Generating an **any** value from the **DynAny** object.
- Comparing two **DynAny** objects for equality.
- Destroying the **DynAny** object.
- Creating a **DynAny** object as a copy of the **DynAny** object.
- Inserting/getting a value of some basic type into/from the **DynAny** object.
- Iterating through the components of a **DynAny.**
- Initializing a **DynAny** object from another **DynAny** object.
- Initializing a **DynAny** object from an **any** value.

13.2.2.1 Obtaining the TypeCode associated with a DynAny object

CORBA::TypeCode type();

A **DynAny** object is created with a **TypeCode** value assigned to it. This **TypeCode** value determines the type of the value handled through the **DynAny** object. The **type** operation returns the **TypeCode** associated with a **DynAny** object.

Note that the **TypeCode** associated with a **DynAny** object is initialized at the time the **DynAny** is created and cannot be changed during the lifetime of the **DynAny** object.

13.2.2.2 Initializing a DynAny object from another DynAny object

void assign(in DynAny dyn_any) raises(TypeMismatch);

The **assign** operation initializes the value associated with a **DynAny** object with the value associated with another **DynAny** object.

If the type of the passed **DynAny** is not equivalent to the type of target **DynAny**, the operation raises TypeMismatch. The current position of the target **DynAny** is set to zero for values that have components and to −1 for values that do not have components.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

13.2.2.3 Initializing a DynAny object from an any value

void from_any(in any value) raises(TypeMismatch, InvalidValue);

The **from_any** operation initializes the value associated with a **DynAny** object with the value contained in an **any**.

If the type of the passed **Any** is not equivalent to the type of target **DynAny**, the operation raises TypeMismatch. If the passed **Any** does not contain a legal value (such as a null string), the operation raises InvalidValue. The current position of the target **DynAny** is set to zero for values that have components and to –1 for values that do not have components.

13.2.2.4 Generating an any value from a DynAny object

any to_any();

The **to_any** operation creates an **any** value from a **DynAny** object. A copy of the **TypeCode** associated with the **DynAny** object is assigned to the resulting **any**. The value associated with the **DynAny** object is copied into the **any**.

13.2.2.5 Comparing DynAny values

boolean equal(in DynAny dyn_any);

The equal operation compares two **DynAny** references for equality and returns true if the **DynAny**s are equal, false otherwise. For **DynAny** references that are not derived from **DynValueCommon**, they are equal if their **TypeCode**s are equivalent and, recursively, all component **DynAny**s are equal. For **DynAny** references that are derived from **DynValueCommon**, they are equal only if they are exactly the same reference. The current position of the two **DynAny**s being compared has no effect on the result of equal. To determine equality of object references, the equal operation uses **Object::is_equivalent**. To determine equality of type codes, the equal operation uses **TypeCode::equivalent**.

NOTE: If two **DynAny**s happen to contain *values* of type **TypeCode**, these values are compared using **TypeCode::equal**. The type codes that *describe* the values of **DynAny**s are always compared using **TypeCode::equivalent**, however. (In the case of comparing two **DynAny**s containing type code values, the type codes describing these type code values are **tk_TypeCode** in each **DynAny**, and will therefore always compare as equivalent.)

13.2.2.6 Destroying a DynAny object

void destroy();

The **destroy** operation destroys a **DynAny** object. This operation frees any resources used to represent the data value associated with a **DynAny** object. **destroy** must be invoked on references obtained from one of the creation operations on the **DynAnyFactory** interface or on a reference returned by **DynAny::copy** to avoid resource leaks. Invoking **destroy** on component **DynAny** objects (for example, on objects returned by the **current_component** operation) does nothing.

Destruction of a **DynAny** object implies destruction of all **DynAny** objects obtained from it. That is, references to components of a destroyed **DynAny** become invalid; invocations on such references raise OBJECT_NOT_EXIST.

It is possible to manipulate a component of a **DynAny** beyond the life time of the **DynAny** from which the component was obtained by making a copy of the component with the **copy** operation before destroying the **DynAny** from which the component was obtained.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

13.2.2.7 Creating a copy of a DynAny object

DynAny copy();

The **copy** operation creates a new **DynAny** object whose value is a deep copy of the **DynAny** on which it is invoked. The operation is polymorphic, that is, invoking it on one of the types derived from **DynAny**, such as **DynStruct**, creates the derived type but returns its reference as the **DynAny** base type.

13.2.2.8 Accessing a value of some basic type in a DynAny object

The insert and get operations enable insertion/extraction of basic data type values into/from a **DynAny** object.

Both bounded and unbounded strings are inserted using **insert_string** and **insert_wstring**. These operations raise the **InvalidValue** exception if the string inserted is longer than the bound of a bounded string.

Calling an insert or get operation on a **DynAny** that has components but has a current position of −1 raises **InvalidValue**.

Get operations raise **TypeMismatch** if the accessed component in the **DynAny** is of a type that is not equivalent to the requested type. (Note that **get_string** and **get_wstring** are used for both unbounded and bounded strings.)

A type is consistent for inserting or extracting a value if its **TypeCode** is equivalent to the **TypeCode** contained in the **DynAny** or, if the **DynAny** has components, is equivalent to the **TypeCode** of the **DynAny** at the current position.

The **get_dyn_any** and **insert_dyn_any** operations are provided to deal with **any** values that contain another **any**. The operations behave identically to **get_any** and **insert_any**, but use parameters of type **DynAny** (instead of any); they are useful to avoid otherwise redundant conversions between any and **DynAny**.

Calling an insert or get operation leaves the current position unchanged.

These operations are necessary to handle basic **DynAny** objects but are also helpful to handle constructed **DynAny** objects. Inserting a basic data type value into a constructed **DynAny** object implies initializing the current component of the constructed data value associated with the **DynAny** object. For example, invoking **insert_boolean** on a **DynStruct** implies inserting a boolean data value at the current position of the associated struct data value. If **dyn_construct** points to a constructed **DynAny** object, then:

```
result = dyn_construct->get_boolean();
```

has the same effect as:

```
DynamicAny::DynAny_var temp =
dyn_construct->current_component();
result = temp->get_boolean();
```

Calling an insert or get operation on a **DynAny** whose current component itself has components raises TypeMismatch.

In addition, availability of these operations enable the traversal of **any**s associated with sequences of basic data types without the need to generate a **DynAny** object for each element in the sequence.

In the same way that basic types are inserted/extracted from a **DynAny** object, arrays or sequences of basic types can be inserted/extracted from a **DynAny**. For example, the **get_boolean_seq** operation extracts a sequence of **boolean**s from a **DynAny** that contains either a sequence or an array of **boolean**s, and the **insert_boolean_seq** operation stores the sequence back into the **DynAny**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The **TypeCode** of the **DynAny**, or the **TypeCode** of the component at the current position of the **DynAny**, must be equivalent to a sequence or array **TypeCode** with the basic type as its element, otherwise the operations raise TypeMismatch. For the insert operations, if the length of the sequence is incompatible with a bounded sequence or array represented by the **DynAny**, then the operations raise InvalidValue.

13.2.2.9 Iterating through components of a DynAny

The **DynAny** interface allows a client to iterate through the components of the values pointed to by **DynStruct**, **DynSequence**, **DynArray**, **DynUnion**, **DynAny**, and **DynValue** objects.

As mentioned previously, a **DynAny** object may be seen as an ordered collection of components, together with a current position.

boolean seek(in long index);

The **seek** operation sets the current position to **index**. The current position is indexed 0 to n–1, that is, index zero corresponds to the first component. The operation returns true if the resulting current position indicates a component of the **DynAny** and false if **index** indicates a position that does not correspond to a component.

Calling **seek** with a negative index is legal. It sets the current position to –1 to indicate no component and returns false. Passing a non-negative index value for a **DynAny** that does not have a component at the corresponding position sets the current position to –1 and returns false.

void rewind();

The **rewind** operation is equivalent to calling **seek(0);**

boolean next();

The **next** operation advances the current position to the next component. The operation returns true while the resulting current position indicates a component, false otherwise. A false return value leaves the current position at –1. Invoking **next** on a **DynAny** without components leaves the current position at –1 and returns false.

unsigned long component_count();

The **component_count** operation returns the number of components of a **DynAny**. For a **DynAny** without components, it returns zero. The operation only counts the components at the top level. For example, if **component_count** is invoked on a **DynStruct** with a single member, the return value is 1, irrespective of the type of the member.

For sequences, the operation returns the current number of elements. For structures, exceptions, and valuetypes, the operation returns the number of members. For arrays, the operation returns the number of elements. For unions, the operation returns 2 if the discriminator indicates that a named member is active; otherwise, it returns 1. For **DynFixed** and **DynEnum**, the operation returns zero.

DynAny current_component() raises(TypeMismatch);

The **current_component** operation returns the **DynAny** for the component at the current position. It does not advance the current position, so repeated calls to **current_component** without an intervening call to **rewind**, **next**, or **seek** return the same component.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The returned **DynAny** object reference can be used to get/set the value of the current component. If the current component represents a complex type, the returned reference can be narrowed based on the **TypeCode** to get the interface corresponding to the to the complex type.

Calling **current_component** on a **DynAny** that cannot have components, such as a **DynEnum** or an empty exception, raises TypeMismatch. Calling **current_component** on a **DynAny** whose current position is –1 returns a nil reference.

The iteration operations, together with **current_component**, can be used to dynamically compose an **any** value. After creating a dynamic any, such as a **DynStruct**, **current_component** and **next** can be used to initialize all the components of the value. Once the dynamic value is completely initialized, **to_any** creates the corresponding **any** value.

13.2.3 The DynFixed Interface

DynFixed objects are associated with values of the IDL **fixed** type.

```
local interface DynFixed : DynAny {
    string get_value();
    boolean set_value(in string val)
        raises (TypeMismatch, InvalidValue);
};
```

Because IDL does not have a generic type that can represent fixed types with arbitrary number of digits and arbitrary scale, the operations use the IDL **string** type.

The **get_value** operation returns the value of a **DynFixed**.

The **set_value** operation sets the value of the **DynFixed**. The **val** string must contain a **fixed** string constant in the same format as used for IDL fixed-point literals. However, the trailing **d** or **D** is optional. If **val** has more fractional digits than specified by the scale of the **DynFixed**, the extra digits are truncated. If the truncated value has more digits than the **DynFixed**, the operation raises InvalidValue. If the value is not too large, **set_value** returns **TRUE** if no truncation was required, **FALSE** otherwise. The return value is **TRUE** if **val** can be represented as the **DynFixed** without loss of precision. If **val** has more fractional digits than can be represented in the **DynFixed**, fractional digits are truncated and the return value is **FALSE**. If **val** does not contain a valid fixed-point literal or contains extraneous characters other than leading or trailing white space, the operation raises TypeMismatch.

13.2.4 The DynEnum Interface

DynEnum objects are associated with enumerated values.

```
local interface DynEnum : DynAny {
    string get_as_string();
    void set_as_string(in string value) raises(InvalidValue);
    unsigned long get_as_ulong();
    void set_as_ulong(in unsigned long value) raises(InvalidValue);
};
```

The **get_as_string** operation returns the value of the **DynEnum** as an IDL identifier.

The **set_as_string** operation sets the value of the **DynEnum** to the enumerated value whose IDL identifier is passed in the **value** parameter. If **value** contains a string that is not a valid IDL identifier for the corresponding enumerated type, the operation raises InvalidValue.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The **get_as_ulong** operation returns the value of the **DynEnum** as the enumerated value's ordinal value. Enumerators have ordinal values 0 to n–1, as they appear from left to right in the corresponding IDL definition.

The **set_as_ulong** operation sets the value of the **DynEnum** as the enumerated value's ordinal value. If **value** contains a value that is outside the range of ordinal values for the corresponding enumerated type, the operation raises InvalidValue.

The current position of a **DynEnum** is always –1.

13.2.5 The DynStruct Interface

DynStruct objects are associated with struct values and exception values.

```
typedef string FieldName;

struct NameValuePair {
    FieldName id;
    any value;
};
typedef sequence<NameValuePair> NameValuePairSeq;

struct NameDynAnyPair {
    FieldName id;
    DynAny value;
};
typedef sequence<NameDynAnyPair> NameDynAnyPairSeq;

local interface DynStruct : DynAny {
    FieldName current_member_name()
        raises(TypeMismatch, InvalidValue);
    CORBA::TCKind current_member_kind()
        raises(TypeMismatch, InvalidValue);
    NameValuePairSeq get_members();
    void set_members(in NameValuePairSeq value)
        raises(TypeMismatch, InvalidValue);
    NameDynAnyPairSeq get_members_as_dyn_any();
    void set_members_as_dyn_any(in NameDynAnyPairSeq value)
        raises(TypeMismatch, InvalidValue);
};

    FieldName current_member_name()
        raises(TypeMismatch, InvalidValue);
```

The **current_member_name** operation returns the name of the member at the current position. If the **DynStruct** represents an empty exception, the operation raises TypeMismatch. If the current position does not indicate a member, the operation raises InvalidValue.

This operation may return an empty string since the **TypeCode** of the value being manipulated may not contain the names of members.

```
    CORBA::TCKind current_member_kind()
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    raises(TypeMismatch, InvalidValue);
```

current_member_kind returns the **TCKind** associated with the member at the current position. If the **DynStruct** represents an empty exception, the operation raises TypeMismatch. If the current position does not indicate a member, the operation raises InvalidValue.

```
NameValuePairSeq get_members();
```

The **get_members** operation returns a sequence of name/value pairs describing the name and the value of each member in the struct associated with a **DynStruct** object. The sequence contains members in the same order as the declaration order of members as indicated by the **DynStruct**'s **TypeCode**. The current position is not affected. The member names in the returned sequence will be empty strings if the **DynStruct**'s **TypeCode** does not contain member names.

```
void set_members(in NameValuePairSeq value)
    raises(TypeMismatch, InvalidValue);
```

The **set_members** operation initializes the struct data value associated with a **DynStruct** object from a sequence of name value pairs. The operation sets the current position to zero if the passed sequences has non-zero length; otherwise, if an empty sequence is passed, the current position is set to −1.

Members must appear in the **NameValuePairSeq** in the order in which they appear in the IDL specification of the struct. If one or more sequence elements have a type that is not equivalent to the **TypeCode** of the corresponding member, the operation raises TypeMismatch. If the passed sequence has a number of elements that disagrees with the number of members as indicated by the **DynStruct**'s **TypeCode**, the operation raises InvalidValue.

If member names are supplied in the passed sequence, they must either match the corresponding member name in the **DynStruct**'s **TypeCode** or must be empty strings, otherwise, the operation raises TypeMismatch. Members must be supplied in the same order as indicated by the **DynStruct**'s **TypeCode**. (The operation makes no attempt to assign member values based on member names.)

The **get_members_as_dyn_any** and **set_members_as_dyn_any** operations have the same semantics as their **Any** counterparts, but accept and return values of type **DynAny** instead of **Any**.

DynStruct objects can also be used for handling exception values. In that case, members of the exceptions are handled in the same way as members of a struct.

13.2.6 The DynUnion Interface

DynUnion objects are associated with unions.

```
local interface DynUnion : DynAny {
    DynAny get_discriminator();
    void set_discriminator(in DynAny d)
        raises(TypeMismatch);
    void set_to_default_member()
        raises(TypeMismatch);
    void set_to_no_active_member()
        raises(TypeMismatch);
    boolean has_no_active_member()
        raises(InvalidValue);
    CORBA::TCKind discriminator_kind();
    DynAny member()
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        raises(InvalidValue);
    FieldName member_name()
        raises(InvalidValue);
    CORBA::TCKind member_kind()
        raises(InvalidValue);
    boolean is_set_to_default_member();
};
```

The **DynUnion** interface allows for the insertion/extraction of an OMG IDL union type into/from a **DynUnion** object.

A union can have only two valid current positions: zero, which denotes the discriminator, and one, which denotes the active member. The **component_count** value for a union depends on the current discriminator: it is 2 for a union whose discriminator indicates a named member, and 1 otherwise.

```
DynAny get_discriminator()
```

The **get_discriminator** operation returns the current discriminator value of the **DynUnion**.

```
void set_discriminator(in DynAny d)
    raises(TypeMismatch);
```

The **set_discriminator** operation sets the discriminator of the **DynUnion** to the specified value. If the **TypeCode** of the **d** parameter is not equivalent to the **TypeCode** of the union's discriminator, the operation raises TypeMismatch.

Setting the discriminator to a value that is consistent with the currently active union member does not affect the currently active member. Setting the discriminator to a value that is inconsistent with the currently active member deactivates the member and activates the member that is consistent with the new discriminator value (if there is a member for that value) by initializing the member to its default value.

Setting the discriminator of a union sets the current position to 0 if the discriminator value indicates a non-existent union member (**has_no_active_member** returns true in this case). Otherwise, if the discriminator value indicates a named union member, the current position is set to 1 (**has_no_active_member** returns false and **component_count** returns 2 in this case).

```
void set_to_default_member()
    raises(TypeMismatch);
```

The **set_to_default_member** operation sets the discriminator to a value that is consistent with the value of the **default** case of a union; it sets the current position to zero and causes **component_count** to return 2. Calling **set_to_default_member** on a union that does not have an explicit **default** case raises TypeMismatch.

```
void set_to_no_active_member()
    raises(TypeMismatch);
```

The **set_to_no_active_member** operation sets the discriminator to a value that does not correspond to any of the union's case labels; it sets the current position to zero and causes **component_count** to return 1. Calling **set_to_no_active_member** on a union that has an explicit **default** case or on a union that uses the entire range of discriminator values for explicit **case** labels raises TypeMismatch.

```
boolean has_no_active_member();
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The **has_no_active_member** operation returns true if the union has no active member (that is, the union's value consists solely of its discriminator because the discriminator has a value that is not listed as an explicit **case** label). Calling this operation on a union that has a **default** case returns false. Calling this operation on a union that uses the entire range of discriminator values for explicit **case** labels returns false.

```
CORBA::TCKind discriminator_kind();
```

The **discriminator_kind** operation returns the **TCKind** value of the discriminator's **TypeCode**.

```
CORBA::TCKind member_kind()
    raises(InvalidValue);
```

The **member_kind** operation returns the **TCKind** value of the currently active member's **TypeCode**. Calling this operation on a union that does not have a currently active member raises InvalidValue.

```
DynAny member()
    raises(InvalidValue);
```

The **member** operation returns the currently active member. If the union has no active member, the operation raises InvalidValue. Note that the returned reference remains valid only for as long as the currently active member does not change. Using the returned reference beyond the life time of the currently active member raises OBJECT_NOT_EXIST.

```
FieldName member_name()
    raises(InvalidValue);
```

The **member_name** operation returns the name of the currently active member. If the union's **TypeCode** does not contain a member name for the currently active member, the operation returns an empty string. Calling **member_name** on a union without an active member raises InvalidValue.

```
boolean is_set_to_default_member();
```

The **is_set_to_default_member** operation returns TRUE if a union has an explicit default label and the discriminator value does not match any of the union's other case labels.

13.2.7 The DynSequence Interface

DynSequence objects are associated with sequences.

```
typedef sequence<any> AnySeq;
typedef sequence<DynAny> DynAnySeq;

local interface DynSequence : DynAny {
    unsigned long get_length();
    void set_length(in unsigned long len)
        raises(InvalidValue);
    AnySeq get_elements();
    void set_elements(in AnySeq value)
        raises(TypeMismatch, InvalidValue);
    DynAnySeq get_elements_as_dyn_any();
    void set_elements_as_dyn_any(in DynAnySeq value)
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        raises(TypeMismatch, InvalidValue);
};
```

```
    unsigned long get_length();
```

The **get_length** operation returns the current length of the sequence.

```
    void set_length(in unsigned long len)
        raises(InvalidValue);
```

The **set_length** operation sets the length of the sequence. Increasing the length of a sequence adds new elements at the tail without affecting the values of already existing elements. Newly added elements are default-initialized.

Increasing the length of a sequence sets the current position to the first newly-added element if the previous current position was −1. Otherwise, if the previous current position was not −1, the current position is not affected.

Increasing the length of a bounded sequence to a value larger than the bound raises InvalidValue.

Decreasing the length of a sequence removes elements from the tail without affecting the value of those elements that remain. The new current position after decreasing the length of a sequence is determined as follows:

- If the length of the sequence is set to zero, the current position is set to −1.
- If the current position is −1 before decreasing the length, it remains at −1.
- If the current position indicates a valid element and that element is not removed when the length is decreased, the current position remains unaffected.
- If the current position indicates a valid element and that element is removed, the current position is set to −1.

```
    DynAnySeq get_elements();
```

The **get_elements** operation returns the elements of the sequence.

```
    void set_elements(in AnySeq value)
        raises(TypeMismatch, InvalidValue);
```

The **set_elements** operation sets the elements of a sequence. The length of the **DynSequence** is set to the length of **value**. The current position is set to zero if **value** has non-zero length and to −1 if **value** is a zero-length sequence.

If value contains one or more elements whose **TypeCode** is not equivalent to the element **TypeCode** of the **DynSequence**, the operation raises TypeMismatch. If the length of value exceeds the bound of a bounded sequence, the operation raises InvalidValue.

The **get_elements_as_dyn_any** and **set_elements_as_dyn_any** operations have the same semantics, but accept and return values of type **DynAny** instead of **Any**.

13.2.8 The DynArray Interface

DynArray objects are associated with arrays.

```
local interface DynArray : DynAny {
    AnySeq get_elements();
    void set_elements(in AnySeq value)
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        raises(TypeMismatch, InvalidValue);
    DynAnySeq get_elements_as_dyn_any();
    void set_elements_as_dyn_any(in DynAnySeq value)
        raises(TypeMismatch, InvalidValue);
};
```

```
    DynAnySeq get_elements();
```

The **get_elements** operation returns the elements of the **DynArray**.

```
    void set_elements(in DynAnySeq value)
        raises(TypeMismatch, InvalidValue);
```

The **set_elements** operation sets the **DynArray** to contain the passed elements. If the sequence does not contain the same number of elements as the array dimension, the operation raises InvalidValue. If one or more elements have a type that is inconsistent with the **DynArray**'s **TypeCode**, the operation raises TypeMismatch.

The **get_elements_as_dyn_any** and **set_elements_as_dyn_any** operations have the same semantics as their **Any** counterparts, but accept and return values of type **DynAny** instead of **Any**.

Note that the dimension of the array is contained in the **TypeCode**, which is accessible through the **type** attribute. It can also be obtained by calling the **component_count** operation.

13.2.9 The DynValueCommon Interface

DynValueCommon provides operations supported by both the **DynValue** and **DynValueBox** interfaces.

```
local interface DynValueCommon : DynAny {
    boolean is_null();
    void set_to_null();
    void set_to_value();
};
```

```
    boolean is_null();
```

The **is_null** operation returns **TRUE** if the **DynValueCommon** represents a null valuetype.

```
    void set_to_null();
```

The **set_to_null** operation changes the representation of a **DynValueCommon** to a null valuetype.

```
    void set_to_value();
```

If the **DynValueCommon** represents a null valuetype, then **set_to_value** replaces it with a newly constructed value, with its components initialized to default values as in **DynAnyFactory::create_dyn_any_from_type_code**. If the **DynValueCommon** represents a non-null valuetype, then this operation has no effect.

A reference to a **DynValueCommon** interface (and interfaces derived from it) exhibit the same sharing semantics as the underlying **valuetype** that it represents. This means that the relationships between **valuetypes** in a graph of valuetypes will remain unchanged when converted into **DynAny** form and vice versa. This is necessary to ensure that applications that use the **DII** and **DSI** can correctly view and preserve the semantics of the **valuetype** graph.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

13.2.10 The DynValue Interface

DynValue objects are associated with non-boxed valuetypes.

```
local interface DynValue : DynValueCommon {
    FieldName current_member_name()
        raises(TypeMismatch, InvalidValue);
    CORBA::TCKind current_member_kind()
        raises(TypeMismatch, InvalidValue);
    NameValuePairSeq get_members()
        raises(InvalidValue);
    void set_members(in NameValuePairSeq value)
        raises(TypeMismatch, InvalidValue);
    NameDynAnyPairSeq get_members_as_dyn_any()
        raises(InvalidValue);
    void set_members_as_dyn_any(in NameDynAnyPairSeq value)
        raises(TypeMismatch, InvalidValue);
};
```

The **DynValue** interface can represent both null and non-null valuetypes. For a **DynValue** representing a non-null valuetype, the **DynValue**'s components comprise the public and private members of the valuetype, including those inherited from concrete base valuetypes, in the order of definition. A **DynValue** representing a null valuetype has no components and a current position of **-1**.

The remaining operations on the **DynValue** interface generally have equivalent semantics to the same operations on **DynStruct**. When invoked on a **DynValue** representing a null valuetype, **get_members** and **get_members_as_dyn_any** raise InvalidValue. When invoked on a **DynValue** representing a null valuetype, **set_members** and **set_members_as_dyn_any** convert the **DynValue** to a non-null valuetype.

WARNING: Indiscriminately changing the contents of private valuetype members can cause the valuetype implementation to break by violating internal constraints. Access to private members is provided to support such activities as ORB bridging and debugging and should not be used to arbitrarily violate the encapsulation of the valuetype.

13.2.11 The DynValueBox Interface

DynValueBox objects are associated with boxed valuetypes.

```
local interface DynValueBox : DynValueCommon {
    any get_boxed_value()
        raises(InvalidValue);
    void set_boxed_value(in any boxed)
        raises(TypeMismatch, InvalidValue);
    DynAny get_boxed_value_as_dyn_any()
        raises(InvalidValue);
    void set_boxed_value_as_dyn_any(in DynAny boxed)
        raises(TypeMismatch);
};
```

The **DynValueBox** interface can represent both null and non-null valuetypes. For a **DynValueBox** representing a non-null valuetype, the **DynValueBox** has a single component of the boxed type. A **DynValueBox** representing a null valuetype has no components and a current position of **-1**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
any get_boxed_value()
    raises(InvalidValue);
```

The **get_boxed_value** operation returns the boxed value as an any. If the **DynBoxedValue** represents a null valuetype, the operation raises InvalidValue.

```
void set_boxed_value(in any boxed)
    raises(TypeMismatch, InvalidValue);
```

The **set_boxed_value** operation replaces the boxed value with the specified value. If the type of the passed Any is not equivalent to the boxed type, the operation raises TypeMismatch. If the passed **Any** does not contain a legal value, the operation raises InvalidValue. If the **DynBoxedValue** represents a **null valuetype**, it is converted to a non-null value.

The **get_boxed_value_as_dyn_any** and **set_boxed_value_as_dyn_any** have the same semantics as their any counterparts, but accept and return values of type **DynAny** instead of any.

13.3 Usage in C++ Language

13.3.1 Dynamic Creation of CORBA::Any values

13.3.1.1 Creating an any that contains a struct

Consider the following IDL definition:

```
// IDL
struct MyStruct {
    long member1;
    boolean member2;
};
```

The following example illustrates how a **CORBA::Any** value may be constructed on the fly containing a value of type **MyStruct**:

```
// C++
CORBA::ORB_var orb = ...;
DynamicAny::DynAnyFactory_var dafact
   = orb->resolve_initial_references("DynAnyFactory");
CORBA::StructMemberSeq mems(2);
CORBA::Any_var result;
CORBA::Long    value1 = 99;
CORBA::Boolean value2 = 1;
mems.length(2);
mems[0].name = CORBA::string_dup("member1");
mems[0].type = CORBA::TypeCode::_duplicate(CORBA::_tc_long);
mems[1].name = CORBA::string_dup("member2");
mems[1].type
   = CORBA::TypeCode::_duplicate(CORBA::_tc_boolean);
```


© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
CORBA::TypeCode_var new_tc = orb->create_struct_tc(
            “IDL:MyStruct:1.0”,
            “MyStruct”,
            mems
);

// Construct the DynStruct object. Values for members are
// the value1 and value2 variables

DynamicAny::DynAny_ptr dyn_any
   = dafact->create_dyn_any(new_tc);
DynamicAny::DynStruct_ptr dyn_struct
   = DynamicAny::DynStruct::_narrow(dyn_any);
CORBA::release(dyn_any);
dyn_struct->insert_long(value1);
dyn_struct->next();
dyn_struct->insert_boolean(value2);
result = dyn_struct->to_any();
dyn_struct->destroy();
CORBA::release(dyn_struct);
```

13.3.2 Dynamic Interpretation of CORBA::Any values

13.3.2.1 Filtering of events

Suppose there is a CORBA object that receives events and prints all those events, which correspond to a data structure containing a member called **is_urgent** whose value is true.

The following fragment of code corresponds to a method that determines if an event should be printed or not. Note that the program allows several struct events to be filtered with respect to some common member.

```
// C++

   CORBA::Boolean Tester::eval_filter(
   DynamicAny::DynAnyFactory_ptr dafact,
   const CORBA::Any & event
)
{
   CORBA::Boolean success = FALSE;
   DynamicAny::DynAny_var;
   try {
         // First, convert the event to a DynAny.
         // Then attempt to narrow it to a DynStruct.
         // The _narrow only returns a reference
         // if the event is a struct.
         dyn_var = dafact->create_dyn_any(event);
         DynamicAny::DynStruct_var dyn_struct
            = DynamicAny::DynStruct::_narrow(dyn_any);
         if (!CORBA::is_nil(dyn_struct)) {
            CORBA::Boolean found = FALSE;
            do {
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
                CORBA::String_var member_name
                   = dyn_struct->current_member_name();
                found = (strcmp(member_name, "is_urgent") == 0);
             } while (!found && dyn_struct->next());
             if (found) {
                // We only create a DynAny object for the member
                // we were looking for:
                DynamicAny::DynAny_var dyn_member
                   = dyn_struct->current_component();
                success = dyn_member->get_boolean();
             }
          }
   }
   catch(...) {};
   if (!CORBA::is_nil(dyn_var))
         dyn_var->destroy();
   return success;
}
```


© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14 The Interface Repository

14.1 Overview

The Interface Repository is the component of the ORB that provides persistent storage of interface definitions—it manages and provides access to a collection of object definitions specified in OMG IDL.

An ORB provides distributed access to a collection of objects using the objects' publicly defined interfaces specified in OMG IDL. The Interface Repository provides for the storage, distribution, and management of a collection of related objects' interface definitions.

For an ORB to correctly process requests, it must have access to the definitions of the objects it is handling. Object definitions can be made available to an ORB in one of two forms:

1. By incorporating the information procedurally into stub routines (e.g., as code that maps C language subroutines into communication protocols).
2. As objects accessed through the dynamically accessible Interface Repository (i.e., as interface objects accessed through OMG IDL-specified interfaces).

In particular, the ORB can use object definitions maintained in the Interface Repository to interpret and handle the values provided in a request to:

- Provide type-checking of request signatures (whether the request was issued through the DII or through a stub).
- Assist in checking the correctness of interface inheritance graphs.
- Assist in providing interoperability between different ORB implementations.

As the interface to the object definitions maintained in an Interface Repository is public, the information maintained in the Repository can also be used by clients and services. For example, the Repository can be used to:

- Manage the installation and distribution of interface definitions.
- Provide components of a CASE environment (for example, an interface browser).
- Provide interface information to language bindings (such as a compiler).
- Provide components of end-user environments (for example, a menu bar constructor).

The complete OMG IDL specification for the Interface Repository is in OMG IDL for Interface Repository on page 284; however, fragments of the specification are used throughout this clause as necessary.

14.2 Scope of an Interface Repository

Interface definitions are maintained in the Interface Repository as a set of objects that are accessible through a set of OMG IDL-specified interface definitions. An interface definition contains a description of the operations it supports, including the types of the parameters, exceptions it may raise, and context information it may use.

In addition, the interface repository stores constant values, which might be used in other interface and value definitions or might simply be defined for programmer convenience and it stores TypeCodes [TypeCodes on page 138], which are values that describe a type in structural terms.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The Interface Repository uses modules as a way to group interfaces and to navigate through those groups by name. Modules can contain constants, typedefs, exceptions, interface/ component/home definitions, and other modules. Modules may, for example, correspond to the organization of OMG IDL definitions. They may also be used to represent organizations defined for administration or other purposes.

The Interface Repository consists of a set of *interface repository objects* that represent the information in it. There are operations that operate on this apparent object structure. It is an implementation's choice whether these objects exist persistently or are created when referenced in an operation on the repository. There are also operations that extract information in an efficient form, obtaining a block of information that describes a whole interface or a whole operation.

An ORB may have access to multiple Interface Repositories. This may occur because

- two ORBs have different requirements for the implementation of the Interface Repository,
- an object implementation (such as an OODB) prefers to provide its own type information, or
- it is desired to have different additional information stored in different repositories.

The use of TypeCodes (TypeCodes on page 138) and repository identifiers is intended to allow different repositories to keep their information consistent.

As shown in Figure 14.1, the same interface **Doc** is installed in two different repositories, one at SoftCo, Inc., which sells Doc objects, and one at Customer, Inc., which buys Doc objects from SoftCo. SoftCo sets the repository id for the Doc interface when it defines it. Customer might first install the interface in its repository in a module where it could be tested before exposing it for general use. Because it has the same repository id, even though the Doc interface is stored in a different repository and is nested in a different module, it is known to be the same.

Meanwhile at SoftCo, someone working on a new Doc interface has given it a new repository id 456, which allows the ORBs to distinguish it from the current product Doc interface.

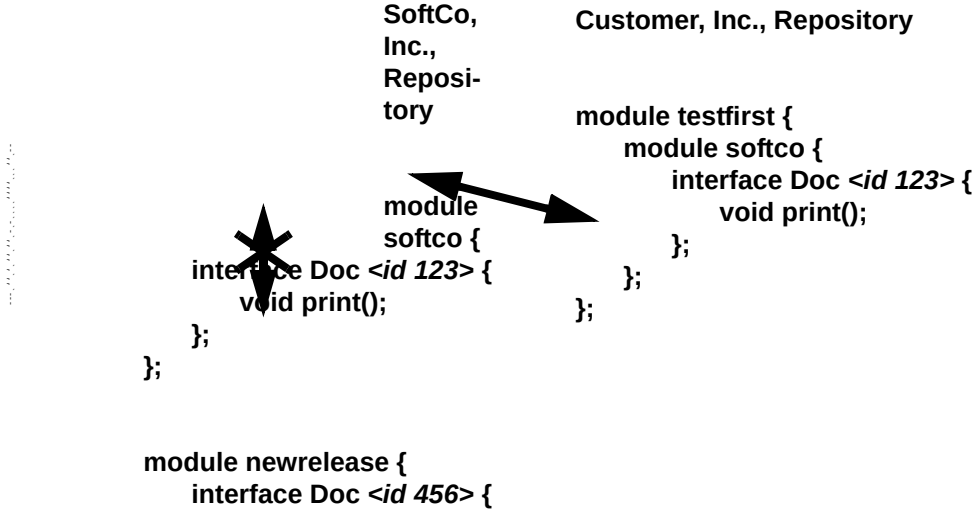


Figure 14.1 - Using Repository IDs to establish correspondence between repositories

Not all interfaces will be visible in all repositories. For example, Customer employees cannot see the new release of the Doc interface. However, widely used interfaces will generally be visible in most repositories.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This Interface Repository specification defines operations for retrieving information from the repository as well as creating definitions within it. There may be additional ways to insert information into the repository (for example, compiling OMG IDL definitions, copying objects from one repository to another).

A critical use of the interface repository information is for connecting ORBs together. When an object is passed in a request from one ORB to another, it may be necessary to create a new object to represent the passed object in the receiving ORB. This may require locating the interface information in an interface repository in the receiving ORB. By getting the repository id from a repository in the sending ORB, it is possible to look up the interface in a repository in the receiving ORB. To succeed, the interface for that object must be installed in both repositories with the same repository id.

14.3 Implementation Dependencies

An implementation of an Interface Repository requires some form of persistent object store. Normally the kind of persistent object store used determines how interface definitions are distributed and/or replicated throughout a network domain. For example, if an Interface Repository is implemented using a filing system to provide object storage, there may be only a single copy of a set of interfaces maintained on a single machine. Alternatively, if an OODB is used to provide object storage, multiple copies of interface definitions may be maintained each of which is distributed across several machines to provide both high-availability and load-balancing.

The kind of object store used may determine the scope of interface definitions provided by an implementation of the Interface Repository. For example, it may determine whether each user has a local copy of a set of interfaces or if there is one copy per community of users. The object store may also determine whether or not all clients of an interface set see exactly the same set at any given point in time or whether latency in distributing copies of the set gives different users different views of the set at any point in time.

An implementation of the Interface Repository is also dependent on the security mechanism in use. The security mechanism (usually operating in conjunction with the object store) determines the nature and granularity of access controls available to constrain access to objects in the repository.

14.3.1 Managing Interface Repositories

Interface Repositories contain the information necessary to allow programs to determine and manipulate the type information at run-time. Programs may attempt to access the interface repository at any time by using the **get_interface** operation on the object reference. Once information has been installed in the repository, programs, stubs, and objects may depend on it. Updates to the repository must be done with care to avoid disrupting the environment. A variety of techniques are available to help do so.

A coherent repository is one whose contents can be expressed as a valid collection of OMG IDL definitions. For example, all inherited interfaces exist, there are no duplicate operation names or other name collisions, all parameters have known types, and so forth. As information is added to the repository, it is possible that it may pass through incoherent states. Media failures or communication errors might also cause it to appear incoherent. In general, such problems cannot be completely eliminated.

Replication is one technique to increase the availability and performance of a shared database. It is likely that the same interface information will be stored in multiple repositories in a computing environment. Using repository IDs, the repositories can establish the identity of the interfaces and other information across the repositories.

Multiple repositories might also be used to insulate production environments from development activity. Developers might be permitted to make arbitrary updates to their repositories, but administrators may control updates to widely used repositories. Some repository implementations might permit sharing of information, for example, several developers'

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

repositories may refer to parts of a shared repository. Other repository implementations might instead copy the common information. In any case, the result should be a repository facility that creates the impression of a single, coherent repository.

The interface repository itself cannot make all repositories have coherent information, and it may be possible to enter information that does not make sense. The repository will report errors that it detects (e.g., defining two attributes with the same name) but might not report all errors, for example, adding an attribute to a base interface may or may not detect a name conflict with a derived interface. Despite these limitations, the expectation is that a combination of conventions, administrative controls, and tools that add information to the repository will work to create a coherent view of the repository information.

Transactions and concurrency control mechanisms defined by the Object Services may be used by some repositories when updating the repository. Those services are designed so that they can be used without changing the operations that update the repository. For example, a repository that supports the Transaction Service would inherit the Repository interface, which contains the update operations, as well as the Transaction interface, which contains the transaction management operations. (For more information about Object Services, including the Transaction and Concurrency Control Services, refer to the individual CORBA Services specifications.)

Often, rather than change the information, new versions will be created, allowing the old version to continue to be valid. The new versions will have distinct repository IDs and be completely different types as far as the repository and the ORBs are concerned. The IR provides storage for version identifiers for named types, but does not specify any additional versioning mechanism or semantics.

14.4 Basics

This sub clause introduces some basic ideas that are important to understanding the Interface Repository. Topics addressed in this sub clause are:

- Names and Identifiers
- Types and TypeCodes
- Interface Repository Objects
- Structure and Navigation of the Interface Repository

14.4.1 Names and Identifiers

Simple names are not necessarily unique within an Interface Repository; they are always relative to an explicit or implicit module. In this context, interface, struct, union, exception, and value type definitions are considered implicit modules.

Scoped names uniquely identify modules, interfaces, components, homes, value and event types, value members, value boxes, constant, typedefs, exceptions, attributes, and operations in an Interface Repository.

Repository identifiers globally identify modules, interfaces, components, homes, value and event types, value members, value boxes, constants, typedefs, exceptions, attributes, and operations. They can be used to synchronize definitions across multiple ORBs and Repositories.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.4.2 Types and TypeCodes

The Interface Repository stores information about types that are not interfaces in a data value called a TypeCode. From the TypeCode alone it is possible to determine the complete structure of a type. See TypeCodes on page 138 for more information on the internal structure of TypeCodes.

14.4.3 Interface Repository Objects

Information about the entities that are managed in an Interface Repository is maintained as a collection of *interface repository objects* of the following types:

- **Repository**: the top-level module for the repository name space; it contains constants, typedefs, exceptions, interface, component, home, value or event type definitions, and modules.
- **ModuleDef**: a logical grouping of interfaces and value types; it contains constants, typedefs, exceptions, interface, component, home, value or event type definitions, and other modules.
- **InterfaceDef**: an interface definition; it contains lists of constants, types, exceptions, operations, and attributes.
- **ExtInterfaceDef**: an extended version of **InterfaceDef** that is capable of accommodating attributes with exceptions.
- **AbstractInterfaceDef**: an abstract interface definition; it contains lists of constants, types, exceptions, operations, and attributes.
- **ExtAbstractInterfaceDef**: an extended version of **AbstractInterfaceDef** that is capable of accommodating attributes with exceptions.
- **LocalInterfaceDef**: a local interface definition; it contains lists of constants, types, exceptions, operations, and attributes.
- **ExtLocalInterfaceDef**: an extended version of **LocalInterfaceDef** that is capable of accommodating attributes with exceptions.
- **ValueDef**: a value type definition that contains lists of constants, types, exceptions, operations, attributes, and members
- **ExtValueDef**: an extended version of **ValueDef** that is capable of accommodating attributes and initializers with exceptions.
- **EventDef**: an event type definition that contains lists of constants, types, exceptions, operations, attributes, and members.
- **ValueBoxDef**: the definition of a boxed value type.
- **ValueMemberDef**: the definition of a member of the value type.
- **AttributeDef**: the definition of an attribute of the interface or value type.
- **ExtAttributeDef**: an extended version of **AttributeDef** that is capable of accommodating attributes with exceptions.
- **OperationDef**: the definition of an operation of the interface, value or event type; it contains lists of parameters and exceptions raised by this operation.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- **TypedefDef**: base interface for definitions of named types that are not interfaces components, homes, or value and event types.
- **ConstantDef**: the definition of a named constant.
- **ExceptionDef**: the definition of an exception that can be raised by an operation.
- **ComponentDef**: a component definition; it contains lists of provides, uses, consumes, publishes, supports, emits, and attributes.
- **HomeDef**: a home definition; it contains lists of constants, types, exceptions, operations, attributes, factories and finders.
- **FactoryDef**: the definition of a factory; it is an operation that is specifically used for creating new instances of components in a home.
- **FinderDef**: the definition of a finder; it is an operation that is specifically used to find components within a home.
- **ProvidesDef**: the definition of an interface that is provided by a component.
- **UsesDef**: the definition of an interface that is used by a component.
- **EmitsDef**: the definition of events that are emitted by a component.
- **PublishesDef**: the definition of events that are published by a component.
- **ConsumesDef**: the definition of events that are consumed by a component.

The interface specifications for each *interface repository object* lists the attributes maintained by that object (see Interface Repository Interfaces on page 228). Many of these attributes correspond directly to OMG IDL statements. An implementation can choose to maintain additional attributes to facilitate managing the Repository or to record additional (proprietary) information about an interface. Implementations that extend the IR interfaces shall do so by deriving new interfaces, not by modifying the standard interfaces.

The *CORBA* specification defines a minimal set of operations for *interface repository objects*. Additional operations that an implementation of the Interface Repository may provide could include operations that provide for the versioning of entities and for the reverse compilation of specifications (i.e., the generation of a file containing an object's OMG IDL specification).

14.4.4 Structure and Navigation of the Interface Repository

The definitions in the Interface Repository are structured as a set of *interface repository objects*. These objects are structured the same way definitions are structured—some objects (definitions) "contain" other objects.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The containment relationships for the *interface repository objects* types in the Interface Repository are shown in Figure 14.2

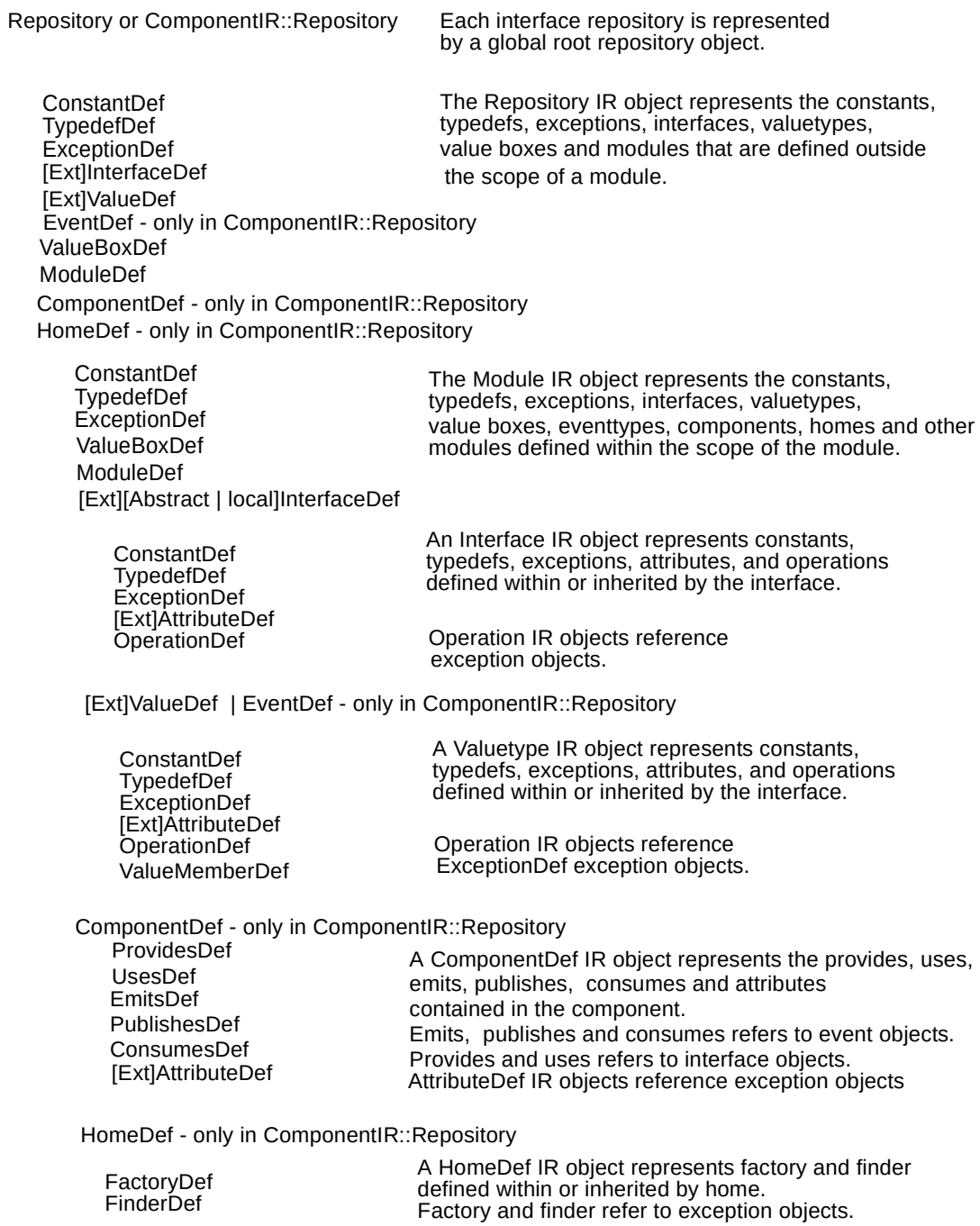


Figure 14.2 - Interface Repository Object Containment

There are three ways to locate an interface in the Interface Repository, by:

1. Obtaining an **InterfaceDef** object directly from the ORB.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

2. Navigating through the module name space using a sequence of names.
3. Locating the **InterfaceDef** object that corresponds to a particular repository identifier.

There are four ways to locate a component in the Interface Repository, by:

1. Obtaining an **ComponentDef** object directly from the ORB.
2. Navigating through the module name space using a sequence of names.
3. Locating the **ComponentDef** object that corresponds to a particular repository identifier.
4. Obtaining the **ComponentDef** from the **HomeDef** object corresponding to its home.

There are three ways to locate a home in the Interface Repository, by:

1. Obtaining a **HomeDef** object directly from the ORB.
2. Navigating through the module name space using a sequence of names.
3. Locating the **HomeDef** object that corresponds to a particular repository identifier.

NOTE: It should be noted that given a **ComponentDef** IR object, it is not possible to obtain the **HomeDef** IR object for the home that manages this component, since there could be multiple such homes, and the actual relation of a specific component to a specific home is available only at runtime. To get to the **HomeDef** object corresponding to the home of a given component, one needs to do a **CCMObject::get_home**, and then do a **CCMHome::get_home_def** on the home thus obtained.

Obtaining an **InterfaceDef** object directly is useful when an object is encountered whose type was not known at compile time. By using the **get_interface** operation on the object reference, it is possible to retrieve the Interface Repository information about the object. That information could then be used to perform operations on the object. Similarly, by using the **CCMObject::get_component_def** operation, it is possible to retrieve the Component Repository information about a component.

Navigating the module name space is useful when information about a particular named interface is desired. Starting at the root module of the repository, it is possible to obtain entries by name.

Locating the **InterfaceDef** object by ID is useful when looking for an entry in one repository that corresponds to another. A repository identifier must be globally unique. By using the same identifier in two repositories, it is possible to obtain the interface identifier for an interface in one repository, and then obtain information about that interface from another repository that may be closer or contain additional information about the interface.

Analogous operations are provided for manipulating value and event types.

The **ComponentIR** module contains the IR Objects that were added to reflect new IDL constructs that were added to support Components. These are built upon the IR interfaces defined in **CORBA** module including **ExtInterfaceDef**, **ExtValueDef**, and **ExtAttributeDef** and thus are backward compatible extensions of the 2.5 and earlier versions of the IR.

14.5 Interface Repository Interfaces

Several interfaces are used as *base interfaces* for objects in the IR. These *base interface*s are not instantiable.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A common set of operations is used to locate objects within the Interface Repository. These operations are defined in the interfaces **IRObject**, **Container**, and **Contained** described below. All IR objects inherit from the **IRObject** interface, which provides an operation for identifying the actual type of the object. Objects that are containers inherit navigation operations from the **Container** interface. Objects that are contained by other objects inherit navigation operations from the **Contained** interface.

The **IDLType** interface is inherited by all IR objects that represent IDL types, including interfaces, typedefs, and anonymous types. The **TypedefDef** interface is inherited by all named non-interface types.

The *base interfaces* **IRObject**, **Contained**, **Container**, **IDLType**, **TypedefDef ComponentIR::Container** and **ComponentIR::EventPortDef** are not instantiable.

All string data in the Interface Repository are encoded as defined by the ISO 8859-1 coded character set.

Interface Repository operations indicate error conditions using the system exceptions BAD_PARAM and BAD_INV_ORDER with specific minor codes. The specific operations that raise these exceptions are documented in the description of the operations. For a description of how these minor codes are encoded in the **ex_body** of standard exceptions see System Exceptions on page 148 and Standard Minor Exception Codes on page 156. The exceptions and minor codes that are used by Interface Repository interfaces are as follows:

Exception	Minor Code	Explanation
BAD_PARAM	2	RID is already defined in IFR
	3	Name already used in the context in IFR
	4	Target is not a valid container
	5	Name clash in inherited context
	31	Attempt to define a oneway operation with non-void result, out or inout parameters or user exceptions.
BAD_INV_ORDER	1	Dependency exists in IFR preventing destruction of this object
	2	Attempt to destroy indestructible objects in IFR

14.5.1 Supporting Type Definitions

Several types are used throughout the IR interface definitions.

```
module CORBA {
    typedef string          Identifier;
    typedef string          ScopedName;
    typedef string          RepositoryId;

    enum DefinitionKind {
        dk_none, dk_all,
        dk_Attribute, dk_Constant, dk_Exception, dk_Interface,
        dk_Module, dk_Operation, dk_Typedef,
        dk_Alias, dk_Struct, dk_Union, dk_Enum,
        dk_Primitive, dk_String, dk_Sequence, dk_Array,
        dk_Repository,
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        dk_Wstring, dk_Fixed,
        dk_Value, dk_ValueBox, dk_ValueMember,
        dk_Native,
        dk_AbstractInterface,
        dk_LocalInterface
        dk_Component, dk_Home,
        dk_Factory, dk_Finder,
        dk_Emits, dk_Publishes, dk_Consumes,
        dk_Provides, dk_Uses,
        dk_Event
    };
};
```

Identifiers are the simple names that identify modules, interfaces, components, homes, value and event types, value members, value boxes, constants, typedefs, exceptions, attributes, operations, ports, and native types. They correspond exactly to OMG IDL identifiers. An **Identifier** is not necessarily unique within an entire Interface Repository; it is unique only within a particular **Repository, ModuleDef**, **InterfaceDef**, **ComponentDef**, **HomeDef**, **ValueDef** , **EventDef**, **OperationDef, FactoryDef**, or **FinderDef**.

A **ScopedName** is a name made up of one or more **Identifiers** separated by the characters "::". They correspond to OMG IDL scoped names.

An *absolute* **ScopedName** is one that begins with "::" and unambiguously identifies a definition in a **Repository**. An *absolute* **ScopedName** in a **Repository** corresponds to a *global name* in an OMG IDL file. A *relative* **ScopedName** does not begin with "**::**" and must be resolved relative to some context.

A **RepositoryId** is an identifier used to uniquely and globally identify a module, interface, component, home, value type, event type, value member, value box, native type, constant, typedef, exception, attribute, or operation. As **RepositoryId**s are defined as strings, they can be manipulated (e.g., copied and compared) using a language binding's string manipulation routines.

A **DefinitionKind** identifies the type of an IR object.

14.5.2 IRObject

The *base interface* **IRObject** represents the most generic interface from which all other Interface Repository interfaces are derived, even the Repository itself.

```
module CORBA {
    interface IRObject {
        // read interface
        readonly attribute DefinitionKind def_kind;
        // write interface
        void destroy ();
    };
};
```

14.5.2.1 Read Interface

The **def_kind type_name** attribute identifies the type of the definition.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.5.2.2 Write Interface

The **destroy** operation causes the object to cease to exist. If the object is a **Container**, **destroy** is applied to all its contents. If the object contains an **IDLType** attribute for an anonymous type, that **IDLType** is destroyed. If the object is currently contained in some other object, it is removed. If **destroy** is invoked on a **Repository** or on a **PrimitiveDef**, then the BAD_INV_ORDER exception is raised with minor value 2. Implementations may vary in their handling of references to an object that is being destroyed, but the Repository should not be left in an incoherent state. Attempt to destroy an object that would leave the repository in an incoherent state shall cause BAD_INV_ORDER exception to be raised with the minor code 1.

14.5.3 Contained

The *base interface* **Contained** is inherited by all Interface Repository interfaces that are contained by other IR objects. All objects within the Interface Repository, except the root object (**Repository**) and definitions of anonymous (**ArrayDef**, **StringDef, WstringDef, FixedDef, and SequenceDef**), and primitive types are contained by other objects.

```
module CORBA {
    typedef string VersionSpec;

    interface Contained : IRObject {
        // read/write interface

        attribute RepositoryId          id;
        attribute Identifier            name;
        attribute VersionSpec           version;

        // read interface

        readonly attribute Container    defined_in;
        readonly attribute ScopedName   absolute_name;
        readonly attribute Repository   containing_repository;

        struct Description {
            DefinitionKind      kind;
            any                 value;
        };

        Description describe ();

        // write interface

        void move (
            in Container        new_container,
            in Identifier       new_name,
            in VersionSpec      new_version
        );
    };
};
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.5.3.1 Read Interface

An object that is contained by another object has an **id** attribute that identifies it globally, and a **name** attribute that identifies it uniquely within the enclosing **Container** object. It also has a **version** attribute that distinguishes it from other versioned objects with the same **name**. IRs are not required to support simultaneous containment of multiple versions of the same named object. Supporting multiple versions will require mechanisms and policy not specified in this document.

Contained objects also have a **defined_in** attribute that identifies the **Container** within which they are defined. Objects can be contained either because they are defined within the containing object (for example, an interface is defined within a module) or because they are inherited by the containing object (for example, an operation may be contained by an interface because the interface inherits the operation from another interface). If an object is contained through inheritance, the **defined_in** attribute identifies the **InterfaceDef** or **ValueDef** from which the object is inherited.

The **absolute_name** attribute is an absolute **ScopedName** that identifies a **Contained** object uniquely within its enclosing **Repository**. If this object's **defined_in** attribute references a **Repository**, the **absolute_name** is formed by concatenating the string "::" and this object's **name** attribute. Otherwise, the **absolute_name** is formed by concatenating the **absolute_name** attribute of the object referenced by this object's **defined_in** attribute, the string "**::**", and this object's **name** attribute.

The **containing_repository** attribute identifies the **Repository** that is eventually reached by recursively following the object's **defined_in** attribute.

The **within** operation returns the list of objects that contain the object. If the object is an interface or module it can be contained only by the object that defines it. Other objects can be contained by the objects that define them and by the objects that inherit them.

The **describe** operation returns a structure containing information about the interface. The description structure associated with each interface is provided below with the interface's definition. The kind of definition described by name of the structure returned is provided with the returned structure. The **kind** field of the returned **Description** struct shall give the **DefinitionKind** for the most derived type of the object. For example, if the **describe** operation is invoked on an attribute object, the **kind** field contains **dk_Attribute name** field contains "AttributeDescription" and the **value** field contains an **any**, which contains the **AttributeDescription** structure. The **kind** field in this must contain **dk_attribute** and not the kind of any **IRObject** from which the **attribute** object is derived. For example returning **dk_all** would be an error.

14.5.3.2 Write Interface

Setting the **id** attribute changes the global identity of this definition. A BAD_PARAM exception is raised with minor code 2 if an object with the specified **id** attribute already exists within this object's **Repository**.

Setting the **name** attribute changes the identity of this definition within its **Container**. A BAD_PARAM exception is raised with minor code 1 if an object with the specified **name** attribute already exists within this object's **Container**. The **absolute_name** attribute is also updated, along with any other attributes that reflect the name of the object. If this object is a **Container**, the **absolute_name** attribute of any objects it contains are also updated.

The **move** operation atomically removes this object from its current **Container**, and adds it to the **Container** specified by **new_container** must satisfy the following conditions:

- It must be in the same **Repository.** If it is not, then BAD_PARAM exception is raised with minor code 4.
- It must be capable of containing this object's type (see Structure and Navigation of the Interface Repository on page 226). If it is not, then BAD_PARAM exception is raised with minor code 4.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- It must not already contain an object with this object's name (unless multiple versions are supported by the IR). If this condition is not satisfied, then BAD_PARAM exception is raised with minor code 3.

The **name** attribute is changed to **new_name**, and the **version** attribute is changed to **new_version**.

The **defined_in** and **absolute_name** attributes are updated to reflect the new container and **name**. If this object is also a **Container**, the **absolute_name** attributes of any objects it contains are also updated.

14.5.4 Container

The *base interface* **Container** is used to form a containment hierarchy in the Interface Repository. A **Container** can contain any number of objects derived from the **Contained** interface. All **Container**s, except for **Repository**, are also derived from **Contained**.

```
module CORBA {
    typedef sequence <Contained> ContainedSeq;

    interface Container : IRObject {
        // read interface

        Contained lookup (in ScopedName search_name);

        ContainedSeq contents (
            in DefinitionKind       limit_type,
            in boolean              exclude_inherited
        );

        ContainedSeq lookup_name (
            in Identifier           search_name,
            in long                 levels_to_search,
            in DefinitionKind       limit_type,
            in boolean              exclude_inherited
        );

        struct Description {
            Contained           contained_object;
            DefinitionKind      kind;
            any                 value;
        };

        typedef sequence<Description> DescriptionSeq;

        DescriptionSeq describe_contents (
            in DefinitionKind       limit_type,
            in boolean              exclude_inherited,
            in long                 max_returned_objs
        );

        // write interface
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
ModuleDef create_module (
    in RepositoryId      id,
    in Identifier        name,
    in VersionSpec       version
);

ConstantDef create_constant (
    in RepositoryId      id,
    in Identifier        name,
    in VersionSpec       version,
    in IDLType           type,
    in any               value
);

StructDef create_struct (
    in RepositoryId      id,
    in Identifier        name,
    in VersionSpec       version,
    in StructMemberSeq   members
);

UnionDef create_union (
    in RepositoryId      id,
    in Identifier        name,
    in VersionSpec       version,
    in IDLType           discriminator_type,
    in UnionMemberSeq    members
);

EnumDef create_enum (
    in RepositoryId      id,
    in Identifier        name,
    in VersionSpec       version,
    in EnumMemberSeq     members
);

AliasDef create_alias (
    in RepositoryId      id,
    in Identifier        name,
    in VersionSpec       version,
    in IDLType           original_type
);

InterfaceDef create_interface (
    in RepositoryId      id,
    in Identifier        name,
    in VersionSpec       version,
    in InterfaceDefSeq   base_interfaces,
);
```


© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
ExceptionDef create_exception(
    in RepositoryId      id,
    in Identifier        name,
    in VersionSpec       version,
    in StructMemberSeq   members
);

ValueDef create_value(
    in RepositoryId      id,
    in Identifier        name,
    in VersionSpec       version,
    in boolean           is_custom,
    in boolean           is_abstract,
    in ValueDef          base_value,
    in boolean           is_truncatable,
    in ValueDefSeq       abstract_base_values,
    in InterfaceDefSeq   supported_interfaces,
    in InitializerSeq    initializers
);

ValueBoxDef create_value_box(
    in RepositoryId      id,
    in Identifier        name,
    in VersionSpec       version,
    in IDLType           original_type_def
);

NativeDef create_native(
    in RepositoryId      id,
    in Identifier        name,
    in VersionSpec       version
);

AbstractInterfaceDef create_abstract_interface(
    in RepositoryId id,
    in Identifier name,
    in VersionSpec version,
    in AbstractInterfaceDefSeq base_interfaces,
);

LocalInterfaceDef create_local_interface(
    in RepositoryId id,
    in Identifier name,
    in VersionSpec version,
    in InterfaceDefSeq base_interfaces
);

ExtValueDef create_ext_value (
    in RepositoryId      id,
    in Identifier        name,
    in VersionSpec       version,
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            in boolean          is_custom,
            in boolean          is_abstract,
            in ValueDef         base_value,
            in boolean          is_truncatable,
            in ValueDefSeq      abstract_base_values,
            in InterfaceDefSeq  supported_interfaces,
            in ExtInitializerSeq initializers
        );
    };
};
```

14.5.4.1 Read Interface

The **lookup** operation locates a definition relative to this container given a scoped name using OMG IDL's name scoping rules. An absolute scoped name (beginning with "::") locates the definition relative to the enclosing **Repository**. If no object is found, a nil object reference is returned.

The **contents** operation returns the list of objects directly contained by or inherited into the object. The operation is used to navigate through the hierarchy of objects. Starting with the Repository object, a client uses this operation to list all of the objects contained by the Repository, all of the objects contained by the modules within the Repository, and then all of the interfaces and value types within a specific module, and so on.

limit_type	If limit_type is set to dk_all "all," objects of all interface types are returned. For example, if this is an InterfaceDef, the attribute, operation, and exception objects are all returned. If limit_type is set to a specific interface, only objects of that interface type are returned. For example, only attribute objects are returned if limit_type is set to dk_Attribute "AttributeDef."
exclude_inherited	If set to TRUE, inherited objects (if there are any) are not returned. If set to FALSE, all contained objects—whether contained due to inheritance or because they were defined within the object—are returned.

The lookup_name operation is used to locate an object by name within a particular object or within the objects contained by that object. Use of values of levels_to_search of 0 or of negative numbers other than -1 is undefined.

search_name	Specifies which name is to be searched for.
levels_to_search	Controls whether the lookup is constrained to the object the operation is invoked on or whether it should search through objects contained by the object as well.

Setting levels_to_search to -1 searches the current object and all contained objects. Setting levels_to_search to 1 searches only the current object. Use of values of levels_to_search of 0 or of negative numbers other than -1 is undefined.

The describe_contents operation combines the contents operation and the describe operation. For each object returned by the contents operation, the description of the object is returned (i.e., the object's describe operation is invoked and the results returned).

max_returned_objs	Limits the number of objects that can be returned in an invocation of the call to the number provided. Setting the parameter to -1 means return all contained objects.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

contents and **describe_contents** return a list of elements in their original order (i.e., the order in which the elements were created in or moved into the container). If **exclude_inherited** is false, the ordering of inherited elements is undefined.

14.5.4.2 Write Interface

The **Container** interface provides operations to create **ModuleDef**s, **ConstantDef**s, **StructDef**s, **UnionDef**s, **EnumDef**s, **AliasDef**s, **InterfaceDef**s, **ValueDef**s **ValueBoxDef**s, and **NativeDef**s as contained objects. The **defined_in** attribute of a definition created with any of these operations is initialized to identify the **Container** on which the operation is invoked, and the **containing_repository** attribute is initialized to its **Repository**.

The **create_<type>** operations all take **id** and **name** parameters that are used to initialize the identity of the created definition. A BAD_PARAM exception is raised with minor code 2 if an object with the specified **id** already exists in the **Repository**. A BAD_PARAM exception with minor code 3 is raised if the specified **name** already exists within this **Container** and multiple versions are not supported. Certain interfaces derived from **Container** may restrict the types of definitions that they may contain. Any **create_<type>** operation that would insert a definition that is not allowed by a Container will raise the BAD_PARAM exception with minor code 4.

The **create_module** operation returns a new empty **ModuleDef**. Definitions can be added using **Container::create_<type>** operations on the new module, or by using the **Contained::move** operation.

The **create_constant** operation returns a new **ConstantDef** with the specified **type** and **value**.

The **create_struct** operation returns a new **StructDef** with the specified **members**. The **type** member of the **StructMember** structures is ignored, and should be set to **TC_void**. See StructDef on page 241 for more information.

The **create_union** operation returns a new **UnionDef** with the specified **discriminator_type** and **members**. The **type** member of the **UnionMember** structures is ignored, and should be set to **TC_void**. See UnionDef on page 242 for more information.

The **create_enum** operation returns a new **EnumDef** with the specified **members**. See EnumDef on page 243 for more information.

The **create_alias** operation returns a new **AliasDef** with the specified **original_type**.

The **create_interface** operation returns a new empty **ExtInterfaceDef** with the specified **base_interfaces**. Type, exception, and constant definitions can be added using **Container::create_<type>** operations on the new **InterfaceDef**. **OperationDefs** can be added using **InterfaceDef::create_operation** and **AttributeDefs** can be added using **InterfaceDef::create_attribute**. Definitions can also be added using the **Contained::move** operation.

The **create_abstract_interface** operation returns a new empty **ExtAbstractInterfaceDef** with the specified **base_interfaces**. Type, exception, and constant definitions can be added using **Container::create_<type>** operations on the new **AbstractInterfaceDef**. **OperationDef**s can be added using **AbstractInterfaceDef::create_operation** and **AttributeDef**s can be added using **AbstractInterfaceDef::create_attribute**. Definitions can also be added using the **Contained::move** operation.

The **create_local_interface** operation returns a new empty **ExtLocalInterfaceDef** with the specified **base_interfaces**. Type, exception, and constant definitions can be added using **Container::create_<type>** operations on the new **LocalInterfaceDef**. **OperationDef**s can be added using **LocalInterfaceDef::create_operation** and **AttributeDef**s can be added using **LocalInterfaceDef::create_attribute**. Definitions can also be added using the **Contained::move** operation.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The **create_value** operation returns a new empty **ValueDef** with the specified base interfaces and values (**base_value**, **supported_interfaces**, and **abstract_base_values**) as well as the other information describing the new values characteristics (**is_custom**, **is_abstract**, **is_truncatable**, and **initializers**). Type, exception, and constant definitions can be added using **Container::create_<type>** operations on the new **ValueDef**. **OperationDefs** can be added using **ValueDef::create_operation** and **AttributeDefs** can be added using **ValueDef::create_attribute**. Definitions can also be added using the **Contained::move** operation.

The **create_value_box** operation returns a new **ValueBoxDef** with the specified **original_type_def**.

The **create_exception** operation returns a new **ExceptionDef** with the specified members. The **type** member of the **StructMember** structures should be set to **TC_void**.

The **create_native** operation returns a new **NativeDef** with the specified **name**.

The **create_ext_value** operation returns a new empty **ExtValueDef** with the specified base interfaces and values (**base_value**, **supported_interfaces**, and **abstract_base_values**) as well as the other information describing the new values characteristics (**is_custom**, **is_abstract**, **is_truncatable**, and **initializers**). The **initializers** argument is of type **ExtInitializerSeq** allowing one to specify user exceptions for initializers. Type, exception, and constant definitions can be added using **Container::create_<type>** operations on the new **ExtValueDef**. **OperationDef**s can be added using **ExtValueDef::create_operation** and **ExtAttributeDef**s can be added using **ExtValueDef::create_ext_attribute**. Definitions can also be added using the **Contained::move** operation.

14.5.5 IDLType

The *base interface* **IDLType** is inherited by all IR objects that represent OMG IDL types. It provides access to the **TypeCode** describing the type, and is used in defining other interfaces wherever definitions of IDL types must be referenced.

```
module CORBA {
    interface IDLType : IRObject {
        readonly attribute TypeCode type;
    };
};
```

The **type** attribute describes the type defined by an object derived from **IDLType**.

14.5.6 Repository

Repository is an interface that provides global access to the Interface Repository that does not support access to information related to CORBA Components. The **Repository** object can contain constants, typedefs, exceptions, interfaces, value types, value boxes, native types, and modules. As it inherits from **Container**, it can be used to look up any definition (whether globally defined or defined within a module or interface) either by **name** or by **id**.

Since **Repository** derives only from **Container** and not from **Contained**, it does not have a **RepositoryId** associated with it. By default it is deemed to have the **RepositoryId** "" (the empty string) for purposes of assigning a value to the **defined_in** field of the **description** structure of **ModuleDef**, **InterfaceDef**, **ValueDef**, **ValueBoxDef, TypedefDef**, **ExceptionDef**, and **ConstantDef** that are contained immediately in the Repository object.

There may be more than one Interface Repository in a particular ORB environment (although some ORBs might require that definitions they use be registered with a particular repository). Each ORB environment will provide a means for obtaining object references to the Repositories available within the environment.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
module CORBA {
    interface Repository : Container {
        // read interface

        Contained lookup_id (in RepositoryId search_id);

        TypeCode get_canonical_typecode(in TypeCode tc);

        PrimitiveDef get_primitive (in PrimitiveKind kind);

        // write interface

        StringDef create_string (in unsigned long bound);

        WstringDef create_wstring(in unsigned long bound);

        SequenceDef create_sequence (
            in unsigned long   bound,
            in IDLType         element_type
        );

        ArrayDef create_array (
            in unsigned long   length,
            in IDLType         element_type
        );

        FixedDef create_fixed(
            in unsigned short digits,
            in short scale
        );
    };
};
```

14.5.6.1 Read Interface

The **lookup_id** operation is used to lookup an object in a **Repository** given its **RepositoryId**. If the **Repository** does not contain a definition for **search_id**, a nil object reference is returned. The **lookup_id** operations always return a nil reference if the value of **search_id** is **IDL:omg.org/CORBA/Object:1.0**, or **IDL:omg.org/CORBA/ValueBase:1.0**, signifying the fact that the implicit base types are not contained in the Interface Repository.

The **get_canonical_typecode** operation looks up the **TypeCode** in the Interface Repository and returns an equivalent **TypeCode** that includes all **repository ids**, **names**, and **member_names**. If the top level **TypeCode** does not contain a **RepositoryId**, such as array and sequence **TypeCodes**, or **TypeCodes** from older ORBs, or if it contains a **RepositoryId** that is not found in the target **Repository**, then a new **TypeCode** is constructed by recursively calling **get_canonical_typecode** on each member **TypeCode** of the original **TypeCode**.

The **get_primitive** operation returns a reference to a **PrimitiveDef** (see PrimitiveDef on page 244) with the specified **kind** attribute. All **PrimitiveDef**s are immutable and are owned by the **Repository**.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.5.6.2 Write Interface

The five **create_<type>** operations that create new IR objects defining anonymous types. As these interfaces are not derived from **Contained**, it is the caller's responsibility to invoke **destroy** on the returned object if it is not successfully used in creating a definition that is derived from **Contained**. Each anonymous type definition must be used in defining exactly one other object.

1. The **create_string** operation returns a new **StringDef** with the specified **bound**, which must be non-zero. The **get_primitive** operation is used for unbounded strings.
2. The **create_wstring** operation returns a new **WstringDef** with the specified **bound**, which must be non-zero. The **get_primitive** operation is used for unbounded strings.
3. The **create_sequence** operation returns a new **SequenceDef** with the specified **bound** and **element_type**.
4. The **create_array** operation returns a new **ArrayDef** with the specified **length** and **element_type**.
5. The **create_fixed** operation returns a new **FixedDef** with the specified number of digits and scale. The number of digits must be from 1 to 31, inclusive.

14.5.7 ModuleDef

A **ModuleDef** can contain constants, typedefs, exceptions, interfaces, value types, value boxes, native types, and other module objects.

```
module CORBA {
    interface ModuleDef : Container, Contained {};

    struct ModuleDescription {
        Identifier      name;
        RepositoryId    id;
        RepositoryId    defined_in;
        VersionSpec     version;
    };
};
```

The inherited **describe** operation for a **ModuleDef** object returns a **ModuleDescription**.

14.5.8 ConstantDef

A **ConstantDef** object defines a named constant.

```
module CORBA {
    interface ConstantDef : Contained {
        readonly attribute TypeCode   type;
        attribute IDLType             type_def;
        attribute any                 value;
    };

    struct ConstantDescription {
        Identifier          name;
        RepositoryId        id;
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        RepositoryId    defined_in;
        VersionSpec     version;
        TypeCode        type;
        any             value;
    };
};
```

14.5.8.1 Read Interface

The **type** attribute specifies the **TypeCode** describing the type of the constant. The type of a constant must be one of the primitive types allowed in constant declarations (see Constant Declaration on page 55). The **type_def** attribute identifies the definition of the type of the constant.

The **value** attribute contains the value of the constant, not the computation of the value (e.g., the fact that it was defined as “1+2”).

The **describe** operation for a **ConstantDef** object returns a **ConstantDescription**.

14.5.8.2 Write Interface

Setting the **type_def** attribute also updates the **type** attribute.

When setting the **value** attribute, the **TypeCode** of the supplied any must be equal to the **type** attribute of the **ConstantDef**.

14.5.9 TypedefDef

The *base interface* **TypedefDef** is inherited by all named non-object types (structures, unions, enumerations, and aliases). The **TypedefDef** interface is not inherited by the definition objects for primitive or anonymous types.

```
module CORBA {
    interface TypedefDef : Contained, IDLType {};

    struct TypeDescription {
        Identifier      name;
        RepositoryId    id;
        RepositoryId    defined_in;
        VersionSpec     version;
        TypeCode        type;
    };
};
```

The inherited **describe** operation for interfaces derived from **TypedefDef** returns a **TypeDescription**.

14.5.10 StructDef

A **StructDef** represents an OMG IDL structure definition. It can contain structs, unions, and enums.

```
module CORBA {

    struct StructMember {
        Identifier      name;
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        TypeCode        type;
        IDLType         type_def;
    };

    typedef sequence <StructMember> StructMemberSeq;

    interface StructDef : TypedefDef, Container {
        attribute StructMemberSeq       members;
    };
};
```

14.5.10.1 Read Interface

The **members** attribute contains a description of each structure member. The inherited **type** attribute is a **tk_struct TypeCode** describing the structure.

14.5.10.2 Write Interface

Setting the **members** attribute also updates the **type** attribute. When setting the **members** attribute, the **type** member of the **StructMember** structure should be set to **TC_void**.

A **StructDef** used as a **Container** may only contain **StructDef**, **UnionDef**, or **EnumDef** definitions.

14.5.11 UnionDef

A **UnionDef** represents an OMG IDL union definition.

```
module CORBA {
    struct UnionMember {
        Identifier      name;
        any             label;
        TypeCode        type;
        IDLType         type_def;
    };
    typedef sequence <UnionMember> UnionMemberSeq;

    interface UnionDef : TypedefDef, Container {
        readonly attribute TypeCode   discriminator_type;
        attribute IDLType             discriminator_type_def;
        attribute UnionMemberSeq      members;
    };
};
```

14.5.11.1 Read Interface

The **discriminator_type** and **discriminator_type_def** attributes describe and identify the union's discriminator type.

The **members** attribute contains a description of each union member. The **label** of each **UnionMemberDescription** is a distinct value of the **discriminator_type**. Adjacent members can have the same **name**. Members with the same **name** must also have the same **type**. A **label** with type **octet** and value 0 indicates the default union member.

The inherited **type** attribute is a **tk_union TypeCode** describing the union.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.5.11.2 Write Interface

Setting the **discriminator_type_def** attribute also updates the **discriminator_type** attribute and setting the **discriminator_type_def** or **members** attribute also updates the **type** attribute.

When setting the **members** attribute, the **type** member of the **UnionMember** structure should be set to **TC_void**.

A **UnionDef** used as a **Container** may only contain **StructDef**, **UnionDef**, or **EnumDef** definitions.

14.5.12 EnumDef

An **EnumDef** represents an OMG IDL enumeration definition.

```
module CORBA {
    typedef sequence <Identifier> EnumMemberSeq;

    interface EnumDef : TypedefDef {
        attribute EnumMemberSeq     members;
    };
};
```

14.5.12.1 Read Interface

The **members** attribute contains a distinct name for each possible value of the enumeration.

The inherited **type** attribute is a **tk_enum TypeCode** describing the enumeration.

14.5.12.2 Write Interface

Setting the **members** attribute also updates the **type** attribute.

14.5.13 AliasDef

An **AliasDef** represents an OMG IDL typedef that aliases another definition.

```
module CORBA {
    interface AliasDef : TypedefDef {
        attribute IDLType   original_type_def;
    };
};
```

14.5.13.1 Read Interface

The **original_type_def** attribute identifies the type being aliased.

The inherited **type** attribute is a **tk_alias TypeCode** describing the alias.

14.5.13.2 Write Interface

Setting the **original_type_def** attribute also updates the **type** attribute.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.5.14 PrimitiveDef

A **PrimitiveDef** represents one of the OMG IDL primitive types. As primitive types are unnamed, this interface is not derived from **TypedefDef** or **Contained**.

```
module CORBA {
    enum PrimitiveKind {
        pk_null, pk_void, pk_short, pk_long, pk_ushort, pk_ulong,
        pk_float, pk_double, pk_boolean, pk_char, pk_octet,
        pk_any, pk_TypeCode, pk_Principal, pk_string, pk_objref,
        pk_longlong, pk_ulonglong, pk_longdouble, pk_wchar, pk_wstring,
        pk_value_base
    };

    interface PrimitiveDef: IDLType {
        readonly attribute PrimitiveKind    kind;
    };
};
```

The **kind** attribute indicates which primitive type the **PrimitiveDef** represents. There are no **PrimitiveDef**s with kind **pk_null**. A **PrimitiveDef** with kind **pk_string** represents an unbounded string. A **PrimitiveDef** with **kind pk_objref** represents the IDL type **Object**. A **PrimitiveDef** with **kind pk_value_base** represents the IDL type **ValueBase**.

The inherited **type** attribute describes the primitive type. All **PrimitiveDef**s are owned by the Repository. References to them are obtained using **Repository::get_primitive**.

14.5.15 StringDef

A **StringDef** represents an IDL bounded string type. The unbounded string type is represented as a **PrimitiveDef**. As string types are anonymous, this interface is not derived from **TypedefDef** or **Contained**.

```
module CORBA {
    interface StringDef : IDLType {
        attribute unsigned long     bound;
    };
};
```

The **bound** attribute specifies the maximum number of characters in the string and must not be zero. The inherited **type** attribute is a **tk_string TypeCode** describing the string.

14.5.16 WstringDef

A **WstringDef** represents an IDL wide string. The unbounded wide string type is represented as a **PrimitiveDef**. As wide string types are anonymous, this interface is not derived from **TypedefDef** or **Contained**.

```
module CORBA {
    interface WstringDef  :  IDLType {
        attribute unsigned long  bound;
    };
};
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The **bound** attribute specifies the maximum number of wide characters in a wide string, and must not be zero. The inherited **type** attribute is a **tk_wstring TypeCode** describing the wide string.

14.5.17 FixedDef

A **FixedDef** represents an IDL fixed point type.

```
module CORBA {
    interface FixedDef  :  IDLType {
        attribute unsigned short digits;
        attribute short scale;
    };
};
```

The **digits** attribute specifies the total number of decimal digits in the number, and must be from 1 to 31, inclusive. The **scale** attribute specifies the position of the decimal point.

The inherited **type** attribute is a **tk_fixed TypeCode**, which describes a fixed-point decimal number.

14.5.18 SequenceDef

A **SequenceDef** represents an IDL sequence type. As sequence types are anonymous, this interface is not derived from **TypedefDef** or **Contained**.

```
module CORBA {
    interface SequenceDef : IDLType {
        attribute unsigned long         bound;
        readonly attribute TypeCode     element_type;
        attribute IDLType               element_type_def;
    };
};
```

14.5.18.1 Read Interface

The **bound** attribute specifies the maximum number of elements in the sequence. A **bound** of zero indicates an unbounded sequence.

The type of the elements is described by **element_type** and identified by **element_type_def**. The inherited **type** attribute is a **tk_sequence TypeCode** describing the sequence.

14.5.18.2 Write Interface

Setting the **element_type_def** attribute also updates the **element_type** attribute. Setting the **bound** or **element_type_def** attribute also updates the **type** attribute.

14.5.19 ArrayDef

An **ArrayDef** represents an IDL array type. As array types are anonymous, this interface is not derived from **TypedefDef** or **Contained**.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

```
module CORBA {
    interface ArrayDef : IDLType {
        attribute unsigned long         length;
        readonly attribute TypeCode     element_type;
        attribute IDLType               element_type_def;
    };
};
```

14.5.19.1 Read Interface

The **length** attribute specifies the number of elements in the array.

The type of the elements is described by **element_type** and identified by **element_type_def**. Since an **ArrayDef** only represents a single dimension of an array, multi-dimensional IDL arrays are represented by multiple **ArrayDef** objects, one per array dimension. The **element_type_def** attribute of the **ArrayDef** representing the leftmost index of the array, as defined in IDL, will refer to the **ArrayDef** representing the next index to the right, and so on. The innermost **ArrayDef** represents the rightmost index and the element type of the multi-dimensional OMG IDL array.

The inherited **type** attribute is a **tk_array TypeCode** describing the array.

14.5.19.2 Write Interface

Setting the **element_type_def** attribute also updates the **element_type** attribute. Setting the **bound** or **element_type_def** attribute also updates the **type** attribute.

14.5.20 ExceptionDef

An **ExceptionDef** represents an exception definition. It can contain structs, unions, and enums.

```
module CORBA {
    interface ExceptionDef : Contained, Container {
        readonly attribute TypeCode   type;
        attribute StructMemberSeq     members;
    };

    struct ExceptionDescription {
        Identifier      name;
        RepositoryId    id;
        RepositoryId    defined_in;
        VersionSpec     version;
        TypeCode        type;
    };
};
```

14.5.20.1 Read Interface

The **type** attribute is a **tk_except TypeCode** describing the exception. The members **attribute** describes any exception members. The **describe** operation for an **ExceptionDef** object returns an **ExceptionDescription**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.5.20.2 Write Interface

Setting the **members** attribute also updates the **type** attribute. When setting the **members** attribute, the **type** member of the **StructMember** structure is ignored and should be set to **TC_void**.

An **ExceptionDef** used as a **Container** may only contain **StructDef**, **UnionDef**, or **EnumDef** definitions.

14.5.21 AttributeDef

An **AttributeDef** represents the information that defines an attribute of an interface, component, home, valuetype, or eventtype.

```
module CORBA {
    enum AttributeMode {ATTR_NORMAL, ATTR_READONLY};

    interface AttributeDef : Contained {
        readonly attribute TypeCode   type;
        attribute IDLType             type_def;
        attribute AttributeMode       mode;
    };

    struct AttributeDescription {
        Identifier          name;
        RepositoryId        id;
        RepositoryId        defined_in;
        VersionSpec         version;
        TypeCode            type;
        AttributeMode       mode;
    };
};
```

14.5.21.1 Read Interface

The **type** attribute provides the **TypeCode** describing the type of this attribute. The **type_def** attribute identifies the object defining the type of this attribute.

The **mode** attribute specifies read only or read/write access for this attribute.

The **describe** operation for an **AttributeDef** object returns an **AttributeDescription**.

14.5.21.2 Write Interface

Setting the **type_def** attribute also updates the **type** attribute.

14.5.22 ExtAttributeDef

An **ExtAttributeDef** represents the information that defines an attribute of an interface, component, home, valuetype, or eventtype that can potentially have user exceptions associated with it.

```
module CORBA{
    struct ExtAttributeDescription {
        Identifier              name;
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        RepositoryId        id;
        RepositoryId        defined_in;
        VersionSpec         version;
        TypeCode            type;
        AttributeMode       mode;
        ExcDescriptionSeq   get_exceptions;
        ExcDescriptionSeq   put_exceptions;
    };

    interface ExtAttributeDef : AttributeDef {

        // read/write interface
        attribute ExcDescriptionSeq get_exceptions;
        attribute ExcDescriptionSeq set_exceptions;

        // read interface
        ExtAttributeDescription describe_attribute();
    };
```

14.5.22.1 Read Interface

The operations inherited from **AttributeDef** behave exactly the same as in **AttributeDef**. In particular, the **def_kind** attribute that has the value **dk_Attribute**, exactly as in **AttributeDef**.

The **get_exceptions** and **set_exceptions** attributes specify the list of exception types that can be raised by the attribute.

The **describe_attribute** operation for an **ExtAttributeDef** object returns an **ExtAttributeDescription**. that contains information about user exceptions in addition to the information that is available through **AttributeDescription**.

14.5.22.2 Write Interface

Same as for **AttributeDef**.

14.5.23 OperationDef

An **OperationDef** represents the information needed to define an operation of an interface.

```
module CORBA {
    enum OperationMode {OP_NORMAL, OP_ONEWAY};

    enum ParameterMode {PARAM_IN, PARAM_OUT, PARAM_INOUT};

    struct ParameterDescription {
        Identifier          name;
        TypeCode            type;
        IDLType             type_def;
        ParameterMode       mode;
    };
    typedef sequence <ParameterDescription> ParDescriptionSeq;

    typedef Identifier ContextIdentifier;
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    typedef sequence <ContextIdentifier> ContextIdSeq;

    typedef sequence <ExceptionDef> ExceptionDefSeq;
    typedef sequence <ExceptionDescription> ExcDescriptionSeq;

    interface OperationDef : Contained {
        readonly attribute TypeCode   result;
        attribute IDLType             result_def;
        attribute ParDescriptionSeq   params;
        attribute OperationMode       mode;
        attribute ContextIdSeq        contexts;
        attribute ExceptionDefSeq     exceptions;
    };

    struct OperationDescription {
        Identifier          name;
        RepositoryId        id;
        RepositoryId        defined_in;
        VersionSpec         version;
        TypeCode            result;
        OperationMode       mode;
        ContextIdSeq        contexts;
        ParDescriptionSeq   parameters;
        ExcDescriptionSeq   exceptions;
    };
};
```

14.5.23.1 Read Interface

The **result** attribute is a **TypeCode** describing the type of the value returned by the operation. The **result_def** attribute identifies the definition of the returned type.

The **params** attribute describes the parameters of the operation. It is a sequence of **ParameterDescription** structures. The order of the **ParameterDescription**s in the sequence is significant. The **name** member of each structure provides the parameter name. The **type** member is a **TypeCode** describing the type of the parameter. The **type_def** member identifies the definition of the type of the parameter. The **mode** member indicates whether the parameter is an in, out, or inout parameter.

The operation's **mode** is either oneway (i.e., no output is returned) or normal.

The **contexts** attribute specifies the list of context identifiers that apply to the operation.

The **exceptions** attribute specifies the list of exception types that can be raised by the operation.

The inherited **describe** operation for an **OperationDef** object returns an **OperationDescription**.

14.5.23.2 Write Interface

Setting the **result_def** attribute also updates the **result** attribute.

The mode attribute can be set to **OP_ONEWAY** only if the result is **TC_void** and all elements of params have a mode of **PARAM_IN**, and the list of exceptions is empty. If the mode is set to **OP_ONEWAY** when these conditions do not hold, a BAD_PARAM exception is raised with minor code 31.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.5.24 InterfaceDef

An **InterfaceDef** object represents interface definition. It can contain constants, typedefs, exceptions, operations, and attributes.

```
module CORBA {
    interface InterfaceDef;
    typedef sequence <InterfaceDef> InterfaceDefSeq;
    typedef sequence <RepositoryId> RepositoryIdSeq;
    typedef sequence <OperationDescription> OpDescriptionSeq;
    typedef sequence <AttributeDescription> AttrDescriptionSeq;

    interface InterfaceDef : Container, Contained, IDLType {
        // read/write interface

        attribute InterfaceDefSeq          base_interfaces;

        // read interface

        boolean is_a (in RepositoryId interface_id);

        struct FullInterfaceDescription {
            Identifier                     name;
            RepositoryId                   id;
            RepositoryId                   defined_in;
            VersionSpec                    version;
            OpDescriptionSeq               operations;
            AttrDescriptionSeq             attributes;
            RepositoryIdSeq                base_interfaces;
            TypeCode                       type;
        };

        FullInterfaceDescription describe_interface();

        // write interface

        AttributeDef create_attribute (
            in RepositoryId        id,
            in Identifier          name,
            in VersionSpec         version,
            in IDLType             type,
            in AttributeMode       mode
        );

        OperationDef create_operation (
            in RepositoryId        id,
            in Identifier          name,
            in VersionSpec         version,
            in IDLType             result,
            in OperationMode       mode,
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            in ParDescriptionSeq  params,
            in ExceptionDefSeq    exceptions,
            in ContextIdSeq       contexts
        );
    };

    struct InterfaceDescription {
        Identifier          name;
        RepositoryId        id;
        RepositoryId        defined_in;
        VersionSpec         version;
        RepositoryIdSeq     base_interfaces;
    };
};
```

14.5.24.1 Read Interface

The **base_interfaces** attribute lists all the interfaces from which this interface inherits.

The **is_a** operation returns **TRUE** if the interface on which it is invoked either is identical to or inherits, directly or indirectly, from the interface identified by its **interface_id** parameter. Otherwise it returns **FALSE**. If the value of **interface_id** is **IDL:omg.org/CORBA/Object:1.0**, **is_a** returns **TRUE** signifying the fact that all interfaces are implicitly derived from the base type **Object**.

The **describe_interface** operation returns a **FullInterfaceDescription** describing the interface, including its operations and attributes. The **operations** and **attributes** fields of the **FullInterfaceDescription** structure include descriptions of all of the operations and attributes in the transitive closure of the inheritance graph of the interface being described.

The inherited **describe** operation for an **InterfaceDef** returns an **InterfaceDescription**.

The inherited **contents** operation returns the list of constants, typedefs, and exceptions defined in this **InterfaceDef** and the list of attributes and operations either defined or inherited in this **InterfaceDef**. If the **exclude_inherited** parameter is set to **TRUE**, only attributes and operations defined within this interface are returned. If the **exclude_inherited** parameter is set to **FALSE**, all attributes and operations are returned.

14.5.24.2 Write Interface

Setting the **base_interfaces** attribute causes a BAD_PARAM exception with minor code 5 to be raised if the **name** attribute of any object contained by this **InterfaceDef** conflicts with the **name** attribute of any object contained by any of the specified base **InterfaceDef**s.

The **create_attribute** operation returns a new **AttributeDef** contained in the **InterfaceDef** on which it is invoked. The **id**, **name**, **version, type_def**, and **mode** attributes are set as specified. The **type** attribute is also set. The **defined_in** attribute is initialized to identify the containing **InterfaceDef**. A BAD_PARAM exception with standard minor code 2 is raised if an object with the specified **id** already exists in the **Repository**. BAD_PARAM exception with standard minor code 3 is raised if an object with the same **name** already exists in this **InterfaceDef**.

The **create_operation** operation returns a new **OperationDef** contained in the **InterfaceDef** on which it is invoked. The **id**, **name**, **version**, **result_def**, **mode**, **params**, **exceptions**, and **contexts** attributes are set as specified. The **result** attribute is also set. The **defined_in** attribute is initialized to identify the containing **InterfaceDef**. A

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

BAD_PARAM exception with standard minor code 2 is raised if an object with the specified **id** already exists in the **Repository**. BAD_PARAM exception with standard minor code 3 is raised if an object with the same **name** already exists in this **InterfaceDef**.

An **InterfaceDef** used as a **Container** may only contain **TypedefDef**, (including definitions derived from **TypedefDef**), **ConstantDef**, and **ExceptionDef** definitions.

14.5.25 ExtInterfaceDef

An **ExtInterfaceDef** object represents interface definition. It can contain constants, typedefs, exceptions, operations, and attributes with exceptions.

```
module CORBA {

    interface InterfaceAttrExtension {

        // read interface

        struct ExtFullInterfaceDescription {
            Identifier              name;
            RepositoryId            id;
            RepositoryId            defined_in;
            VersionSpec             version;
            OpDescriptionSeq        operations;
            ExtAttrDescriptionSeq   attributes;
            RepositoryIdSeq         base_interfaces;
            TypeCode                type;
        };
        ExtFullInterfaceDescription describe_ext_interface();

        // write interface
        ExtAttributeDef create_ext_attribute (
            in RepositoryId         id,
            in Identifier           name,
            in VersionSpec          version,
            in IDLType              type,
            in AttributeMode        mode,
            in ExceptionDefSeq      get_exceptions,
            in ExceptionDefSeq      set_exceptions
        );
    };

    interface ExtInterfaceDef : InterfaceDef,
        InterfaceAttrExtension {
    };
};
```

14.5.25.1 Read Interface

All operations and attributes inherited from **InterfaceDef** behave the same as for **InterfaceDef**. In particular, the **def_kind** attribute has the value **dk_Interface**, exactly as in **InterfaceDef**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The inherited **describe_ext_interfaces** operation returns the **ExtFullInterfaceDescription** structure that contains information about attributes with exceptions, in addition to the information found in **FullInterfaceDescription**.

14.5.25.2 Write Interface

All operations and attributes inherited from **InterfaceDef** behave the same as for **InterfaceDef**.

The inherited **create_ext_attribute** operation returns a new **ExtAttributeDef** contained in the **ExtInterfaceDef** on which it is invoked. The **id**, **name**, **version**, **type_def**, **mode, get_exceptions** and **set_exceptions** attributes are set as specified. The **type** attribute is also set. The **defined_in** attribute is initialized to identify the containing **ExtInterfaceDef**. A BAD_PARAM exception with standard minor code 2 is raised if an object with the specified **id** already exists in the **Repository**. BAD_PARAM exception with standard minor code 3 is raised if an object with the same **name** already exists in this **ExtInterfaceDef**.

14.5.26 AbstractInterfaceDef

An **AbstractInterfaceDef** object represents a CORBA 2.3 abstract interface definition. It can contain constants, typedefs, exceptions, operations, and attributes. Its base interfaces can only contain **AbstractInterfaceDef**s.

```
module CORBA {
    interfaceAbstractInterfaceDef;
    typedef sequence <AbstractInterfaceDef> AbstractInterfaceDefSeq;
    interface AbstractInterfaceDef : InterfaceDef {
    };
};
```

14.5.26.1 Read Interface

The inherited **base_interfaces** attribute returns a list of abstract interfaces from which this abstract interface inherits.

NOTE: base_interfaces is of type **InterfaceDefSeq**, but since **AbstractInterfaceDef** is derived from **InterfaceDef**, a list of **AbstractInterfaceDefs** can legitimately be returned in an **InterfaceDefSeq**.

The inherited **is_a** operation returns **TRUE** if the interface on which it is invoked either is identical to or inherits, directly or indirectly, from the abstract interface identified by its **interface_id** parameter, or if the value of **interface_id** is **IDL:omg.org/CORBA/AbstractBase:1.0**. Otherwise it returns **FALSE**.

The inherited **describe_interface** operation returns a **FullInterfaceDescription** describing the abstract interface, including its operations and attributes.

The inherited **describe** operation for an **AbstractInterfaceDef** returns an **InterfaceDescription**.

The inherited **contents** operation returns the list of constants, typedefs, and exceptions defined in this **AbstractInterfaceDef** and the list of attributes and operations either defined or inherited in this **AbstractInterfaceDef**. If the **exclude_inherited** parameter is set to **TRUE**, only attributes and operations defined within this abstract interface are returned. If the **exclude_inherited** parameter is set to **FALSE**, all attributes and operations are returned.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.5.26.2 Write Interface

Setting the inherited **base_interfaces** attribute causes a BAD_PARAM exception with standard minor code 5 to be raised if the name attribute of any object contained by this **AbstractInterfaceDef** conflicts with the name attribute of any object contained by any of the specified base **AbstractInterfaceDef**s. If any of the **InterfaceDef**s in **base_interface** are not **AbstractInterfaceDef**s, then a BAD_PARAM exception with standard minor code 11 is raised.

The inherited **create_attribute** operation returns a new **AttributeDef** contained in the **AbstractInterfaceDef** on which it is invoked. The **id**, **name**, **version**, **type_def**, and **mode** attributes are set as specified. The **type** attribute is also set. The **defined_in** attribute is initialized to identify the containing **AbstractInterfaceDef**. A BAD_PARAM exception with standard minor code 2 is raised if an object with the specified **id** already exists in the **Repository**. BAD_PARAM exception with standard minor code 3 is raised if an object with the same **name** already exists in this **AbstractInterfaceDef**.

The inherited **create_operation** operation returns a new **OperationDef** contained in the **AbstractInterfaceDef** on which it is invoked. The **id**, **name**, **version**, **result_def**, **mode**, **params**, **exceptions**, and **contexts** attributes are set as specified. The **result** attribute is also set. The **defined_in** attribute is initialized to identify the containing **AbstractInterfaceDef**. A BAD_PARAM exception with standard minor code 2 is raised if an object with the specified **id** already exists in the **Repository**. BAD_PARAM exception with standard minor code 3 is raised if an object with the same **name** already exists in this **AbstractInterfaceDef**.

14.5.27 ExtAbstractInterfaceDef

An **ExtAbstractInterfaceDef** object represents an abstract interface definition. It can contain constants, typedefs, exceptions, operations, and attributes with exceptions. Its base interfaces can only contain **ExtAbstractInterfaceDef**s.

```
module CORBA {

    interface ExtAbstaractInterfaceDef :   AbstractInterfaceDef,
                                           InterfaceAttrExtension {
    };
};
```

14.5.27.1 Read Interface

All operations and attributes inherited from **AbstractInterfaceDef** behave the same as for **AbstaractInterfaceDef**. In particular, the **def_kind** attribute has the value **dk_AbstractInterface**, exactly as in **AbstaractInterfaceDef**.

The inherited **describe_ext_interface** operation returns the **ExtFullInterfaceDescription** structure that contains information about attributes with exceptions, in addition to the information found in **FullInterfaceDescription**.

14.5.27.2 Write Interface

All operations and attributes inherited from **AbstaractInterfaceDef** behave the same as for **AbstractInterfaceDef**.

The inherited **create_ext_attribute** operation returns a new **ExtAttributeDef** contained in the **ExtAbstractInterfaceDef** on which it is invoked. The **id**, **name**, **version**, **type_def**, **mode, get_exceptions,** and **set_exceptions** attributes are set as specified. The **type** attribute is also set. The **defined_in** attribute is initialized to identify the containing **ExtAbstractInterfaceDef**. A BAD_PARAM exception with standard minor code 2 is raised if an object with the specified **id** already exists in the **Repository**. BAD_PARAM exception with standard minor code 3 is raised if an object with the same **name** already exists in this **ExtAbstractInterfaceDef**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.5.28 LocalInterfaceDef

A **LocalInterfaceDef** object represents a local interface definition. It can contain constants, typedefs, exceptions, operations, and attributes. Its base interfaces can only contain **InterfaceDefs** or **LocalInterfaceDefs**.

```
module CORBA {
    interfaceLocalInterfaceDef;
    typedef sequence <LocalInterfaceDef> LocalInterfaceDefSeq;

    interface LocalInterfaceDef : InterfaceDef {
    };
};
```

14.5.28.1 Read Interface

The inherited **base_interfaces** attribute returns a list of interfaces, local or otherwise, from which this local interface inherits.

NOTE: **base_interfaces** is of type **InterfaceDefSeq**, but since **LocalInterfaceDef** is derived from **InterfaceDef**, a list that consists of some regular **InterfaceDefs** and some **LocalInterfaceDefs** can legitimately be returned in an **InterfaceDefSeq**.

The inherited **is_a** operation returns **TRUE** if the local interface on which it is invoked either is identical to or inherits, directly or indirectly, from the local interface identified by its **interface_id** parameter, or if the value of **interface_id** is **IDL:omg.org/CORBA/LocalBase:1.0**. Otherwise it returns **FALSE**.

The inherited **describe_interface** operation returns a **FullInterfaceDescription** describing the local interface, including its operations and attributes.

The inherited **describe** operation for a **LocalInterfaceDef** returns an **InterfaceDescription**.

The inherited **contents** operation returns the list of constants, typedefs, and exceptions defined in this **LocalInterfaceDef** and the list of attributes and operations either defined or inherited in this **LocalInterfaceDef**. If the **exclude_inherited** parameter is set to **TRUE**, only attributes and operations defined within this local interface are returned. If the **exclude_inherited** parameter is set to **FALSE**, all attributes and operations are returned.

14.5.28.2 Write Interface

Setting the inherited **base_interfaces** attribute causes a BAD_PARAM exception with standard minor code 5 to be raised if the name attribute of any object contained by this **LocalInterfaceDef** conflicts with the name attribute of any object contained by any of the specified base **InterfaceDef**s (local or otherwise).

The inherited **create_attribute** operation returns a new **AttributeDef** contained in the **LocalInterfaceDef** on which it is invoked. The **id**, **name**, **version**, **type_def**, and **mode** attributes are set as specified. The **type** attribute is also set. The **defined_in** attribute is initialized to identify the containing **LocalInterfaceDef**. A BAD_PARAM exception with standard minor code 2 is raised if an object with the specified **id** already exists in the **Repository**. BAD_PARAM exception with standard minor code 3 is raised if an object with the same **name** already exists in this **LocalInterfaceDef**.

The inherited **create_operation** operation returns a new **OperationDef** contained in the **LocalInterfaceDef** on which it is invoked. The **id**, **name**, **version**, **result_def**, **mode**, **params**, **exceptions**, and **contexts** attributes are set as specified. The **result** attribute is also set. The **defined_in** attribute is initialized to identify the containing

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

LocalInterfaceDef. A BAD_PARAM exception with standard minor code 2 is raised if an object with the specified **id** already exists in the **Repository**. BAD_PARAM exception with standard minor code 3 is raised if an object with the same **name** already exists in this **LocalInterfaceDef**.

14.5.29 ExtLocalInterfaceDef

An **ExtLocalInterfaceDef** object represents a local interface definition. It can contain constants, typedefs, exceptions, operations, and attributes with exceptions. Its base interfaces can only contain **ExtInterfaceDef**s or **ExtLocalInterfaceDef**s.

```
module CORBA {

    interface ExtLocalInterfaceDef :    LocalInterfaceDef,
                                        InterfaceAttrExtension {
    };
};
```

14.5.29.1 Read Interface

All operations and attributes inherited from **LocalInterfaceDef** behave the same as for **LocalInterfaceDef**. In particular, the **def_kind** attribute has the value **dk_LocalInterface**, exactly as in **LocalInterfaceDef**.

The inherited **describe_ext_interface** operation returns the **ExtFullInterfaceDescription** structure that contains information about attributes with exceptions, in addition to the information found in **FullInterfaceDescription**.

14.5.29.2 Write Interface

All operations and attributes inherited from **LocalInterfaceDef** behave the same as for **LocalInterfaceDef**.

The inherited **create_ext_attribute** operation returns a new **ExtAttributeDef** contained in the **ExtLocalInterfaceDef** on which it is invoked. The **id**, **name**, **version**, **type_def**, **mode, get_exceptions,** and **set_exceptions** attributes are set as specified. The **type** attribute is also set. The **defined_in** attribute is initialized to identify the containing **ExtLocalInterfaceDef**. A BAD_PARAM exception with standard minor code 2 is raised if an object with the specified **id** already exists in the **Repository**. BAD_PARAM exception with standard minor code 3 is raised if an object with the same **name** already exists in this **ExtLocalInterfaceDef**.

14.5.30 ValueMemberDef

A **ValueMemberDef** IR Object represents a value member.

```
module CORBA {
    typedef short Visibility;
        const Visibility PRIVATE_MEMBER = 0;
        const Visibility PUBLIC_MEMBER = 1;

    struct ValueMember {
        Identifier          name;
        RepositoryId        id;
        RepositoryId        defined_in;
        VersionSpec         version;
        TypeCode            type;
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        IDLType             type_def;
        Visibility          access;
    };

    typedef sequence <ValueMember> ValueMemberSeq;

    interface ValueMemberDef : Contained {
        readonly attribute TypeCode   type;
        attribute IDLType             type_def;
        attribute Visibility          access;
    };
};
```

14.5.30.1 Read Interface

The **type** attribute provides the **TypeCode** describing the type of this value member. The **type_def** attribute identifies the object defining the type of this value member. The **access** attribute specifies private or public access for this value member. The describe operation for a **ValueMemberDef** object returns a **ValueMember**.

14.5.30.2 Write Interface

Setting the **type_def** attribute also updates the **type** attribute.

14.5.31 ValueDef

A **ValueDef** object represents a value definition. It can contain constants, typedefs, exceptions, operations, and attributes.

```
module CORBA {
    interface ValueDef;
    typedef sequence <ValueDef> ValueDefSeq;

    struct Initializer {
        StructMemberSeq  members;
        Identifier       name;
    };

    typedef sequence<Initializer> InitializerSeq;

    interface ValueDef : Container, Contained, IDLType {
        // read/write interface

        attribute InterfaceDefSeq  supported_interfaces;
        attribute InitializerSeq   initializers;
        attribute ValueDef         base_value;
        attribute ValueDefSeq      abstract_base_values;
        attribute boolean          is_abstract;
        attribute boolean          is_custom;
        attribute boolean          is_truncatable;

        // read interface
        boolean is_a(
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        in RepositoryId         id
    );

    struct FullValueDescription {
        Identifier              name;
        RepositoryId            id;
        boolean                 is_abstract;
        boolean                 is_custom;
        RepositoryId            defined_in;
        VersionSpec             version;
        OpDescriptionSeq        operations;
        AttrDescriptionSeq      attributes;
        ValueMemberSeq          members;
        InitializerSeq          initializers;
        RepositoryIdSeq         supported_interfaces;
        RepositoryIdSeq         abstract_base_values;
        boolean                 is_truncatable;
        RepositoryId            base_value;
        TypeCode                type;
    };

    FullValueDescription describe_value();

    // write interface

    ValueMemberDef create_value_member(
        in RepositoryId         id,
        in Identifier           name,
        in VersionSpec          version,
        in IDLType              type,
        in Visibility           access
    );

    AttributeDef create_attribute(
        in RepositoryId         id,
        in Identifier           name,
        in VersionSpec          version,
        in IDLType              type,
        in AttributeMode        mode
    );

    OperationDef create_operation (
        in RepositoryId         id,
        in Identifier           name,
        in VersionSpec          version,
        in IDLType              result,
        in OperationMode        mode,
        in ParDescriptionSeq    params,
        in ExceptionDefSeq      exceptions,
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            in ContextIdSeq         contexts
        );
    };

    struct ValueDescription {
        Identifier              name;
        RepositoryId            id;
        boolean                 is_abstract;
        boolean                 is_custom;
        RepositoryId            defined_in;
        VersionSpec             version;
        RepositoryIdSeq         supported_interfaces;
        RepositoryIdSeq         abstract_base_values;
        boolean                 is_truncatable;
        RepositoryId            base_value;
    };
};
```

14.5.31.1 Read Interface

The **supported_interfaces** attribute lists the interfaces that this value type supports.

The **initializers** attribute lists the initializers this value type supports.

The **base_value** attribute describes the value type from which this value inherits.

The **abstract_base_values** attribute lists the abstract value types from which this value inherits.

The **is_abstract** attribute is **TRUE** if the value is an abstract value type.

The **is_custom** attribute is **TRUE** if the value uses custom marshaling.

The **is_truncatable** attribute is **TRUE** if the value inherits “safely” (i.e., supports truncation) from another value.

The **is_a** operation returns **TRUE** if the value on which it is invoked either is identical to or inherits, directly or indirectly, from the interface or value identified by its **id** parameter or if the value of **id** is **IDL:omg.org/CORBA/ValueBase:1.0.** Otherwise it returns **FALSE**.

The **describe_value** operation returns a **FullValueDescription** describing the value, including its operations and attributes.

The inherited **describe** operation for a **ValueDef** returns a **ValueDescription**.

The inherited **contents** operation returns the list of constants, typedefs, and exceptions defined in this **ValueDef** and the list of attributes, operations, and members either defined or inherited in this **ValueDef**. If the **exclude_inherited** parameter is set to **TRUE**, only attributes, operations, and members defined within this value are returned. If the **exclude_inherited** parameter is set to **FALSE**, all attributes, operations, and members are returned.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.5.31.2 Write Interface

Setting the **supported_interfaces, base_value,** or **abstract_base_values** attribute causes a BAD_PARAM exception with minor code 5 to be raised if the **name** attribute of any object contained by this **ValueDef** conflicts with the **name** attribute of any object contained by any of the specified bases. If an attempt is made to set the **supported_interfaces** attribute to an **InterfaceDefSeq** that contains more than one **InterfaceDef** that is not an **AbstractInterfaceDef**, then the BAD_PARAM exception shall be raised with standard minor code 12.

The **create_value_member** operation returns a new **ValueMemberDef** contained in the **ValueDef** on which it is invoked. The **id**, **name**, **version, type_def**, and **access** attributes are set as specified. The **type** attribute is also set. The **defined_in** attribute is initialized to identify the containing **ValueDef**. A BAD_PARAM exception with minor code 2 is raised if an object with the specified **id** already exists in the **Repository**. A BAD_PARAM exception with minor code 3 is raised if an object with the same **name** already exists in this **ValueDef**.

The **create_attribute** operation returns a new **AttributeDef** contained in the **ValueDef** on which it is invoked. The **id**, **name**, **version, type_def**, and **mode** attributes are set as specified. The **type** attribute is also set. The **defined_in** attribute is initialized to identify the containing **ValueDef**. A BAD_PARAM exception with minor code 2 is raised if an object with the specified **id** already exists in the **Repository**. A BAD_PARAM exception with minor code 3 is raised if an object with the same **name** already exists in this **ValueDef**.

The **create_operation** operation returns a new **OperationDef** contained in the **ValueDef** on which it is invoked. The **id**, **name**, **version**, **result_def**, **mode**, **params**, **exceptions**, and **contexts** attributes are set as specified. The **result** attribute is also set. The **defined_in** attribute is initialized to identify the containing **ValueDef**. A BAD_PARAM exception with minor code 2 is raised if an object with the specified **id** already exists in the **Repository**. A BAD_PARAM exception with minor code 3 is raised if an object with the same **name** already exists in this **ValueDef**.

A **ValueDef** used as a **Container** may only contain **TypedefDef**, (including definitions derived from **TypedefDef**), **ConstantDef**, and **ExceptionDef** definitions.

14.5.32 ExtValueDef

An **ExtValueDef** object represents a value definition. It can contain constants, typedefs, exceptions, operations, and attributes with exceptions. Value definitions that contain initializers with user exceptions can also be represented in **ExtValueDef** objects.

```
module CORBA {

    struct ExtInitializer {
        StructMemberSeq     members;
        ExcDescriptionSeq   exceptions;
        Identifier          name;
    };
    typedef sequence <ExtInitializer> ExtInitializerSeq;

    interface ExtValueDef : ValueDef {

        // read/write interface
        attribute ExtInitializerSeq ext_initializers;

        // read interface
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        struct ExtFullValueDescription {
            Identifier              name;
            RepositoryId            id;
            boolean                 is_abstract;
            boolean                 is_custom;
            RepositoryId            defined_in;
            VersionSpec             version;
            OpDescriptionSeq        operations;
            ExtAttrDescriptionSeq   attributes;
            ValueMemberSeq          members;
            ExtInitializerSeq       initializers;
            RepositoryIdSeq         supported_interfaces;
            RepositoryIdSeq         abstract_base_values;
            boolean                 is_truncatable;
            RepositoryId            base_value;
            TypeCode                type;
        };

        ExtFullValueDescription describe_ext_value();

        // write interface
        ExtAttributeDef create_ext_attribute (
            in RepositoryId         id,
            in Identifier           name,
            in VersionSpec          version,
            in IDLType              type,
            in AttributeMode        mode,
            in ExceptionDefSeq      get_exceptions,
            in ExceptionDefSeq      set_exceptions
        );
    };
};
```

14.5.32.1 Read Interface

All operations and attributes inherited from **ValueDef** behave the same as for **ValueDef**. In particular, the **def_kind** attribute has the value **dk_Value**, exactly as in **ValueDef.**

The **ext_initializers** attribute lists the initializers with exceptions that this value type supports.

The inherited **initializers** attribute lists the same initializers as in **ext_initializers** but does not have the exception information.

The **describe_ext_value** operation returns the **ExtFullValueDescription** structure that contains information about attributes with exceptions and initializers with exceptions, in addition to the information found in **FullValueDescription**.

14.5.32.2 Write Interface

All operations and attributes inherited from **ValueDef** behave the same as for **ValueDef**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The **create_ext_attribute** operation returns a new **ExtAttributeDef** contained in the **ExtValueDef** on which it is invoked. The **id**, **name**, **version**, **type_def**, **mode, get_exceptions**, and **set_exceptions** attributes are set as specified. The **type** attribute is also set. The **defined_in** attribute is initialized to identify the containing **ExtValueDef**. A BAD_PARAM exception with standard minor code 2 is raised if an object with the specified **id** already exists in the **Repository**. BAD_PARAM exception with standard minor code 3 is raised if an object with the same **name** already exists in this **ExtValueDef**.

14.5.33 ValueBoxDef

A **ValueBoxDef** object represents a value box definition. It merely identifies the IDL **type_def** that is being "boxed."

```
module CORBA {
    interface ValueBoxDef : TypedefDef {
        attribute IDLType original_type_def;
    };
};
```

14.5.33.1 Read Interface

The **original_type_def** attribute identifies the type being boxed. The inherited **type** attribute is a **tk_value_box TypeCode** describing the value box.

14.5.33.2 Write Interface

Setting the **original_type_def** attribute also updates the **type** attribute.

14.5.34 NativeDef

A **NativeDef** object represents a native definition.

```
module CORBA {
    interface NativeDef : TypedefDef {};
};
```

The inherited **type** attribute is a **tk_native TypeCode** describing the native type.

14.6 Component Interface Repository Interfaces

The **IRObject**s that represent IDL concepts that are specific to the Components extension are described in this sub clause. These IRObjects can be contained only in a **ComponentIR::Repository** described in this sub clause.

14.6.1 ComponentIR::Container

The *base interface* **ComponentIR::Container** is used to form a containment hierarchy in the Component Interface Repository.

```
module CORBA {
    module ComponentIR {
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        interface Container {
            ComponentDef create_component (
                in RepositoryId id,
                in Identifier name,
                in VersionSpec version,
                in ComponentDef base_component,
                in InterfaceDefSeq supports_interfaces
            );

            HomeDef create_home (
                in RepositoryId id,
                in Identifier name,
                in VersionSpec version,
                in HomeDef base_home,
                in ComponentDef managed_component,
                in InterfaceDefSeq supports_interfaces,
                in ValueDef primary_key
            );

            EventDef create_event (
                in RepositoryId      id,
                in Identifier        name,
                in VersionSpec       version,
                in boolean           is_custom,
                in boolean           is_abstract,
                in ValueDef          base_value,
                in boolean           is_truncatable,
                in ValueDefSeq       abstract_base_values,
                in InterfaceDefSeq   supported_interfaces,
                in ExtInitializerSeq initializers
            );
        };
    };
};
```

14.6.1.1 Write Interface

The three **create_<type>** operations defined in the **ComponentIR::Container** interface create new empty IR objects defining component, home, and event types. The **defined_in** attribute of a definition created with any of these operations is initialized to identify the **ComponentIR::Container** on which the operation is invoked, and the **containing_repository** attribute is initialized to its **ComponentIR::Repository**.

These **create_<type>** operations all take **id** and **name** parameters that are used to initialize the identity of the created definition.

- A BAD_PARAM exception is raised with minor code 2 if an object with the specified **id** already exists in the **ComponentIR::Repository**.
- A BAD_PARAM exception with minor code 3 is raised if the specified **name** already exists within this **ComponentIR::Container** and multiple versions are not supported.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The **create_component** operation returns a new empty **ComponentDef** with the specified **base_component**, and the specified **supports_interfaces**. **AttributeDef**s can be added using **ComponentDef::create_attribute**. **ComponentDef::create_provides**, **ComponentDef::create_uses**, **ComponentDef::create_emits**, **ComponentDef::create_publishes**, and **ComponentDef::create_consumes** can be used to add **ProvidesDef**s, **UsesDef**s, **EmitsDef**s, **PublishesDef**s, and **ConsumesDef**s respectively. Definitions can also be added using the **Contained::move** operation.

The **create_home** operation returns a new **HomeDef** with the specified **base_home**, **managed_component, supported_interfaces,** and **primary_key**. Type, exception, and constant definitions can be added using **Container::create_<type>** operations on the new **HomeDef**s. **OperationDef**s can be added using **HomeDef::create_operation** and **AttributeDef**s can be added using **HomeDef::create_attribute**. **FinderDef**s and **FactoryDef**s can be added using **HomeDef::create_finder** and **HomeDef::create_factory** respectively. Definitions can also be added using the **Contained::move** operation.

The **create_event** operation returns a new empty **EventDef** with the specified base interfaces and events (**base_value**, **supported_interfaces**, and **abstract_base_values**) as well as the other information describing the new events characteristics (**is_custom**, **is_abstract**, **is_truncatable**, and **initializers**). The **initializers** argument is of type **ExtInitializerSeq** allowing one to specify user exceptions for initializers. Type, exception, and constant definitions can be added using **Container::create_<type>** operations on the new **EventDef**. **OperationDef**s can be added using **ExtValueDef::create_operation** and **ExtAttributeDef**s can be added using **ExtValueDef::create_ext_attribute**. Definitions can also be added using the **Contained::move** operation.

14.6.2 ComponentIR::Repository

ComponentIR::Repository is an interface that provides global access to the Interface Repository that supports access to information related to CORBA Components. The **ComponentIR::Repository** object can contain components, home, and event definitions in addition to everything else that a **Repository** type can contain. As it inherits from **Container** and **ComponentIR::Container**, it can be used to look up any definition (whether globally defined or defined within a module or interface) either by name or by id.

Since **ComponentIR::Repository** derives from **CORBA::Repository** and hence from **Container** and not from **Contained**, it does not have a **RepositoryId** associated with it. By default it is deemed to have the **RepositoryId** "" (the empty string) for purposes of assigning a value to the **defined_in** field of the description structure of **ModuleDef**, **InterfaceDef**, **ValueDef**, **ValueBoxDef**, **ComponentDef**, **HomeDef**, **EventDef**, **TypedefDef**, **ExceptionDef**, and **ConstantDef** that are contained immediately in the **ComponentIR::Repository** object. Since **ComponentIR::Repository** derives from **ComponentIR::Container**, it can contain **ComponentDef**s, **HomeDef**s as well as **EventDef**s.

```
module CORBA {
    module ComponentIR {

        interface Repository : CORBA::Repository, Container {};
    };
};
```

14.6.2.1 Read Interface

ComponentIR::Repository has the same read operations as **Repository**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.6.2.2 Write Interface

Write operations inherited from **ComponentIR::Container** behave the same way as in **ComponentIR::Container**.

The rest of the write operations are inherited from **CORBA::Repository** and behave the same way as in **CORBA::Repository**.

14.6.3 ComponentIR::ProvidesDef

A **ComponentIR::ProvidesDef** object represents an interface that is provided by a component.

```
module CORBA {
    module ComponentIR {

        interface ProvidesDef : Contained {
            attribute InterfaceDef interface_type;
        };

        struct ProvidesDescription {
            Identifier name;
            RepositoryId id;
            RepositoryId defined_in;
            VersionSpec version;
            RepositoryId interface_type;
        };
    };
};
```

14.6.3.1 Read Interface

The attribute **interface_type** returns the object identifying the interface that is provided by the component.

The inherited operation **describe** returns a **ProvidesDescription**.

14.6.3.2 Write Interface

Setting the attribute **interface_type** changes the object identifying the interface that is provided by the component.

The rest of the write operations are inherited from **CORBA::Contained** and behave the same way as in **CORBA::Contained**.

14.6.4 ComponentIR::UsesDef

A **ComponentIR::UsesDef** object represents an interface that is used by a component.

```
module CORBA {
    module ComponentIR {

        interface UsesDef : Contained {
            attribute InterfaceDef interface_type;
            attribute boolean is_multiple;
        };
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        struct UsesDescription {
            Identifier name;
            RepositoryId id;
            RepositoryId defined_in;
            VersionSpec version;
            RepositoryId interface_type;
            boolean is_multiple;
        };
    };
};
```

14.6.4.1 Read Interface

The attribute **interface_type** returns the object identifying the interface that is used by the component.

The attribute **is_multiple** is **TRUE** if the interface is used multiple times.

The inherited operation **describe** returns a **UsesDescription**.

14.6.4.2 Write Interface

Setting the attribute **interface_type** changes the object identifying the interface that is used by the component. Setting the attribute **is_multiple** changes the multiplicity of the used interface.

The rest of the write operations are inherited from **CORBA::Contained** and behave the same way as in **CORBA::Contained**.

14.6.5 ComponentIR::EventDef

A **ComponentIR::EventDef** object represents an eventtype definition. It can contain constants, typedefs, exceptions, operations, and attributes with exceptions. Eventtype definitions that contain initializers with user exceptions can also be represented in **ComponentIR::EventDef** objects.

```
module CORBA {
    module ComponentIR {

        interface EventDef : ExtValueDef {};
};
```

The read and write interfaces for **ComponentIR::EventDef** have the same semantics as the read and write interfaces for **ExtValueDef**.

14.6.6 ComponentIR::EventPortDef

A **ComponentIR::EventPortDef** object represents an event port definition. It refers to an **EventDef** object that contains the actual information about the event. This interface is never instantiated as itself. It is instantiated only as one of its derived types (i.e., **EmitsDef**, **PublishesDef**, or **ConsumesDef**).

```
module CORBA {
    module ComponentIR {
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        interface EventPortDef : Contained {
            // read/write interface
            attribute EventDef event;

            // read interface
            boolean is_a (in RepositoryId event_id);
        };

        struct EventPortDescription {
            Identifier              name;
            RepositoryId            id;
            RepositoryId            defined_in;
            VersionSpec             version;
            RepositoryId            event;
        };
    };
};
```

14.6.6.1 Read Interface

The **event** attribute returns the object containing the definition of the event for this event port.

The **is_a** operation returns **TRUE** if the event value associated with this **EventPortDef** is identical to or inherits from the event value associated with the **EventPortDef** identified by the **event_id**.

The inherited **describe** operation returns an **EventPortDescription**.

14.6.6.2 Write Interface

Setting the attribute **event** changes the object containing the definition of the event for this event port.

The rest of the write operations are inherited from **CORBA::Contained** and behave the same way as in **CORBA::Contained**.

14.6.7 ComponentIR::EmitsDef

A **ComponentIR::EmitsDef** object represents the port definition of an event that is emitted by a component.

```
module CORBA {
    module ComponentIR {

        interface EmitsDef : EventPortDef {};
    };
};
```

14.6.7.1 Read Interface

The read interface for **EmitsDef** has the same semantics as the read interface for **EventPortDef**.

14.6.7.2 Write Interface

The write interface for **EmitsDef** has the same semantics as the write interface for **EventPortDef**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.6.8 ComponentIR::PublishesDef

A **ComponentIR::PublishesDef** object represents the port definition of an event that is published by a component.

```
module CORBA {
    module ComponentIR {

        interface PublishesDef : EventPortDef {};
    };
};
```

14.6.8.1 Read Interface

The read interface for **PublishesDef** has the same semantics as the read interface for **EventPortDef**.

14.6.8.2 Write Interface

The write interface for **PublishesDef** has the same semantics as the write interface for **EventPortDef**.

14.6.9 ComponentIR::ConsumesDef

A **ComponentIR::ConsumesDef** object represents the port definition of an event that is consumed by a component.

```
module CORBA {
    module ComponentIR {

        interface ConsumesDef : EventPortDef {};
    };
};
```

14.6.9.1 Read Interface

The read interface for **ConsumesDef** has the same semantics as the read interface for **EventPortDef**.

14.6.9.2 Write Interface

The write interface for **ConsumesDef** has the same semantics as the write interface for **EventPortDef**.

14.6.10 ComponentIR::ComponentDef

A **ComponentIR::ComponentDef** object represents the definition of a component. It contains provides, uses, emits, publishes, consumes, and attributes.

```
module CORBA {
    module ComponentIR {

        interface ComponentDef : ExtInterfaceDef {
            // read/write interface
            attribute ComponentDef base_component;
            attribute InterfaceDefSeq supported_interfaces;
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        // write interface
        ProvidesDef create_provides (
            in RepositoryId id,
            in Identifier name,
            in VersionSpec version,
            in InterfaceDef interface_type
        );

        UsesDef create_uses (
            in RepositoryId id,
            in Identifier name,
            in VersionSpec version,
            in InterfaceDef interface_type,
            in boolean is_multiple
        );

        EmitsDef create_emits (
            in RepositoryId id,
            in Identifier name,
            in VersionSpec version,
            in EventDef event
        );

        PublishesDef create_publishes (
            in RepositoryId id,
            in Identifier name,
            in VersionSpec version,
            in EventDef event
        );

        ConsumesDef create_consumes (
            in RepositoryId id,
            in Identifier name,
            in VersionSpec version,
            in EventDef event
        );
    };

    typedef sequence<ProvidesDescription>
                                        ProvidesDescriptionSeq;
    typedef sequence<UsesDescription> UsesDescriptionSeq;
    typedef sequence<EventPortDescription>
                                        EventPortDescriptionSeq;

    struct ComponentDescription {
        Identifier name;
        RepositoryId id;
        RepositoryId defined_in;
        VersionSpec version;
        RepositoryId base_component;
        RepositoryIdSeq supported_interfaces;
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            ProvidesDescriptionSeq provided_interfaces;
            UsesDescriptionSeq used_interfaces;
            EventPortDescriptionSeq emits_events;
            EventPortDescriptionSeq publishes_events;
            EventPortDescriptionSeq consumes_events;
            ExtAttrDescriptionSeq attributes;
            TypeCode type;
        };
    };
};
```

14.6.10.1 Read Interface

The **base_component** attribute returns the component that this component derives from.

The **supported_interfaces** attribute lists the interfaces that this component type supports.

The inherited **is_a** operation returns **TRUE** if the component on which it is invoked either is identical to or inherits from the component identified by its **id** parameter. Otherwise it returns **FALSE**.

The inherited **describe** operation for a **ComponentDef** returns a **ComponentDescription**.

The inherited **contents** operation returns the list of attributes, provides, uses, emits, publishes, and consumes either defined or inherited in this **ComponentDef**. If the **exclude_inherited** parameter is set to **TRUE**, only attributes, provides, uses, emits, publishes, and consumes defined within this object are returned. If the **exclude_inherited** parameter is set to **FALSE**, all attributes, provides, uses, emits, publishes, and consumes are returned.

14.6.10.2 Write Interface

Setting the **base_component** attribute causes a BAD_PARAM exception with minor code 5 to be raised if the **name** attribute of any object contained by this **ComponentDef** conflicts with the **name** attribute of any object contained by the specified base **ComponentDef**.

Setting the **supported_interfaces** attribute changes the interfaces that this component type supports.

The **create_<type>** operations defined in the **ComponentIR::ComponentDef** interface create new corresponding empty IR objects. The **defined_in** attribute is initialized to identify the containing **ComponentDef**, and the **containing_repository** attribute is initialized to its **ComponentIR::Repository**.

These **create_<type>** operations all take **id** and **name** parameters that are used to initialize the identity of the created definition. A BAD_PARAM exception is raised with minor code 2 if an object with the specified **id** already exists in the **ComponentIR::Repository**. A BAD_PARAM exception with minor code 3 is raised if the specified **name** already exists within this **ComponentDef** and multiple versions are not supported.

The inherited **create_ext_attribute** operation returns a new **ExtAttributeDef** contained in the **ComponentDef** on which it is invoked. The **id**, **name**, **version**, **type_def**, **mode, get_exceptions,** and **set_exceptions** attributes are set as specified. The **type** attribute is also set.

The inherited **create_operation**, and all other **create_*** operations inherited from **Container** and **Contained** return BAD_PARAM exception with minor code 4.

The **create_provides** operation returns a new **ProvidesDef** contained in the **ComponentDef** on which it is invoked. The **id**, **name**, **version**, and **interface_type** attributes are set as specified.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The **create_uses** operation returns a new **UsesDef** contained in the **ComponentDef** on which it is invoked. The **id**, **name**, **version**, **interface_type**, and **is_multiple** attributes are set as specified.

The **create_emits**, **create_publishes**, and **create_consumes** operations respectively return new **EmitsDef**, **PublishesDef**, and **ConsumesDef** contained in the **ComponentDef** on which it is invoked. The **id**, **name**, **version**, and **event** attributes are set as specified.

A **ComponentDef** used as a **Container** may not contain any **TypedefDef** (including definitions derived from **TypedefDef)**, **ConstantDef**, or **ExceptionDef** definitions.

A **ComponentDef** used as an **InterfaceDef** may only contain **ExtAttributeDef** definitions.

14.6.11 ComponentIR::FactoryDef

A **ComponentIR::FactoryDef** object represents the definition of a factory operation in a home.

```
module CORBA {
    module ComponentIR {

        interface FactoryDef : OperationDef { // only PARAM_IN parameters
        };
    };
};
```

14.6.11.1 Read Interface

The **result** attribute is a **TypeCode** describing the type of the value returned by the operation, which is always **tk_component** for **FactoryDef**. The **result_def** attribute identifies the definition of the returned type, which is always a **ComponentDef** in case of **FactoryDef**.

The **params** attribute describes the parameters of the operation. It is a sequence of **ParameterDescription** structures. The order of the **ParameterDescription**s in the sequence is significant. The **name** member of each structure provides the parameter name. The **type** member is a **TypeCode** describing the type of the parameter. The **type_def** member identifies the definition of the type of the parameter. The **mode** member indicates whether the parameter is an in, out, or inout parameter. For **FactoryDef** the value of mode for all parameters is **PARAM_IN**.

The operation's **mode** is always **normal** for **FactoryDef**.

The **kind** attribute is always **OP_IDL** for **FactoryDef**.

The **contexts** attribute specifies the list of context identifiers that apply to the operation, and is an empty list for **FactoryDef**.

The **exceptions** attribute specifies the list of exception types that can be raised by the operation.

The inherited **describe** operation for a **FactoryDef** object returns an **OperationDescription**.

14.6.11.2 Write Interface

Setting the **result_def** attribute has no effect.

The **mode** and **contexts** attributes cannot be changed.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.6.12 ComponentIR::FinderDef

A **ComponentIR::FinderDef** object represents the definition of a finder operation in a home.

```
module CORBA {
    module ComponentIR {

        interface FinderDef : OperationDef { // only PARAM_IN parameters
        };
    };
};
```

14.6.12.1 Read Interface

The **result** attribute is a **TypeCode** describing the type of the value returned by the operation, which is always **tk_component** for **FinderDef**. The **result_def** attribute identifies the definition of the returned type, which is always a **ComponentDef** in case of a **FinderDef**.

The **params** attribute describes the parameters of the operation. It is a sequence of **ParameterDescription** structures. The order of the **ParameterDescription**s in the sequence is significant. The **name** member of each structure provides the parameter name. The **type** member is a **TypeCode** describing the type of the parameter. The **type_def** member identifies the definition of the type of the parameter. The **mode** member indicates whether the parameter is an in, out, or inout parameter. For **FinderDef** the value of mode for all parameters is **PARAM_IN**.

The operation's **mode** is always **normal** for **FinderDef**.

The **kind** attribute is always **OP_IDL** for **FinderDef**.

The **contexts** attribute specifies the list of context identifiers that apply to the operation, and is an empty list for **FinderDef**.

The **exceptions** attribute specifies the list of exception types that can be raised by the operation.

The inherited **describe** operation for a **FinderDef** object returns an **OperationDescription**.

14.6.12.2 Write Interface

Setting the **result_def** attribute has no effect.

The **mode** and **contexts** attributes cannot be changed.

14.6.13 ComponentIR::HomeDef

A **ComponentIR::HomeDef** object represents the definition of a home. It contains attributes, operations, factories, and finders.

```
module CORBA {
    module ComponentIR {

        interface HomeDef : ExtInterfaceDef {
            // read/write interface
            attribute HomeDef base_home;
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            attribute InterfaceDefSeq supported_interfaces;
            attribute ComponentDef managed_component;
            attribute ValueDef primary_key;

            // write interface
            FactoryDef create_factory (
                in RepositoryId id,
                in Identifier name,
                in VersionSpec version,
                in ParDescriptionSeq params,
                in ExceptionDefSeq exceptions
            );

            FinderDef create_finder (
                in RepositoryId id,
                in Identifier name,
                in VersionSpec version,
                in ParDescriptionSeq params,
                in ExceptionDefSeq exceptions
            );
        };

        struct HomeDescription {
            Identifier name;
            RepositoryId id;
            RepositoryId defined_in;
            VersionSpec version;
            RepositoryId base_home;
            RepositoryId managed_component;
            ValueDescription primary_key;
            OpDescriptionSeq factories;
            OpDescriptionSeq finders;
            OpDescriptionSeq operations;
            ExtAttrDescriptionSeq attributes;
            TypeCode type;
        };
    };
};
```

14.6.13.1 Read Interface

The **base_home** attribute returns the home that this home definition derives from.

The **supported_interfaces** attribute lists the interfaces that this home type supports.

The **managed_component** attribute returns the component that this home manages.

The **primary_key** attribute returns the primary key that is associated with this home.

The inherited **is_a** operation returns **TRUE** if the home on which it is invoked either is identical to or inherits from the home identified by its **id** parameter. Otherwise it returns **FALSE**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The inherited **describe** operation for a **HomeDef** returns a **HomeDescription**.

The inherited **contents** operation returns the list of constants, typedefs, exceptions, attributes, operations, finders, and factories defined or inherited in this **HomeDef**. If the **exclude_inherited** parameter is set to **TRUE**, only objects defined within this home are returned. If the **exclude_inherited** parameter is set to **FALSE**, all objects are returned.

14.6.13.2 Write Interface

Setting the **base_home** attribute causes a BAD_PARAM exception with minor code 5 to be raised if the **name** attribute of any object contained by this **HomeDef** conflicts with the **name** attribute of any object contained by the specified base **HomeDef**.

The **create_<type>** operations defined in the **HomeDef** interface create new corresponding empty IR objects. The **defined_in** attribute is initialized to identify the containing **HomeDef**, and the **containing_repository** attribute is initialized to its **ComponentIR::Repository**.

These **create_<type>** operations all take **id** and **name** parameters that are used to initialize the identity of the created definition. A BAD_PARAM exception is raised with minor code 2 if an object with the specified **id** already exists in the **ComponentIR::Repository**. A BAD_PARAM exception with minor code 3 is raised if the specified **name** already exists within this **HomeDef** and multiple versions are not supported.

The inherited **create_ext_attribute** operation returns a new **ExtAttributeDef** contained in the **HomeDef** on which it is invoked. The **id**, **name, version**, **type_def**, **mode, get_exceptions**, and **set_exceptions** attributes are set as specified. The **type** attribute is also set.

The inherited **create_operation** operation returns a new **OperationDef** contained in the **HomeDef** on which it is invoked. The **id**, **name**, **version**, **result_def**, **mode**, **params**, **exceptions**, and **contexts** attributes are set as specified. The **result** attribute is also set.

The **create_factory** operation returns a new **FactoryDef** contained in the **HomeDef** on which it is invoked. The **id**, **name, version, params**, and **exceptions** attributes are set as specified. The parameters in the **params** attribute must all be of **PARAM_IN** type.

The **create_finder** operation returns a new **FinderDef** contained in the **HomeDef** on which it is invoked. The **id**, **name, versions, params**, and **exceptions** attributes are set as specified. The parameters in the **params** attribute must all be of **PARAM_IN** type.

A **HomeDef** used as a **Container** may only contain **TypedefDef** (including definitions derived from **TypedefDef)**, **ConstantDef**, and **ExceptionDef** definitions.

14.7 RepositoryIds

RepositoryIds are values that can be used to establish the identity of information in the repository. A **RepositoryId** is represented as a string, allowing programs to store, copy, and compare them without regard to the structure of the value. It does not matter what format is used for any particular **RepositoryId**. However, conventions are used to manage the name space created by these IDs.

RepositoryIds may be associated with OMG IDL definitions in a variety of ways. Installation tools might generate them, they might be defined with pragmas in OMG IDL source, or they might be supplied with the package to be installed. Ensuring consistency of **RepositoryId**s with the IDL source or the IR contents is the responsibility of the programmer allocating **Repositoryid**s.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The format of the id is a short format name followed by a colon (":") followed by characters according to the format. This specification defines four formats:

1. one derived from OMG IDL names,
2. one that uses Java class names and Java serialization version UIDs,
3. one that uses DCE UUIDs, and
4. another intended for short-term use, such as in a development environment.

Since new repository ID formats may be added from time to time, compliant IDL compilers must accept any string value of the form

"<format>:<string>"

provided as the argument to the id pragma and use it as the repository ID. The OMG maintains a registry of allocated format identifiers. The **<format>** part of the ID may not contain a colon (:) character.

The version and prefix pragmas only affect default repository IDs that are generated by the IDL compiler using the IDL format.

14.7.1 OMG IDL Format

The OMG IDL format for **RepositoryIds** primarily uses OMG IDL scoped names to distinguish between definitions. It also includes an optional unique prefix, and major and minor version numbers.

The **RepositoryId** consists of three components, separated by colons, (":")

1. The first component is the format name, "IDL."
2. (".").The second component is a list of identifiers, separated by "/" characters. These identifiers are arbitrarily long sequences of alphabetic, digit, underscore ("_"), hyphen ("-"), and period (".") characters. Typically, the first identifier is a unique prefix, and the rest are the OMG IDL Identifiers that make up the scoped name of the definition. The second component shall not contain a trailing slash ("/") and it shall not begin with the characters underscore ("_"), hyphen ("-"), or period (".").
3. The third component is made up of major and minor version numbers, in decimal format, separated by a ".". When two interfaces have **RepositoryId**s differing only in minor version number it can be assumed that the definition with the higher version number is upwardly compatible with (i.e., can be treated as derived from) the one with the lower minor version number.

14.7.2 RMI Hashed Format

The OMG IDL format defined above does not include any structural information. Identity of IDL types determined for this format depends upon the names used in the **RepositoryID** being correct. For interfaces, if stubs and skeletons are not actually in synch, even though the **RepositoryIds** report they are, the worst that can happen is that the result of an invocation is a BAD_OPERATION exception. With value types, these kinds of errors are more problematic. An inconsistency between the stub and skeleton marshaling/unmarshaling code can confuse the marshaling engine and may even corrupt memory and/or cause a crash failure.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The RMI Hashed format is used for Java RMI values mapped to IDL using the Java to IDL Mapping (see the *Java/IDL Language Mapping* document). It is computed based upon the structural information of the original Java definition. Whenever the Java definition changes, the hash function will (statistically) produce a hash code, which is different from the previous one. When an ORB run time receives a **value** with a different hash from what is expected, it is free to raise a BAD_PARAM exception. It may also try to resolve the incompatibility by some means. If it is not successful, then it shall raise the BAD_PARAM exception.

An RMI Hashed **RepositoryId** consists of either three or four components, separated by colons:

RMI: <class name> : <hash code> [: <serialization version UID>]

The class name is a Java class name as returned by the **`getName`** method of **`java.lang.Class`**. Any characters not in *ISO Latin 1* are replaced by "**\U**" followed by the 4 hexadecimal characters (in upper case) representing the *Unicode* value.

For classes that do not implement **`java.io.Serializable`**, and for interfaces, the hash code is always zero, and the **RepositoryID** does not contain a *serial version UID*.

For classes that implement **`java.io.Externalizable`**, the hash code is always the *64-bit value 1*.

For classes that implement **`java.io.Serializable`** but not **`java.io.Externalizable`**, the hash code is a *64-bit hash of a stream of bytes* (transcribed as a 16-digit upper case hex string). An instance of **`java.lang.DataOutputStream`** is used to convert primitive data types to a sequence of bytes. The sequence of items in the stream is as follows:

1. The hash code of the superclass, written as a 64-bit long.
2. The value 1 if the class has no **`writeObject`** method, or the value 2 if the class has a **`writeObject`** method, written as a 32-bit integer.
3. For each field of the class that is mapped to IDL, sorted lexicographically by Java field name, in increasing order:
   a. Java field name, in *UTF encoding*
   b. field descriptor, as defined by the *Java Virtual Machine Specification*, in *UTF encoding*.

The *National Institute of Standards and Technology* (*NIST*) *Secure Hash Algorithm* (*SHA-1*) is executed on the stream of bytes produced by **`DataOutputStream`**, producing a *20 byte array of values, sha[0..19*]. The hash code is assembled from the *first 8 bytes* of this array as follows:

```
          long hash = 0;
for (int i = 0; i < Math.min(8, sha.length); i++) {
      hash += (long)(sha[i] & 255) << (i * 8);
}
```

For Serializable (including Externalizable) classes, the Java serialization version **`UID`**, transcribed as a 16 digit upper-case hex string, shall be appended to the **`RepositoryId`** following the hash code and a colon. The Java serialization version **`UID`** is defined in the Java Object Serialization Specification.

Examples for the valuetype **`::foo::bar`** would be

```
RMI:foo/bar;:1234567812345678
RMI:foo/bar;:1234567812345678:ABCD123456781234
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

An example of a Java array of valuetype **::foo::bar** would be

RMI:[Lfoo.bar;:1234567812345678:ABCD123456781234

For a Java class **x\u03bCy** that contains a Unicode character not in ISO Latin 1, an example **RepositoryId** is

RMI:foo.x\U03BCy:8765432187654321

A conforming implementation that uses this format shall implement the standard hash algorithm defined above.

14.7.3 DCE UUID Format

DCE UUID format **RepositoryId**s start with the characters “DCE:” and are followed by the printable form of the UUID, a colon, and a decimal minor version number, for example: “DCE:700dc518-0110-11ce-ac8f-0800090b5d3e:1.”

14.7.4 LOCAL Format

Local format **RepositoryId**s start with the characters “LOCAL:” and are followed by an arbitrary string. Local format IDs are not intended for use outside a particular repository, and thus do not need to conform to any particular convention. Local IDs that are just consecutive integers might be used within a development environment to have a very cheap way to manufacture the IDs while avoiding conflicts with well-known interfaces.

14.7.5 Pragma Directives for RepositoryId

Three pragma directives (id, prefix, and version), are specified to accommodate arbitrary **RepositoryId** formats and still support the OMG IDL **RepositoryId** format with minimal annotation. The prefix and version pragma directives apply only to the IDL format. An IDL compiler must interpret these annotations as specified. Conforming IDL compilers may support additional non-standard pragmas, but must not refuse to compile IDL source containing non-standard pragmas that are not understood by the compiler.

14.7.5.1 The ID Pragma

An OMG IDL pragma of the format

#pragma ID <name> “<id>”

associates an arbitrary **RepositoryId** string with a specific OMG IDL name. The **<name>** can be a fully or partially scoped name or a simple identifier, interpreted according to the usual OMG IDL name lookup rules relative to the scope within which the pragma is contained. The **<id>** must be a repository ID of the form described in RepositoryIds on page 274.

An attempt to assign a repository ID to the same IDL construct a second time shall be an error unless the repository ID used in the attempt is identical to the previous one.

```
interface A {};
#pragma ID A “IDL:A:1.1”
#pragma ID A “IDL:X:1.1”       // Compile-time error

interface B {};
#pragma ID B “IDL:BB:1.1”
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
#pragma ID B “IDL:BB:1.1”      // OK, same ID
```

It is also an error to apply an ID to a forward-declared IDL construct (interface, valuetype, structure, and union) and then later assign a different ID to that IDL construct.

14.7.5.2 The Prefix Pragma

An OMG IDL pragma is of the form:

#pragma prefix “<string>”

This sets the current prefix used in generating OMG IDL format **RepositoryId**s. For example, the **RepositoryId** for the initial version of interface **Printer** defined on module **Office** by an organization known as “SoftCo” might be “IDL:SoftCo/Office/Printer:1.0.”

Since the “prefix” pragma applies to Repository Ids of the IDL format, the <string> above shall be a list of one or more identifiers, separated by the “/” characters. These identifiers are arbitrarily long sequences of alphabetic, digit, underscore (“_”), hyphen (“-”), and period (“.”) characters. The string shall not contain a trailing slash ("/") and it shall not begin with the characters underscore ("_"), hyphen ("-"), or period (".").

This format makes it convenient to generate and manage a set of IDs for a collection of OMG IDL definitions. The person creating the definitions sets a prefix (“SoftCo”), and the IDL compiler or other tool can synthesize all the needed IDs.

Because **RepositoryId**s may be used in many different computing environments and ORBs, as well as over a long period of time, care must be taken in choosing them. Prefixes that are distinct, such as trademarked names, domain names, UUIDs, and so forth, are preferable to generic names such as “document.”

The specified prefix applies to **RepositoryId**s generated after the pragma until the end of the current scope is reached or another prefix pragma is encountered. An IDL file forms a scope for this purpose, so a prefix resets to the previous prefix at the end of the scope of an included file:

```
// A.idl
#pragma prefix “A”
interface A {};

// B.idl
#pragma prefix “B”
#include “A.idl”
interface B {};
```

The repository IDs for interfaces A and B in this case are:

```
IDL:A/A:1.0
IDL:B/B:1.0
```

Similarly, a prefix in an including file does not affect the prefix of an included file:

```
// C.idl
interface C {};

// D.idl
#pragma prefix “D”
#include “C.idl”
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
interface D {};
```

The repository IDs for interface C and D in this case are:

```
IDL:C:1.0
IDL:D/D:1.0
```

If an included file does not contain a #pragma prefix, the current prefix implicitly resets to the empty prefix:

```
// E.idl
interface E {};

// F.idl
module M {
  #include <E.idl>
};
```

The repository IDs for module M and interface E in this case are:

```
IDL:M:1.0
IDL:E:1.0
```

If a #include directive appears at non-global scope and the included file contains a prefix pragma, the included file's prefix takes precedence, for example:

```
// A.idl
#pragma prefix "A"
interface A {};

// B.idl
#pragma prefix "B"
module M {
#include "A.idl"
};
```

The repository ID for module M and interface A in this case are:

```
IDL:B/M:1.0
IDL:A/A:1.0
```

Forward-declared constructs (interfaces, value types, structures, and unions) must have the same prefix in effect wherever they appear. Attempts to assign conflicting prefixes to a forward-declared construct result in a compile-time diagnostic. For example:

```
#pragma prefix "A"
interface A;        // Forward decl.

#pragma prefix "B"
interface A;        // Compile-time error
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
#pragma prefix “C”
interface A {       // Compile-time error
    void op();
};
```

A prefix pragma of the form

```
#pragma prefix “”
```

resets the prefix to the empty string. For example:

```
#pragma prefix “X”
interface X {};
#pragma prefix “”
interface Y {};
```

The repository IDs for interface X and Y in this case are:

```
IDL:X/X:1.0
IDL:Y:1.0
```

If a specification contains both a prefix pragma and an ID or version pragma, the prefix pragma does not affect the repository ID for an ID pragma, but does affect the repository ID for a version pragma:

```
#pragma prefix “A”
interface A {};
interface B {};
interface C {};
#pragma ID B “IDL:myB:1.0”
#pragma version C 9.9
```

The repository IDs for this specification are:

```
IDL:A/A:1.0
IDL:myB:1.0
IDL:A/C:9.9
```

A #pragma prefix must appear before the beginning of an IDL definition. Placing a #pragma prefix elsewhere has undefined behavior, for example:

```
interface Bar
    #pragma prefix “foo”    // Undefined behavior
    {
    // ...
};
```

14.7.5.3 The Version Pragma

An OMG IDL pragma of the format:

#pragma version <name> <major>.<minor>

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

provides the version specification used in generating an OMG IDL format **RepositoryId** for a specific OMG IDL name. The **<name>** can be a fully or partially scoped name or a simple identifier, interpreted according to the usual OMG IDL name lookup rules relative to the scope within which the pragma is contained. The **<major>** and **<minor>** components are decimal unsigned shorts.

If no version pragma is supplied for a definition, version 1.0 is assumed. If an attempt is made to change the version of a repository ID that was specified with an ID pragma, a compliant compiler shall emit a diagnostic:

```
interface A {};
#pragma ID A “IDL:myA:1.1”
#pragma version A 9.9          // Compile-time error
```

An attempt to assign a version to the same IDL construct a second time shall be an error unless the version used in the attempt is identical to the existing one.

```
interface A {};
#pragma version A 1.1
#pragma version A 1.1          // OK
#pragma version A 1.2          // Error

interface B {};
#pragma ID B “IDL:myB:1.2”
#pragma version B 1.2          // OK
```

14.7.5.4 Generation of OMG IDL - Format IDs

A definition is globally identified by an OMG IDL - format **RepositoryId** if no ID pragma is encountered for it.

The ID string shall be generated by starting with the string “IDL:”. Then, if the current prefix pragma is a non-empty string, it is appended, followed by a “/” character. Next, the components of the scoped name of the definition, relative to the scope in which any prefix that applies was encountered, are appended, separated by “/” characters. Finally, a “:” and the version specification are appended.

For example, the following OMG IDL:

```
module M1 {
    typedef long T1;
    typedef long T2;
    #pragma ID T2 “DCE:d62207a2-011e-11ce-88b4-0800090b5d3e:3”
};

#pragma prefix “P1”

module M2 {
    module M3 {
        #pragma prefix “P2”
        typedef long T3;
    };
    typedef long T4;
    #pragma version T4 2.4
};
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

specifies types with the following scoped names and **RepositoryId**s:

::M1::T1	IDL:M1/T1:1.0
::M1::T2	DCE:d62207a2-011e-11ce-88b4-0800090b5d3e:3
::M2::M3::T3	IDL:P2/T3:1.0
::M2::T4	IDL:P1/M2/T4:2.4

For this scheme to provide reliable global identity, the prefixes used must be unique. Two non-colliding options are suggested: Internet domain names and DCE UUIDs.

Furthermore, in a distributed world where different entities independently evolve types, a convention must be followed to avoid the same **RepositoryId** being used for two different types. Only the entity that created the prefix has authority to create new IDs by simply incrementing the version number. Other entities must use a new prefix, even if they are only making a minor change to an existing type.

Prefix pragmas can be used to preserve the existing IDs when a module or other container is renamed or moved.

```
module M4 {
#pragma prefix “P1/M2”
    module M3 {
#pragma prefix “P2”
        typedef long T3;
    };
    typedef long T4;
#pragma version T4 2.4
};
```

This OMG IDL declares types with the same global identities as those declared in module M2 above.

See The Prefix Pragma on page 278 for further details of the effects of various prefix pragma settings on the generated **RepositoryId**s.

14.7.6 For More Information

OMG IDL for Interface Repository on page 284 shows the OMG IDL specification of the IR, including the #pragma directive. Preprocessing on page 36 contains additional, general information on the pragma directive.

14.7.7 RepositoryIDs for OMG-Specified Types

Interoperability between implementations of official OMG specifications, including but not limited to CORBA, CORBA Services, and CORBA Facilities, depends on unambiguous specification of **RepositoryId**s for all IDL-defined types in such specifications.

All official OMG IDL files shall contain the following pragma prefix directive:

```
#pragma prefix “omg.org”
```

unless said file already contains a pragma prefix identifying the original source of the file (e.g., “**w3c.org**”).

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Revisions to existing OMG specifications must not change the definition of an existing type in any way. Two types with different repository Ids are considered different types, regardless of which part of the repository Id differs.

If an implementation must extend an OMG-specified interface, interoperability requires it to derive a new interface from the standard interface, rather than modify the standard definition.

14.7.8 Uniqueness Constraints on Repository IDs

Within an IDL definition, a module must have the same repository ID throughout. For example:

```
#pragma prefix "A"
module M {
    // ...
};

#pragma prefix "B"
module M {          // Error, inconsistent repository ID
    // ...
};
```

This definition attempts to use the same type name M with two different repository IDs in the same compilation unit. Compilers shall issue a diagnostic for this error.

The same error can arise through inclusion of source files in the same compilation unit. For example:

```
// File1.idl
module M {
    module N {
        // ...
    };
#pragma ID N "abc"
};

// File2.idl
module M {
    module N {
        // ...
    };
};

// File3.idl
#include "File1.idl
#include "File2.idl    // Error, inconsistent repository ID
```

Similarly:

```
// File1.idl
 module M {
     // ...
 };
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
// File2.idl
#include File1.idl
#pragma prefix "X"
module M {          // Error, inconsistent repository ID
    // ...
};
```

Such errors are detectable only if they occur in a single compilation unit (or in files included in a single compilation unit); if, in different compilation units, different repository IDs are used for the same module, and these compilation units are combined into a single executable, the behavior is undefined.

14.8 OMG IDL for Interface Repository

This sub clause contains the complete OMG IDL specification for the Interface Repository.

```
module CORBA {
    typeprefix CORBA “omg.org”;
    typedef string Identifier;
    typedef string ScopedName;
    typedef string RepositoryId;

    enum DefinitionKind {
        dk_none, dk_all,
        dk_Attribute, dk_Constant, dk_Exception, dk_Interface,
        dk_Module, dk_Operation, dk_Typedef,
        dk_Alias, dk_Struct, dk_Union, dk_Enum,
        dk_Primitive, dk_String, dk_Sequence, dk_Array,
        dk_Repository,
        dk_Wstring, dk_Fixed,
        dk_Value, dk_ValueBox, dk_ValueMember,
        dk_Native,
        dk_AbstractInterface,
        dk_LocalInterface
        dk_Component, dk_Home,
        dk_Factory, dk_Finder,
        dk_Emits, dk_Publishes, dk_Consumes,
        dk_Provides, dk_Uses,
        dk_Event
    };

    interface IRObject {
        // read interface
        readonly attribute DefinitionKind def_kind;
        // write interface
        void destroy ();
    };

    typedef string VersionSpec;

    interface Contained;
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
interface Repository;
interface Container;

interface Contained : IRObject {

    // read/write interface

    attribute RepositoryId id;
    attribute Identifier name;
    attribute VersionSpec version;

    // read interface

    readonly attribute Container defined_in;
    readonly attribute ScopedName absolute_name;
    readonly attribute Repository containing_repository;

    struct Description {
        DefinitionKind kind;
        any value;
    };

    Description describe ();

    // write interface

    void move (
        in Container        new_container,
        in Identifier       new_name,
        in VersionSpec      new_version
    );
};

interface ModuleDef;
interface ConstantDef;
interface IDLType;
interface StructDef;
interface UnionDef;
interface EnumDef;
interface AliasDef;
interface InterfaceDef;
interface ExceptionDef;
interface NativeDef;
typedef sequence <InterfaceDef> InterfaceDefSeq;
interface ValueDef;
typedef sequence <ValueDef> ValueDefSeq;
interface ValueBoxDef;
interface AbstractInterfaceDef;
typedef sequence <AbstractInterfaceDef> AbstractInterfaceDefSeq;
interface LocalInterfaceDef;
typedef sequence <LocalInterfaceDef> LocalInterfaceDefSeq;
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
interface ExtInterfaceDef;
typedef sequence <ExtInterfaceDef> ExtInterfaceDefSeq;
interface ExtValueDef;
typedef sequence <ExtValueDef> ExtValueDefSeq;
interface ExtAbstractInterfaceDef;
typedef sequence <ExtAbstractInterfaceDef>
    ExtAbstractInterfaceDefSeq;
interface ExtLocalInterfaceDef;
typedef sequence <ExtLocalInterfaceDef>
    ExtLocalInterfaceDefSeq;

typedef sequence <Contained> ContainedSeq;
struct StructMember {
    Identifier          name;
    TypeCode            type;
    IDLType             type_def;
};

typedef sequence <StructMember> StructMemberSeq;

struct Initializer {
    StructMemberSeq  members;
    Identifier          name;
};
typedef sequence <Initializer> InitializerSeq;

struct ExceptionDescription {
    Identifier              name;
    RepositoryId            id;
    RepositoryId            defined_in;
    VersionSpec             version;
    TypeCode                type;
};
typedef sequence <ExceptionDescription> ExcDescriptionSeq;

struct ExtInitializer {
    StructMemberSeq     members;
    ExcDescriptionSeq   exceptions;
    Identifier              name;
};
typedef sequence <ExtInitializer> ExtInitializerSeq;

 struct UnionMember {
    Identifier      name;
    any             label;
    TypeCode        type;
    IDLType         type_def;
};

typedef sequence <UnionMember> UnionMemberSeq;
```


--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
typedef sequence <Identifier> EnumMemberSeq;

interface Container : IRObject {
    // read interface

    Contained lookup (
        in ScopedName       search_name);

    ContainedSeq contents (
        in DefinitionKind   limit_type,
        in boolean          exclude_inherited
    );

    ContainedSeq lookup_name (
        in Identifier       search_name,
        in long             levels_to_search,
        in DefinitionKind   limit_type,
        in boolean          exclude_inherited
    );

    struct Description {
        Contained           contained_object;
        DefinitionKind      kind;
        any                 value;
    };

    typedef sequence<Description> DescriptionSeq;

    DescriptionSeq describe_contents (
        in DefinitionKind   limit_type,
        in boolean          exclude_inherited,
        in long             max_returned_objs
    );

    // write interface

    ModuleDef create_module (
        in RepositoryId     id,
        in Identifier       name,
        in VersionSpec      version
    );

    ConstantDef create_constant (
        in RepositoryId     id,
        in Identifier       name,
        in VersionSpec      version,
        in IDLType          type,
        in any              value
    );

    StructDef create_struct (
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        in RepositoryId         id,
        in Identifier           name,
        in VersionSpec          version,
        in StructMemberSeq      members
    );

    UnionDef create_union (
        in RepositoryId         id,
        in Identifier           name,
        in VersionSpec          version,
        in IDLType              discriminator_type,
        in UnionMemberSeq       members
    );

    EnumDef create_enum (
        in RepositoryId         id,
        in Identifier           name,
        in VersionSpec          version,
        in EnumMemberSeq        members
    );

    AliasDef create_alias (
        in RepositoryId         id,
        in Identifier           name,
        in VersionSpec          version,
        in IDLType              original_type
    );

    InterfaceDef create_interface (
        in RepositoryId         id,
        in Identifier           name,
        in VersionSpec          version,
        in InterfaceDefSeq      base_interfaces,
    );

    ValueDef create_value(
        in RepositoryId         id,
        in Identifier           name,
        in VersionSpec          version,
        in boolean              is_custom,
        in boolean              is_abstract,
        in ValueDef             base_value,
        in boolean              is_truncatable,
        in ValueDefSeq          abstract_base_values,
        in InterfaceDefSeq      supported_interfaces,
        in InitializerSeq       initializers
    );

    ValueBoxDef create_value_box(
        in RepositoryId         id,
        in Identifier           name,
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        in VersionSpec          version,
        in IDLType              original_type_def
    );

    ExceptionDef create_exception(
        in RepositoryId         id,
        in Identifier           name,
        in VersionSpec          version,
        in StructMemberSeq      members
    );

    NativeDef create_native(
        in RepositoryId         id,
        in Identifier           name,
        in VersionSpec          version
    );

    AbstractInterfaceDef create_abstract_interface (
        in RepositoryId id,
        in Identifier name,
        in VersionSpec version,
        in AbstractInterfaceDefSeq base_interfaces,
    );

    LocalInterfaceDef create_local_interface (
        in RepositoryId id,
        in Identifier name,
        in VersionSpec version,
        in InterfaceDefSeq base_interfaces
    );

    ExtValueDef create_ext_value (
        in RepositoryId         id,
        in Identifier           name,
        in VersionSpec          version,
        in boolean              is_custom,
        in boolean              is_abstract,
        in ValueDef             base_value,
        in boolean              is_truncatable,
        in ValueDefSeq          abstract_base_values,
        in InterfaceDefSeq      supported_interfaces,
        in ExtInitializerSeq    initializers
    );
};

interface IDLType : IRObject {
    readonly attribute TypeCode type;
};

interface PrimitiveDef;
interface StringDef;
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
interface SequenceDef;
interface ArrayDef;
interface WstringDef;
interface FixedDef;

enum PrimitiveKind {
    pk_null, pk_void, pk_short, pk_long, pk_ushort, pk_ulong,
    pk_float, pk_double, pk_boolean, pk_char, pk_octet,
    pk_any, pk_TypeCode, pk_Principal, pk_string, pk_objref,
    pk_longlong, pk_ulonglong, pk_longdouble,
    pk_wchar, pk_wstring, pk_value_base
};

interface Repository : Container {
    // read interface

    Contained lookup_id (in RepositoryId search_id);

    TypeCode get_canonical_typecode(in TypeCode tc);

    PrimitiveDef get_primitive (in PrimitiveKind kind);

    // write interface

    StringDef create_string (in unsigned long bound);

    WstringDef create_wstring (in unsigned long bound);

    SequenceDef create_sequence (
        in unsigned long      bound,
        in IDLType            element_type
    );

    ArrayDef create_array (
        in unsigned long      length,
        in IDLType            element_type
    );

    FixedDef create_fixed (
        in unsigned short     digits,
        in short              scale
    );
};

interface ModuleDef : Container, Contained {
};

struct ModuleDescription {
    Identifier          name;
    RepositoryId        id;
    RepositoryId        defined_in;
```


© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    VersionSpec         version;
};

interface ConstantDef : Contained {
    readonly attribute TypeCode type;
    attribute IDLType type_def;
    attribute any value;
};

struct ConstantDescription {
    Identifier          name;
    RepositoryId        id;
    RepositoryId        defined_in;
    VersionSpec         version;
    TypeCode            type;
    any                 value;
};
interface TypedefDef : Contained, IDLType {
};

struct TypeDescription {
    Identifier              name;
    RepositoryId            id;
    RepositoryId            defined_in;
    VersionSpec             version;
    TypeCode                type;
};

interface StructDef : TypedefDef, Container {
    attribute StructMemberSeq   members;
};

interface UnionDef : TypedefDef, Container {
    readonly attribute TypeCode discriminator_type;
    attribute IDLType           discriminator_type_def;
    attribute UnionMemberSeq    members;
};

interface EnumDef : TypedefDef {
    attribute EnumMemberSeq     members;
};

interface AliasDef : TypedefDef {
    attribute IDLType           original_type_def;
};

interface NativeDef : TypedefDef {
};
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
interface PrimitiveDef: IDLType {
    readonly attribute PrimitiveKind kind;
};

interface StringDef : IDLType {
    attribute unsigned long         bound;
};

interface WstringDef : IDLType {
    attribute unsigned long         bound;
};

interface FixedDef : IDLType {
    attribute unsigned short        digits;
    attribute short                 scale;
};

interface SequenceDef : IDLType {
    attribute unsigned long         bound;
    readonly attribute TypeCode     element_type;
    attribute IDLType               element_type_def;
};

interface ArrayDef : IDLType {
    attribute unsigned long         length;
    readonly attribute TypeCode     element_type;
    attribute IDLType               element_type_def;
};

interface ExceptionDef : Contained, Container {
    readonly attribute TypeCode     type;
    attribute StructMemberSeq       members;
};

enum AttributeMode {ATTR_NORMAL, ATTR_READONLY};

interface AttributeDef : Contained {
    readonly attribute TypeCode     type;
    attribute IDLType               type_def;
    attribute AttributeMode         mode;
};

struct AttributeDescription {
    Identifier              name;
    RepositoryId            id;
    RepositoryId            defined_in;
    VersionSpec             version;
    TypeCode                type;
    AttributeMode           mode;
};
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
struct ExtAttributeDescription {
    Identifier          name;
    RepositoryId        id;
    RepositoryId        defined_in;
    VersionSpec         version;
    TypeCode            type;
    AttributeMode       mode;
    ExcDescriptionSeq   get_exceptions;
    ExcDescriptionSeq   put_exceptions;
};

interface ExtAttributeDef : AttributeDef {

    // read/write interface
    attribute ExcDescriptionSeq get_exceptions;
    attribute ExcDescriptionSeq set_exceptions;

    // read interface
    ExtAttributeDescription describe_attribute ();
};

enum OperationMode {OP_NORMAL, OP_ONEWAY};
enum ParameterMode {PARAM_IN, PARAM_OUT, PARAM_INOUT};

struct ParameterDescription {
    Identifier          name;
    TypeCode            type;
    IDLType             type_def;
    ParameterMode       mode;
};

typedef sequence <ParameterDescription> ParDescriptionSeq;
typedef Identifier ContextIdentifier;
typedef sequence <ContextIdentifier> ContextIdSeq;
typedef sequence <ExceptionDef> ExceptionDefSeq;

interface OperationDef : Contained {
    readonly attribute TypeCode   result;
    attribute IDLType             result_def;
    attribute ParDescriptionSeq   params;
    attribute OperationMode       mode;
    attribute ContextIdSeq        contexts;
    attribute ExceptionDefSeq     exceptions;
};

struct OperationDescription {
    Identifier          name;
    RepositoryId        id;
    RepositoryId        defined_in;
    VersionSpec         version;
    TypeCode            result;
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        OperationMode       mode;
        ContextIdSeq        contexts;
        ParDescriptionSeq   parameters;
        ExcDescriptionSeq   exceptions;
    };

    typedef sequence <RepositoryId> RepositoryIdSeq;
    typedef sequence <OperationDescription> OpDescriptionSeq;
    typedef sequence <AttributeDescription> AttrDescriptionSeq;
    typedef sequence <ExtAttributeDescription> ExtAttrDescriptionSeq;

    interface InterfaceDef : Container, Contained, IDLType {
        // read/write interface

        attribute InterfaceDefSeq          base_interfaces;

        // read interface

        boolean is_a (
            in RepositoryId         interface_id
        );

        struct FullInterfaceDescription {
            Identifier              name;
            RepositoryId            id;
            RepositoryId            defined_in;
            VersionSpec             version;
            OpDescriptionSeq        operations;
            AttrDescriptionSeq      attributes;
            RepositoryIdSeq         base_interfaces;
            TypeCode                type;
        };

        FullInterfaceDescription describe_interface();

        // write interface
        AttributeDef create_attribute (
            in RepositoryId         id,
            in Identifier           name,
            in VersionSpec          version,
            in IDLType              type,
            in AttributeMode        mode
        );

        OperationDef create_operation (
            in RepositoryId         id,
            in Identifier           name,
            in VersionSpec          version,
            in IDLType              result,
            in OperationMode        mode,
            in ParDescriptionSeq    params,
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        in ExceptionDefSeq      exceptions,
        in ContextIdSeq         contexts
    );
};

struct InterfaceDescription {
    Identifier                  name;
    RepositoryId                id;
    RepositoryId                defined_in;
    VersionSpec                 version;
    RepositoryIdSeq             base_interfaces;
};

interface InterfaceAttrExtension {

    // read interface

    struct ExtFullInterfaceDescription {
        Identifier              name;
        RepositoryId            id;
        RepositoryId            defined_in;
        VersionSpec             version;
        OpDescriptionSeq        operations;
        ExtAttrDescriptionSeq   attributes;
        RepositoryIdSeq         base_interfaces;
        TypeCode                type;
    };

    ExtFullInterfaceDescription describe_ext_interface ();

    // write interface
    ExtAttributeDef create_ext_attribute (
        in RepositoryId         id,
        in Identifier           name,
        in VersionSpec          version,
        in IDLType              type,
        in AttributeMode        mode,
        in ExceptionDefSeq      get_exceptions,
        in ExceptionDefSeq      set_exceptions
    );
};

interface ExtInterfaceDef : InterfaceDef,
                            InterfaceAttrExtension {
};

typedef short Visibility;
const Visibility PRIVATE_MEMBER = 0;
const Visibility PUBLIC_MEMBER = 1;

struct ValueMember {
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        Identifier              name;
        RepositoryId            id;
        RepositoryId            defined_in;
        VersionSpec             version;
        TypeCode                type;
        IDLType                 type_def;
        Visibility              access;
    };

    typedef sequence <ValueMember> ValueMemberSeq;

    interface ValueMemberDef : Contained {
        readonly attribute TypeCode type;
        attribute IDLType type_def;
        attribute Visibility access;
    };

    interface ValueDef : Container, Contained, IDLType {
        // read/write interface

        attribute InterfaceDefSeq supported_interfaces;
        attribute InitializerSeq initializers;
        attribute ValueDef base_value;
        attribute ValueDefSeq abstract_base_values;
        attribute boolean is_abstract;
        attribute boolean is_custom;
        attribute boolean is_truncatable;

        // read interface
        boolean is_a(
            in RepositoryId         id
        );

        struct FullValueDescription {
            Identifier              name;
            RepositoryId            id;
            boolean                 is_abstract;
            boolean                 is_custom;
            RepositoryId            defined_in;
            VersionSpec             version;
            OpDescriptionSeq        operations;
            AttrDescriptionSeq      attributes;
            ValueMemberSeq          members;
            InitializerSeq          initializers;
            RepositoryIdSeq         supported_interfaces;
            RepositoryIdSeq         abstract_base_values;
            boolean                 is_truncatable;
            RepositoryId            base_value;
            TypeCode                type;
        };
```


© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        FullValueDescription describe_value();

        // write interface

        ValueMemberDef create_value_member(
            in RepositoryId         id,
            in Identifier           name,
            in VersionSpec          version,
            in IDLType              type,
            in Visibility           access
        );

        AttributeDef create_attribute(
            in RepositoryId         id,
            in Identifier           name,
            in VersionSpec          version,
            in IDLType              type,
            in AttributeMode        mode
        );

        OperationDef create_operation (
            in RepositoryId         id,
            in Identifier           name,
            in VersionSpec          version,
            in IDLType              result,
            in OperationMode        mode,
            in ParDescriptionSeq    params,
            in ExceptionDefSeq      exceptions,
            in ContextIdSeq         contexts
        );
    };

    struct ValueDescription {
        Identifier          name;
        RepositoryId        id;
        boolean             is_abstract;
        boolean             is_custom;
        RepositoryId        defined_in;
        VersionSpec         version;
        RepositoryIdSeq     supported_interfaces;
        RepositoryIdSeq     abstract_base_values;
        boolean             is_truncatable;
        RepositoryId        base_value;
    };

    interface ExtValueDef : ValueDef {

        // read/write interface
        attribute ExtInitializerSeq ext_initializers;

        // read interface
```


--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
        struct ExtFullValueDescription {
            Identifier              name;
            RepositoryId            id;
            boolean                 is_abstract;
            boolean                 is_custom;
            RepositoryId            defined_in;
            VersionSpec             version;
            OpDescriptionSeq        operations;
            ExtAttrDescriptionSeq   attributes;
            ValueMemberSeq          members;
            ExtInitializerSeq       initializers;
            RepositoryIdSeq         supported_interfaces;
            RepositoryIdSeq         abstract_base_values;
            boolean                 is_truncatable;
            RepositoryId            base_value;
            TypeCode                type;
        };

        ExtFullValueDescription describe_ext_value ();

        // write interface
        ExtAttributeDef create_ext_attribute (
            in RepositoryId         id,
            in Identifier           name,
            in VersionSpec          version,
            in IDLType              type,
            in AttributeMode        mode,
            in ExceptionDefSeq      get_exceptions,
            in ExceptionDefSeq      set_exceptions
        );
    };

    interface ValueBoxDef : TypedefDef {
        attribute IDLType original_type_def;
    };

    interface AbstractInterfaceDef : InterfaceDef {
    };

    interface ExtAbstractInterfaceDef :     AbstractInterfaceDef,
                                            InterfaceAttrExtension {
    };

    interface LocalInterfaceDef : InterfaceDef {
    };

    interface ExtLocalInterfaceDef :        LocalInterfaceDef,
                                            InterfaceAttrExtension {
    };

//
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
module ComponentIR {
    typeprefix ComponentIR “omg.org”;

    interface ComponentDef;
    interface HomeDef;

    interface EventDef : ExtValueDef {};

    interface Container{
        ComponentDef create_component (
            in RepositoryId     id,
            in Identifier       name,
            in VersionSpec      version,
            in ComponentDef     base_component,
            in InterfaceDefSeq supports_interfaces
        );

        HomeDef create_home (
            in RepositoryId     id,
            in Identifier       name,
            in VersionSpec      version,
            in HomeDef          base_home,
            in ComponentDef     managed_component,
            in InterfaceDefSeq supports_interfaces,
            in ValueDef         primary_key
        );

        EventDef create_event (
            in RepositoryId         id,
            in Identifier           name,
            in VersionSpec          version,
            in boolean              is_custom,
            in boolean              is_abstract,
            in ValueDef             base_value,
            in boolean              is_truncatable,
            in ValueDefSeq          abstract_base_values,
            in InterfaceDefSeq      supported_interfaces,
            in ExtInitializerSeq    initializers
        );
    };

    interface ModuleDef : CORBA::ModuleDef, Container{};

    interface Repository : CORBA::Repository, Container{};

    interface ProvidesDef : Contained {
        attribute InterfaceDef interface_type;
    };

    struct ProvidesDescription {
        Identifier      name;
        RepositoryId    id;
        RepositoryId    defined_in;
        VersionSpec     version;
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        RepositoryId    interface_type;
    };

    interface UsesDef : Contained {
        attribute InterfaceDef  interface_type;
        attribute boolean       is_multiple;
    };

    struct UsesDescription {
        Identifier      name;
        RepositoryId    id;
        RepositoryId    defined_in;
        VersionSpec     version;
        RepositoryId    interface_type;
        boolean         is_multiple;
    };

    interface EventPortDef : Contained {

        // read/write interface
        attribute EventDef event;

        // read interface
        boolean is_a (in RepositoryId event_id);
    };

    struct EventPortDescription {
        Identifier      name;
        RepositoryId    id;
        RepositoryId    defined_in;
        VersionSpec     version;
        RepositoryId    event;
    };

    interface EmitsDef : EventPortDef {};

    interface PublishesDef : EventPortDef {};

    interface ConsumesDef : EventPortDef {};

    interface ComponentDef : ExtInterfaceDef {

        // read/write interface
        attribute ComponentDef base_component;
        attribute InterfaceDefSeq supported_interfaces;

        // write interface
        ProvidesDef create_provides (
            in RepositoryId     id,
            in Identifier       name,
            in VersionSpec      version,
            in InterfaceDef     interface_type
        );

        UsesDef create_uses (
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        in RepositoryId     id,
        in Identifier       name,
        in VersionSpec      version,
        in InterfaceDef     interface_type,
        in boolean          is_multiple
    );

    EmitsDef create_emits (
        in RepositoryId     id,
        in Identifier       name,
        in VersionSpec      version,
        in EventDef         event
    );

    PublishesDef create_publishes (
        in RepositoryId     id,
        in Identifier       name,
        in VersionSpec      version,
        in EventDef         event
    );

    ConsumesDef create_consumes (
        in RepositoryId     id,
        in Identifier       name,
        in VersionSpec      version,
        in EventDef         event
    );
};

typedef sequence<ProvidesDescription>
                                    ProvidesDescriptionSeq;
typedef sequence<UsesDescription> UsesDescriptionSeq;
typedef sequence<EventPortDescription>
                                    EventPortDescriptionSeq;

struct ComponentDescription {
    Identifier                  name;
    RepositoryId                id;
    RepositoryId                defined_in;
    VersionSpec                 version;
    RepositoryId                base_component;
    RepositoryIdSeq             supported_interfaces;
    ProvidesDescriptionSeq      provided_interfaces;
    UsesDescriptionSeq          used_interfaces;
    EventPortDescriptionSeq     emits_events;
    EventPortDescriptionSeq     publishes_events;
    EventPortDescriptionSeq     consumes_events;
    ExtAttrDescriptionSeq       attributes;
    TypeCode                    type;
};

interface FactoryDef : OperationDef {};
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        interface FinderDef : OperationDef {};

        interface HomeDef : ExtInterfaceDef {

            // read/write interface
            attribute HomeDef base_home;
            attribute InterfaceDefSeq supported_interfaces;
            attribute ComponentDef managed_component;
            attribute ValueDef primary_key;

            // write interface
            FactoryDef create_factory (
                in RepositoryId         id,
                in Identifier           name,
                in VersionSpec          version,
                in ParDescriptionSeq    params,
                in ExceptionDefSeq      exceptions
            );

            FinderDef create_finder (
                in RepositoryId         id,
                in Identifier           name,
                in VersionSpec          version,
                in ParDescriptionSeq    params,
                in ExceptionDefSeq      exceptions
            );
        };

        struct HomeDescription {
            Identifier              name;
            RepositoryId            id;
            RepositoryId            defined_in;
            VersionSpec             version;
            RepositoryId            base_home;
            RepositoryId            managed_component;
            ValueDescription        primary_key;
            OpDescriptionSeq        factories;
            OpDescriptionSeq        finders;
            OpDescriptionSeq        operations;
            ExtAttrDescriptionSeq   attributes;
            TypeCode                type;
        };
    };
};
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

15 The Portable Object Adapter

This clause describes the Portable Object Adapter, or POA. It presents the design goals, a description of the abstract model of the POA and its interfaces, followed by a detailed description of the interfaces themselves.

15.1 Overview

The POA is designed to meet the following goals:

- Allow programmers to construct object implementations that are portable between different ORB products.
- Provide support for objects with persistent identities. More precisely, the POA is designed to allow programmers to build object implementations that can provide consistent service for objects whose lifetimes (from the perspective of a client holding a reference for such an object) span multiple server lifetimes.
- Provide support for transparent activation of objects.
- Allow a single servant to support multiple object identities simultaneously.
- Allow multiple distinct instances of the POA to exist in a server.
- Provide support for transient objects with minimal programming effort and overhead.
- Provide support for implicit activation of servants with POA-allocated Object Ids.
- Allow object implementations to be maximally responsible for an object's behavior. Specifically, an implementation can control an object's behavior by establishing the datum that defines an object's identity, determining the relationship between the object's identity and the object's state, managing the storage and retrieval of the object's state, providing the code that will be executed in response to requests, and determining whether or not the object exists at any point in time.
- Avoid requiring the ORB to maintain persistent state describing individual objects, their identities, where their state is stored, whether certain identity values have been previously used or not, whether an object has ceased to exist or not, and so on.
- Provide an extensible mechanism for associating policy information with objects implemented in the POA.
- Allow programmers to construct object implementations that inherit from static skeleton classes, generated by OMG IDL compilers, or a DSI implementation.

15.2 Abstract Model Description

The POA interfaces described in this clause imply a particular abstract computational model. This sub clause presents that model and defines terminology and basic concepts that will be used in subsequent sub clauses.

This sub clause provides the rationale for the POA design, describes some of its intended uses, and provides a background for understanding the interface descriptions.

15.2.1 Model Components

The model supported by the POA is a specialization of the general object model described in the OMA guide. Most of the elements of the CORBA object model are present in the model described here, but there are some new components, and some of the names of existing components are defined more precisely than they are in the CORBA object model. The abstract model supported by the POA has the following components:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- *Client*—A client is a computational context that makes requests on an object through one of its references.
- *Server*—A server is a computational context in which the implementation of an object exists. Generally, a server corresponds to a process. Note that *client* and *server* are roles that programs play with respect to a given object. A program that is a client for one object may be the server for another. The same process may be both client and server for a single object.
- *Object*—In this discussion, we use *object* to indicate a CORBA object in the abstract sense, that is, a programming entity with an identity, an interface, and an implementation. From a client's perspective, the object's identity is encapsulated in the object's reference. This specification defines the server's view of object identity, which is explicitly managed by object implementations through the POA interface.
- *Servant*—A servant is a programming language object or entity that implements requests on one or more objects. Servants generally exist within the context of a server process. Requests made on an object's references are mediated by the ORB and transformed into invocations on a particular servant. In the course of an object's lifetime it may be associated with (that is, requests on its references will be targeted at) multiple servants.
- *Object Id*—An Object Id is a value that is used by the POA and by the user-supplied implementation to identify a particular abstract CORBA object. Object Id values may be assigned and managed by the POA, or they may be assigned and managed by the implementation. Object Id values are hidden from clients, encapsulated by references. Object Ids have no standard form; they are managed by the POA as uninterpreted octet sequences.

  Note that the Object Id defined in this specification is a mechanical device used by an object implementation to correlate incoming requests with references it has previously created and exposed to clients. It does not constitute a unique logical identity for an object in any larger sense. The assignment and interpretation of Object Id values is primarily the responsibility of the application developer, although the **SYSTEM_ID** policy enables the POA to generate Object Id values for the application.
- *Object Reference*—An object reference in this model is the same as in the CORBA object model. This model implies, however, that a reference specifically encapsulates an Object Id and a POA identity.

  Note that a concrete reference in a specific ORB implementation will contain more information, such as the location of the server and POA in question. For example, it might contain the full name of the POA (the names of all POAs starting from the root and ending with the specific POA). The reference might not, in fact, actually contain the Object Id, but instead contain more compact values managed by the ORB that can be mapped to the Object Id. This is a description of the abstract information model implied by the POA. Whatever encoding is used to represent the POA name and the Object Id must not restrict the ability to use any legal character in a POA name or any legal octet in an Object Id.
- *POA*—A POA is an identifiable entity within the context of a server. Each POA provides a namespace for Object Ids and a namespace for other (nested or child) POAs. Policies associated with a POA describe characteristics of the objects implemented in that POA. Nested POAs form a hierarchical name space for objects within a server.
- *Policy*—A Policy is an object associated with a POA by an application in order to specify a characteristic shared by the objects implemented in that POA. This specification defines policies controlling the POA's threading model as well as

  a variety of other options related to the management of objects. Other specifications may define other policies that affect how an ORB processes requests on objects implemented in the POA.
- *POA Manager*—A POA manager is an object that encapsulates the processing state of one or more POAs. Using operations on a POA manager, the developer can cause requests for the associated POAs to be queued or discarded. The developer can also use the POA manager to deactivate the POAs.
- *POA Manger Factory* -- A POA Manager Factory allows explicit creation of POA managers and lookup of existing POA managers. With explicit creation, the developer can control the identity (the name) of a POA manager as well as pass configuration policies to the factory operation.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- *Servant Manager*—A servant manager is an object that the application developer can associate with a POA. The ORB will invoke operations on servant managers to activate servants on demand, and to deactivate servants. Servant managers are responsible for managing the association of an object (as characterized by its Object Id value) with a particular servant, and for determining whether an object exists or not. There are two kinds of servant managers, called **ServantActivator** and **ServantLocator**; the type used in a particular situation depends on policies in the POA.
- *Adapter Activator*—An adapter activator is an object that the application developer can associate with a POA. The ORB will invoke an operation on an adapter activator when a request is received for a child POA that does not currently exist. The adapter activator can then create the required POA on demand.

15.2.2 Model Architecture

This section describes the architecture of the abstract model implied by the POA, and the interactions between various components. The ORB is an abstraction visible to both the client and server. The POA is an object visible to the server. User-supplied implementations are registered with the POA (this statement is a simplification; more detail is provided below). Clients hold references upon which they can make requests. The ORB, POA, and implementation all cooperate to determine which servant the operation should be invoked on, and to perform the invocation.

Figure 15.1 shows the detail of the relationship between the POA and the implementation. Ultimately, a POA deals with an Object Id and an active servant. By *active servant*, we mean a programming object that exists in memory and has been presented to the POA with one or more associated object identities. There are several ways for this association to be made.

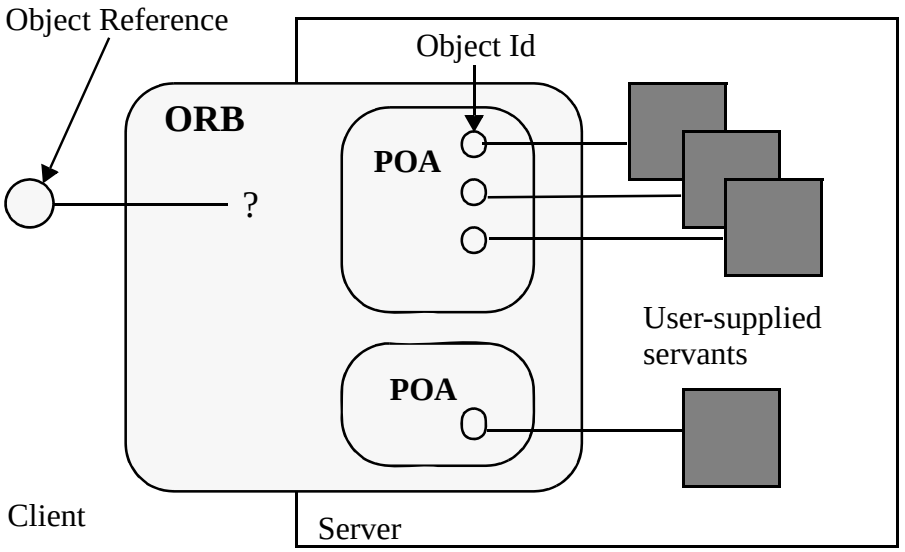


Figure 15.1 - Abstract POA Model

If the POA supports the **RETAIN** policy, it maintains a map, labeled *Active Object Map*, that associates Object Ids with active servants, each association constituting an active object. If the POA has the **USE_DEFAULT_SERVANT** policy, a default servant may be registered with the POA. Alternatively, if the POA has the **USE_SERVANT_MANAGER** policy, a user-written servant manager may be registered with the POA. If the Active Object Map is not used, or a request arrives for

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

an object not present in the Active Object Map, the POA either uses the default servant to perform the request or it invokes the servant manager to obtain a servant to perform the request. If the **RETAIN** policy is used, the servant returned by a servant manager is retained in the Active Object Map. Otherwise, the servant is used only to process the one request.

In this specification, the term *active* is applied equally to servants, Object Ids, and objects. An object is active in a POA if the POA's Active Object Map contains an entry that associates an Object Id with an existing servant. When this specification refers to *active Object Ids* and *active servants*, it means that the Object Id value or servant in question is part of an entry in the Active Object Map. An Object Id can appear in a POA's Active Object Map only once.

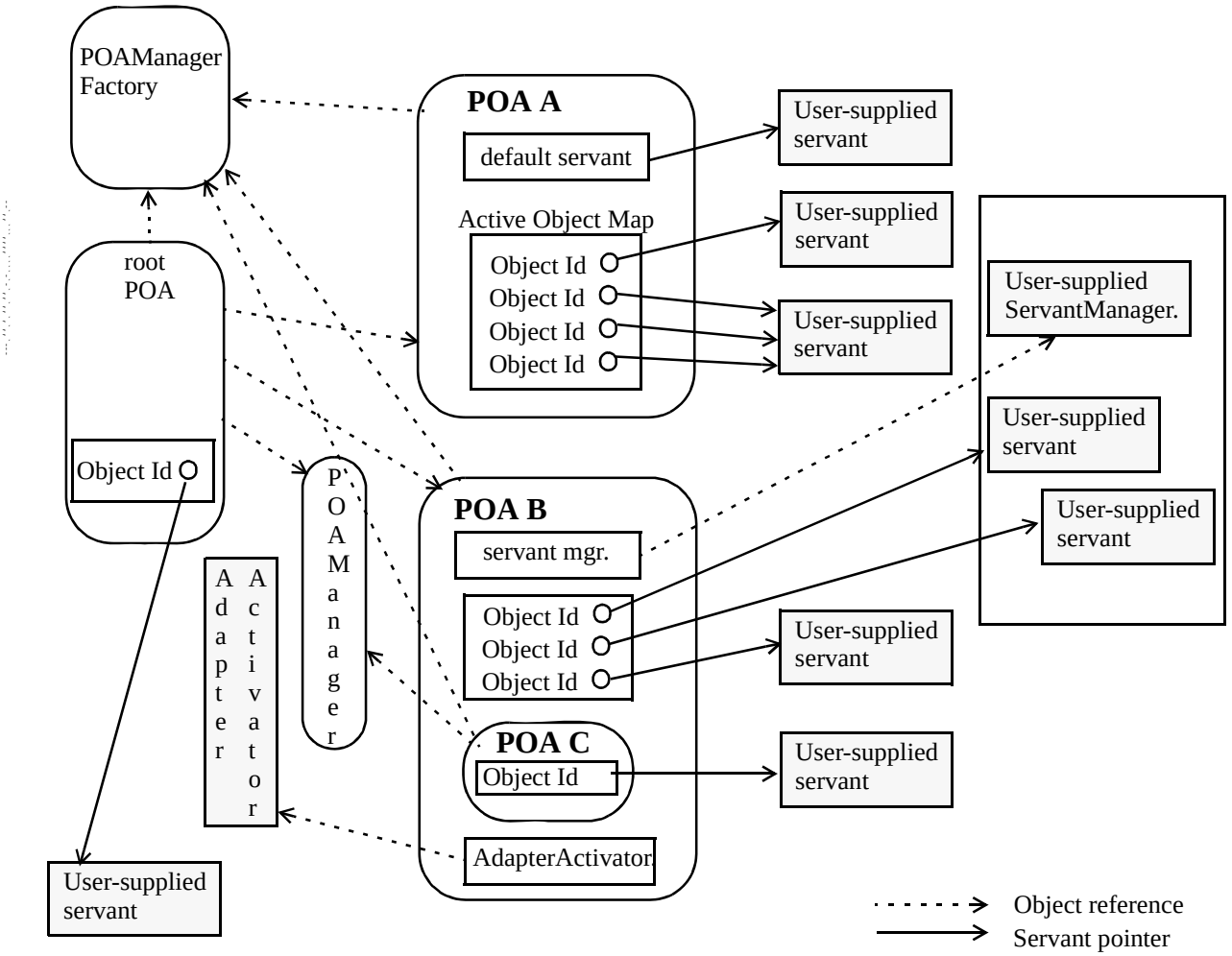


Figure 15.1 - POA Architecture

15.2.3 POA Creation

To implement an object using the POA requires that the server application obtain a POA object. A distinguished POA object, called the *root POA*, is managed by the ORB and provided to the application using the ORB initialization interface under the initial object name "RootPOA." The application developer can create objects using the root POA if those default policies are suitable. The root POA has the following policies.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- Thread Policy: **ORB_CTRL_MODEL**
- Lifespan Policy: **TRANSIENT**
- Object Id Uniqueness Policy: **UNIQUE_ID**
- Id Assignment Policy: **SYSTEM_ID**
- Servant Retention Policy: **RETAIN**
- Request Processing Policy: **USE_ACTIVE_OBJECT_MAP_ONLY**
- Implicit Activation Policy: **IMPLICIT_ACTIVATION**

The developer can also create new POAs. Creating a new POA allows the application developer to declare specific policy choices for the new POA and to provide a different adapter activator and servant manager (these are callback objects used by the POA to activate objects and nested POAs on demand). Creating new POAs also allows the application developer to partition the name space of objects, as Object Ids are interpreted relative to a POA. Finally, by creating new POAs, the developer can independently control request processing for multiple sets of objects.

A POA is created as a child of an existing POA using the **create_POA** operation on the parent POA. When a POA is created, the POA is given a name that must be unique with respect to all other POAs with the same parent.

POA objects are not persistent. No POA state can be assumed to be saved by the ORB. It is the responsibility of the server application to create and initialize the appropriate POA objects during server initialization or to set an **AdapterActivator** to create POA objects needed later.

Creating the appropriate POA objects is particularly important for persistent objects, objects whose existence can span multiple server lifetimes. To support an object reference created in a previous server process, the application must recreate the POA that created the object reference as well as all of its ancestor POAs. To ensure portability, each POA must be created with the same name as the corresponding POA in the original server process and with the same policies. (It is the user's responsibility to create the POA with these conditions.)

A portable server application can presume that there is no conflict between its POA names and the POA names chosen by other applications. It is the responsibility of the ORB implementation to provide a way to support this behavior.

Each distinct ORB created as the result of an **ORB_init** call in an application has its own separate root POA and POA namespace.

15.2.4 Reference Creation

Object references are created in servers. Once they are created, they may be exported to clients.

From this model's perspective, object references encapsulate object identity information and information required by the ORB to identify and locate the server and POA with which the object is associated (that is, in whose scope the reference was created.) References are created in the following ways:

- The server application may directly create a reference with the **create_reference** and **create_reference_with_id** operations on a POA object. These operations collect the necessary information to constitute the reference, either from information associated with the POA or as parameters to the operation. These operations only create a reference. In doing so, they bring the abstract object into existence, but do not associate it with an active servant.
- The server application may explicitly activate a servant, associating it with an object identity using the **activate_object** or **activate_object_with_id** operations. Once a servant is activated, the server application can map the servant to its corresponding reference using the **servant_to_reference** or **id_to_reference** operations.
- The server application may cause a servant to implicitly activate itself. This behavior can only occur if the POA has

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

been created with the **IMPLICIT_ACTIVATION** policy. If an attempt is made to obtain an object reference corresponding to an inactive servant, the POA may automatically assign a generated unique Object Id to the servant and activate the resulting object. The reference may be obtained by invoking **POA::servant_to_reference** with an inactive servant, or by performing an explicit or implicit type conversion from the servant to a reference type in programming language mappings that permit this conversion.

Once a reference is created in the server, it can be made available to clients in a variety of ways. It can be advertised through the OMG Naming and Trading Services. It can be converted to a string via **ORB::object_to_string** and published in some way that allows the client to discover the string and convert it to a reference using **ORB::string_to_object**. It can be returned as the result of an operation invocation.

Once a reference becomes available to a client, that reference constitutes the identity of the object from the client's perspective. As long as the client program holds and uses that reference, requests made on the reference should be sent to the "same" object.

NOTE: The meaning of object identity and "sameness" is at present the subject of debate in the OMG. This specification does not attempt to resolve that debate in any way, particularly by defining a concrete notion of identity that is exposed to clients, beyond the existing notions of identity described in the CORBA specifications and the OMA guide.

The states of servers and implementation objects are opaque to clients. This specification deals primarily with the view of the ORB from the server's perspective.

15.2.5 Object Activation States

At any point in time, a CORBA object may or may not be associated with an active servant.

If the POA has the **RETAIN** policy, the servant and its associated Object Id are entered into the Active Object Map of the appropriate POA. This type of activation can be accomplished in one of the following ways.

- The server application itself explicitly activates individual objects (via the **activate_object** or **activate_object_with_id** operations).
- The server application instructs the POA to activate objects on demand by having the POA invoke a user-supplied servant manager. The server application registers this servant manager with **set_servant_manager**.
- Under some circumstances (when the **IMPLICIT_ACTIVATION** policy is also in effect and the language binding allows such an operation), the POA may implicitly activate an object when the server application attempts to obtain a reference for a servant that is not already active (that is, not associated with an Object Id).

If the **USE_DEFAULT_SERVANT** policy is also in effect, the server application instructs the POA to activate unknown objects by having the POA invoke a single servant no matter what the Object Id is. The server application registers this servant with **set_servant**.

If the POA has the **NON_RETAIN** policy, for every request, the POA may use either a default servant or a servant manager to locate an active servant. From the POA's point of view, the servant is active only for the duration of that one request. The POA does not enter the servant-object association into the Active Object Map.

15.2.6 Request Processing

A request must be capable of conveying the Object Id of the target object as well as the identification of the POA that created the target object reference. When a client issues a request, the ORB first locates an appropriate server (perhaps starting one if needed) and then it locates the appropriate POA within that server.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

If the POA does not exist in the server process, the application has the opportunity to re-create the required POA by using an adapter activator. An adapter activator is a user-implemented object that can be associated with a POA. It is invoked by the ORB when a request is received for a non-existent child POA. The adapter activator has the opportunity to create the required POA. If it does not, the client receives the OBJECT_NOT_EXIST exception with standard minor code 2.

Once the ORB has located the appropriate POA, it delivers the request to that POA. The further processing of that request depends both upon the policies associated with that POA as well as the object's current state of activation.

If the POA has the **RETAIN** policy, the POA looks in the Active Object Map to find out if there is a servant associated with the Object Id value from the request. If such a servant exists, the POA invokes the appropriate method on the servant.

If the POA has the **NON_RETAIN** policy or has the **RETAIN** policy but didn't find a servant in the Active Object Map, the POA takes the following actions:

- If the POA has the **USE_DEFAULT_SERVANT** policy, a default servant has been associated with the POA so the POA will invoke the appropriate method on that servant. If no servant has been associated with the POA, the POA raises the OBJ_ADAPTER system exception with standard minor code 3.
- If the POA has the **USE_SERVANT_MANAGER** policy, a servant manager has been associated with the POA so the POA will invoke **incarnate** or **preinvoke** on it to find a servant that may handle the request. (The choice of method depends on the **NON_RETAIN** or **RETAIN** policy of the POA.) If no servant manager has been associated with the POA, the POA raises the OBJ_ADAPTER system exception with standard minor code 4.
- If the **USE_OBJECT_MAP_ONLY** policy is in effect, the POA raises the OBJECT_NOT_EXIST system exception with standard minor code 2.

If a servant manager is located and invoked, but the servant manager is not directly capable of incarnating the object, it (the servant manager) may deal with the circumstance in a variety of ways, all of which are the application's responsibility. Any system exception raised by the servant manager will be returned to the client in the reply. In addition to standard system exceptions, a servant manager is capable of raising a ForwardRequest exception. This exception includes an object reference. The ORB will process this exception as specified in Common Information for Servant Manager Types on page 321.

15.2.7 Implicit Activation

A POA can be created with a policy that indicates that its objects may be implicitly activated. This policy, **IMPLICIT_ACTIVATION**, also requires the **SYSTEM_ID** and **RETAIN** policies.

When a POA supports implicit activation, an inactive servant may be implicitly activated in that POA by certain operations that logically require an *Object Id* to be assigned to that servant. (**IMPLICIT_ACTIVATION** does not disallow explicit activation; instead, it enables both implicit and explicit activation.)

Implicit activation of an object involves allocating a system-generated Object Id and registering the servant with that *Object Id* in the *Active Object Map*. The interface associated with the implicitly activated object is determined from the servant (using static information from the skeleton, or, in the case of a dynamic servant, using the **_primary_interface()** operation).

The operations that support implicit activation include:

- The **POA::servant_to_reference** operation, which takes a servant parameter and returns a reference.
- The **POA::servant_to_id** operation, which takes a servant parameter and returns an Object Id.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

- Operations supported by a language mapping to obtain an object reference or an Object Id for a servant. For example, the **`_this()`** servant member function in C++ returns an object reference for the servant.
- Implicit conversions supported by a language mapping that convert a servant to an object reference or an Object Id.

The last two categories of operations are language-mapping-dependent.

If the POA has the **UNIQUE_ID** policy, then implicit activation will occur when any of these operations are performed on a servant that is not currently active (that is, it is associated with no Object Id in the POA's Active Object Map).

If the POA has the **MULTIPLE_ID** policy, the **servant_to_reference** and **servant_to_id** operations will *always* perform implicit activation, even if the servant is already associated with an Object Id. The behavior of language mapping operations in the **MULTIPLE_ID** case is specified by the language mapping. For example, in C++, the **`_this()`** servant member function will not implicitly activate a **MULTIPLE_ID** servant if the invocation of **`_this()`** is immediately within the dynamic context of a request invocation directed by the POA to that servant; instead, it returns the object reference used to issue the request.

NOTE: The exact timing of implicit activation is ORB implementation-dependent. For example, instead of activating the object immediately upon creation of a local object reference, the ORB could defer the activation until the Object Id is actually needed (for example, when the object reference is exported outside the process).

15.2.8 Multi-threading

The POA does not require the use of threads and does not specify what support is needed from a threads package. However, in order to allow the development of portable servers that utilize threads, the behavior of the POA and related interfaces when used within a multiple-thread environment must be specified.

Specifying this behavior does not require that an ORB must support being used in a threaded environment, nor does it require that an ORB must utilize threads in the processing of requests. The only requirement given here is that if an ORB does provide support for multi-threading, these are the behaviors that will be supported by that ORB. This allows a programmer to take advantage of multiple ORBs that support threads in a portable manner across those ORBs.

The POA's processing is affected by the thread-related calls available in the ORB: **work_pending**, **perform_work**, **run**, and **shutdown**.

15.2.8.1 POA Threading Models

The POA supports three models of threading when used in conjunction with multi-threaded ORB implementations; ORB controlled, single thread and main-thread behavior. The three models can be used together or independently. All can be used in environments where a single-threaded ORB is used.

The threading model associated with a POA is indicated when the POA is created by including a **ThreadPolicy** object in the policies parameter of the POA's **create_POA** operation. Once a POA is created with one model, it cannot be changed to the other. All uses of the POA within the server must conform to that threading model associated with the POA.

15.2.8.2 Using the Single Thread Model

Requests for each single-threaded POA are processed sequentially. In a multi-threaded environment, upcalls made by this POA to servants shall not be made concurrently. This provides a degree of safety for code that is multi-thread-unaware.

NOTE: In a multi-threaded environment, requests to distinct single-threaded POAs may be processed concurrently.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The POA will still allow reentrant calls from an object implementation to itself, or to another object implementation managed by the same POA.

15.2.8.3 Using the ORB Controlled Model

The ORB controlled model of threading is used in environments where the developer wants the ORB/POA to control the use of threads in the manner provided by the ORB. This model can also be used in environments that do not support threads.

In this model, the ORB is responsible for the creation, management, and destruction of threads used with one or more POAs.

15.2.8.4 Using the Main Thread Model

Requests for all main-thread POAs are processed sequentially. In a multi-threaded environment, all upcalls made by all POAs with this policy to servants are made in a manner that is safe for code that is multi-thread-unaware.

If the environment has special requirements that some code must run on a distinguished “main” thread, servant upcalls will be processed on that thread.

NOTE: Not all environments have such a special requirement. If not, while requests will be processed sequentially they might not all be processed by the same thread.

15.2.8.5 Limitations When Using Multiple Threads

There are no guarantees that the ORB and POA will do anything specific about dispatching requests across threads with a single POA. Therefore, a server programmer who wants to use one or more POAs within multiple threads must take on all of the serialization of access to objects within those threads.

There may be requests active for the same object being dispatched within multiple threads at the same time. The programmer must be aware of this possibility and code with it in mind.

15.2.9 Dynamic Skeleton Interface

The POA is designed to enable programmers to connect servants to:

- type-specific skeletons, typically generated by OMG IDL compilers, or
- dynamic skeletons.

Servants that are members of type-specific skeleton classes are referred to as type-specific servants. Servants connected to dynamic skeletons are used to implement the Dynamic Skeleton Interface (DSI) and are referred to as DSI servants.

Whether a CORBA object is being incarnated by a DSI servant or a type-specific servant is transparent to its clients. Two CORBA objects supporting the same interface may be incarnated, one by a DSI servant and the other with a type-specific servant. Furthermore, a CORBA object may be incarnated by a DSI servant only during some period of time, while the rest of the time is incarnated by a static servant.

The mapping for POA DSI servants is language-specific, with each language providing a set of interfaces to the POA. These interfaces are used only by the POA. The interfaces required are the following.

- Take a **CORBA::ServerRequest** object from the POA and perform the processing necessary to execute the request.
- Return the Interface Repository Id identifying the most-derived interface supported by the target CORBA object in a request.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The reason for the first interface is the entire reason for existence of the DSI: to be able to handle any request in the way the programmer wishes to handle it. A single DSI servant may be used to incarnate several CORBA objects, potentially supporting different interfaces.

The reason for the second interface can be understood by comparing DSI servants to type-specific servants.

A type-specific servant may incarnate several CORBA objects but all of them will support the same IDL interface as the most-derived IDL interface. In C++, for example, an IDL interface **Window** in module **GraphicalSystem** will generate a type-specific skeleton class called **Window** in namespace **POA_GraphicalSystem.** A type-specific servant that is directly derived from the **POA_GraphicalSystem::Window** skeleton class may incarnate several CORBA objects at a time, but all those CORBA objects will support the **GraphicalSystem::Window** interface as the most-derived interface.

A DSI servant may incarnate several CORBA objects, not necessarily supporting the same IDL interface as the most-derived IDL interface.

In both cases (type-specific and DSI) the POA may need to determine, at runtime, the Interface Repository Id identifying the most-derived interface supported by the target CORBA object in a request. The POA should be able to determine this by asking the servant that is going to serve the CORBA object.

In the case of type-specific servants, the POA obtains that information from the type-specific skeleton class from which the servant is directly derived. In the case of DSI servants, the POA obtains that information by using the second language-specific interface above.

15.2.10 Location Transparency

The POA supports location transparency for objects implemented using the POA. Unless explicitly stated to the contrary, all POA behavior described in this specification applies regardless of whether the client is local (same process) or remote. For example, like a request from a remote client, a request from a local client may cause object activation if the object is not active, block indefinitely if the target object's POA is in the holding state, be rejected if the target object’s POA is in the discarding or inactive states, be delivered to a thread-unaware object implementation, or be delivered to a different object if the target object's servant manager raises the ForwardRequest exception. The Object Id and POA of the target object will also be available to the server via the **Current** object, regardless of whether the client is local or remote.

NOTE: The implication of these requirements on the ORB implementation is to require the ORB to mediate all requests to POA-based objects, even if the client is co-resident in the same process. This specification is not intended to change CORBAServices specifications that allow for behaviors that are not location transparent. This specification does not prohibit (nonstandard) POA extensions to support object behavior that is not location-transparent.

15.3 Interfaces

The POA-related interfaces are defined in a module separate from the **CORBA** module, the **PortableServer** module. It consists of these interfaces:

- POA
- POAManager
- POAManagerFactory
- ServantManager
- ServantActivator
- ServantLocator

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- AdapterActivator
- ThreadPolicy
- LifespanPolicy
- IdUniquenessPolicy
- IdAssignmentPolicy
- ImplicitActivationPolicy
- ServantRetentionPolicy
- RequestProcessingPolicy
- Current

In addition, the POA defines the **Servant** native type.

All local objects specified in this clause except for **AdapterActivator**, **ServantManager**, **ServantActivator** and **ServantLocator** override the default behavior of the **Object::get_orb** operation and return the **ORB** that is associated with the root POA local object.

15.3.1 The Servant IDL Type

This specification defines a native type **PortableServer::Servant**. Values of the type **Servant** are programming-language-specific implementations of CORBA interfaces. Each language mapping must specify how **Servant** is mapped to the programming language data type that corresponds to an object implementation. The **Servant** type has the following characteristics and constraints.

- Values of type **Servant** are opaque from the perspective of CORBA application programmers. There are no operations that can be performed directly on them by user programs. They can be passed as parameters to certain POA operations. Some language mappings may allow **Servant** values to be implicitly converted to object references under appropriate conditions.
- Values of type **Servant** support a language-specific programming interface that can be used by the ORB to obtain a default POA for that servant. This interface is used only to support implicit activation. A language mapping may provide a default implementation of this interface that returns the root POA of a default ORB.
- Values of type Servant provide default implementations of the standard object reference operations **get_interface**, **is_a**, **repository_id**, and **non_existent**. These operations can be overridden by the programmer to provide additional behavior needed by the object implementation. The default implementations of **get_interface**, **repository_id,** and **is_a** operations use the most derived interface of a static servant or the most derived interface retrieved from a dynamic servant to perform the operation. The default implementation of the **non_existent** operation returns **FALSE**. These operations are invoked by the POA just like any other operation invocation, so the **PortableServer::Current** interface and any language-mapping-provided method of accessing the invocation context are available.
- Values of type **Servant** must be testable for identity.
- Values of type **Servant** have no meaning outside of the process context or address space in which they are generated.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

15.3.2 POAManager Interface

Each POA object has an associated **POAManager** object. A POA manager may be associated with one or more POA objects. A POA manager encapsulates the processing state of the POAs it is associated with. Using operations on the POA manager, an application can cause requests for those POAs to be queued or discarded, and can cause the POAs to be deactivated.

Each **POAManager** has a unique string as its identity. The scope of the **POAManager** identity is the **ORB**, so no two **POAManagers** within the same **ORB** can have the same identity (but **POAManagers** in different **ORB**s can). The **POAManager** for the Root **POA** has the name "**RootPOAManager**."

If a **POAManager** is created implicitly (as part of the creation of a new **POA**), it is assigned a unique identity by the **ORB** run time. If a **POAManager** is created explicitly (using the **POAManagerFactory**), its identity is the string passed to the factory operation. (An empty identity string is legal.) A **POAManager** is destroyed implicitly, when the last of its **POA**s is destroyed.

POAManager is a local interface.

15.3.2.1 Processing States

A POA manager has four possible processing states; *active*, *inactive*, *holding*, and *discarding*. The processing state determines the capabilities of the associated POAs and the disposition of requests received by those POAs. Figure 15.1 illustrates the processing states and the transitions between them. For simplicity of presentation, this specification sometimes describes these states as POA states, referring to the POA or POAs that have been associated with a particular POA manager. A POA manager is created in the *holding* state. The root POA is therefore initially in the *holding* state.

For simplicity in the figure and the explanation, operations that would not cause a state change are not shown. For example, if a POA is in "active" state, it does not change state due to an activate operation. Such operations complete successfully with no special notice.

The only exception is the inactive state: a **deactivate** operation invoked in the inactive state may block under certain circumstances. See deactivate on page 317 for details.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

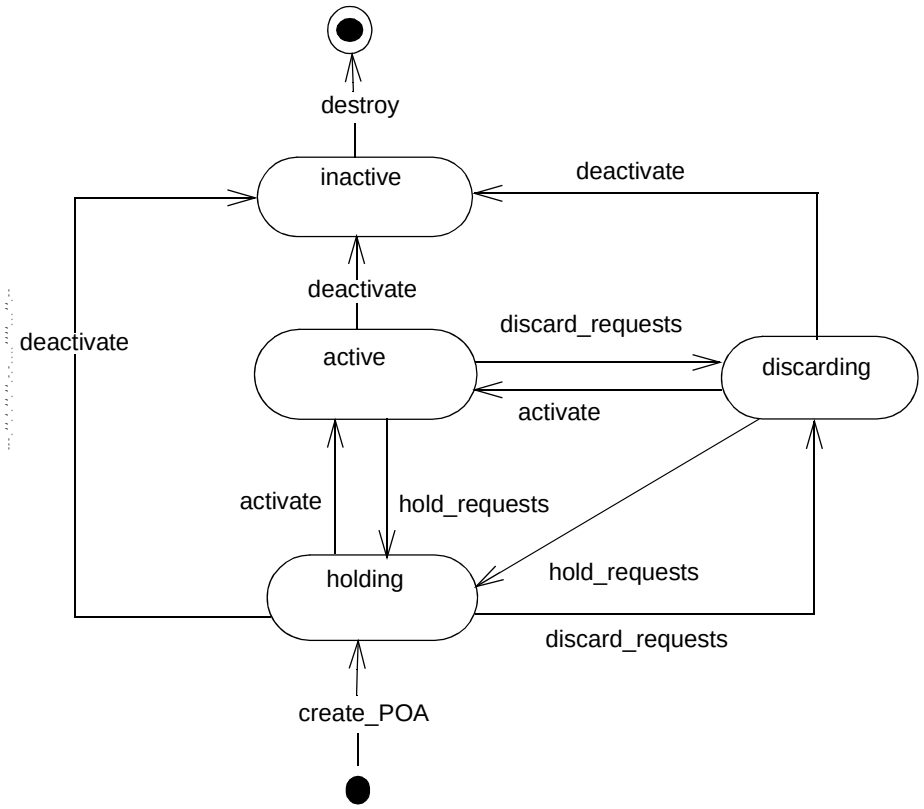


Figure 15.1 Processing States

Active State

When a POA manager is in the *active* state, the associated POAs will receive and start processing requests (assuming that appropriate thread resources are available). Note that even in the active state, a POA may need to queue requests depending upon the ORB implementation and resource limits. The number of requests that can be received and/or queued is an implementation limit. If this limit is reached, the POA should return a TRANSIENT system exception, with standard minor code 1, to indicate that the client should re-issue the request.

A user program can legally transition a POA manager from the *active* state to either the *discarding*, *holding*, or *inactive* state by calling the **discard_requests**, **hold_requests**, or **deactivate** operations, respectively. The POA enters the *active* state through the use of the **activate** operation when in the *discarding* or *holding* state.

Discarding State

When a POA manager is in the *discarding* state, the associated POAs will discard all incoming requests (whose processing has not yet begun). When a request is discarded, the TRANSIENT system exception, with standard minor code 1, must be returned to the client-side to indicate that the request should be re-issued. (Of course, an ORB may always reject a request for other reasons and raise some other system exception.)

In addition, when a POA manager is in the *discarding* state, the adapter activators registered with the associated POAs will not get called. Instead, requests that require the invocation of an adapter activator will be discarded, as described in the previous paragraph.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The primary purpose of the *discarding* state is to provide an application with flow-control capabilities when it determines that an object's implementation or POA is being flooded with requests. It is expected that the application will restore the POA manager to the *active* state after correcting the problem that caused flow-control to be needed.

A POA manager can legally transition from the *discarding* state to either the *active*, *holding*, or *inactive* state by calling the **activate**, **hold_requests**, or **deactivate** operations, respectively. The POA enters the *discarding* state through the use of the **discard_requests** operation when in the *active* or *holding* state.

Holding State

When a POA manager is in the *holding* state, the associated POAs will queue incoming requests. The number of requests that can be queued is an implementation limit. If this limit is reached, the POAs may discard requests and return the TRANSIENT system exception, with standard minor code 1, to the client to indicate that the client should reissue the request. (Of course, an ORB may always reject a request for other reasons and raise some other system exception.)

In addition, when a POA manager is in the *holding* state, the adapter activators registered with the associated POAs will not get called. Instead, requests that require the invocation of an adapter activator will be queued, as described in the previous paragraph.

A POA manager can legally transition from the *holding* state to either the *active*, *discarding*, or *inactive* state by calling the **activate**, **discard_requests**, or **deactivate** operations, respectively. The POA enters the *holding* state through the use of the **hold_requests** operation when in the *active* or *discarding* state. A POA manager is created in the holding state.

Inactive State

The *inactive* state is entered when the associated POAs are to be shut down. Unlike the *discarding* state, the *inactive* state is not a temporary state. When a POA manager is in the *inactive* state, the associated POAs will reject new requests. The rejection mechanism used is specific to the vendor. The GIOP location forwarding mechanism and CloseConnection message are examples of mechanisms that could be used to indicate the rejection. If the client is co-resident in the same process, the ORB could raise the OBJ_ADAPTER system exception, with standard minor code 1, to indicate that the object implementation is unavailable.

In addition, when a POA manager is in the *inactive* state, the adapter activators registered with the associated POAs will not get called. Instead, requests that require the invocation of an adapter activator will be rejected, as described in the previous paragraph.

The *inactive* state is entered using the **deactivate** operation. It is legal to enter the *inactive* state from either the *active*, *holding*, or *discarding* states.

If the transition into the *inactive* state is a result of calling **deactivate** with an **etherealize_objects** parameter of

- TRUE - the associated POAs will call **etherealize** for each active object associated with the POA once all currently executing requests have completed processing (if the POAs have the **RETAIN** and **USE_SERVANT_MANAGER** policies). If a servant manager has been registered for the POA, the POA will get rid of the object. If there are any queued requests that have not yet started executing, they will be treated as if they were new requests and rejected.
- FALSE - No deactivations or etherealizations will be attempted.

15.3.2.2 activate

void activate()
raises (AdapterInactive);

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This operation changes the state of the POA manager to *active*. If issued while the POA manager is in the *inactive* state, the AdapterInactive exception is raised. Entering the *active* state enables the associated POAs to process requests.

15.3.2.3 hold_requests

void hold_requests(in boolean wait_for_completion)
raises(AdapterInactive);

This operation changes the state of the POA manager to *holding*. If issued while the POA manager is in the *inactive* state, the AdapterInactive exception is raised. Entering the *holding* state causes the associated POAs to queue incoming requests. Any requests that have been queued but have not started executing will continue to be queued while in the *holding* state.

If the **wait_for_completion** parameter is **FALSE**, this operation returns immediately after changing the state. If the parameter is **TRUE** and the current thread is not in an invocation context dispatched by some POA belonging to the same ORB as this POA, this operation does not return until either there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) or the state of the POA manager is changed to a state other than *holding*. If the parameter is **TRUE** and the current thread is in an invocation context dispatched by some POA belonging to the same ORB as this POA the BAD_INV_ORDER system exception with standard minor code 3 is raised and the state is not changed.

15.3.2.4 discard_requests

void discard_requests(in boolean wait_for_completion)
raises (AdapterInactive);

This operation changes the state of the POA manager to *discarding*. If issued while the POA manager is in the *inactive* state, the AdapterInactive exception is raised. Entering the *discarding* state causes the associated POAs to discard incoming requests. In addition, any requests that have been queued but have not started executing are discarded. When a request is discarded, a TRANSIENT system exception with standard minor code 1 is returned to the client.

If the **wait_for_completion** parameter is FALSE, this operation returns immediately after changing the state. If the parameter is TRUE and the current thread is not in an invocation context dispatched by some POA belonging to the same ORB as this POA, this operation does not return until either there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) or the state of the POA manager is changed to a state other than *discarding*. If the parameter is TRUE and the current thread is in an invocation context dispatched by some POA belonging to the same ORB as this POA the BAD_INV_ORDER system exception with standard minor code 3 is raised and the state is not changed.

15.3.2.5 deactivate

void deactivate(in boolean etherealize_objects,
in boolean wait_for_completion);

This operation changes the state of the POA manager to *inactive*. This operation has no affect on the POA manager's state if it is already in the *inactive* state, but may still block if **wait_for_completion** is TRUE and another call to **deactivate** on the same POA manager is pending. Entering the inactive state causes the associated POAs to reject requests that have not begun to be executed as well as any new requests.

After changing the state, if the **etherealize_objects** parameter is

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- TRUE - the POA manager will cause all associated POAs that have the **RETAIN** and **USE_SERVANT_MANAGER** policies to perform the **etherealize** operation on the associated servant manager for all active objects.
- FALSE - the **etherealize** operation is not called. The purpose is to provide developers with a means to shut down POAs in a crisis (for example, unrecoverable error) situation.

If the **wait_for_completion** parameter is FALSE, this operation will return immediately after changing the state. If the parameter is TRUE and the current thread is not in an invocation context dispatched by some POA belonging to the same ORB as this POA, this operation does not return until there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) and, in the case of a TRUE **etherealize_objects**, all invocations of **etherealize** have completed for POAs having the **RETAIN** and **USE_SERVANT_MANAGER** policies. If the parameter is **TRUE** and the current thread is in an invocation context dispatched by some POA belonging to the same ORB as this POA the BAD_INV_ORDER system exception with standard minor code **3** is raised and the state is not changed.

If **deactivate** is called multiple times before destruction is complete (because there are active requests), the **etherealize_objects** parameter applies only to the first call of **deactivate;** subsequent calls with conflicting **etherealize_objects** settings will use the value of the **etherealize_objects** from the first call. The **wait_for_completion** parameter will be handled as defined above for each individual call (some callers may choose to block, while others may not).

15.3.2.6 get_state

```
enum State {HOLDING, ACTIVE, DISCARDING, INACTIVE};
State get_state();
```

This operation returns the state of the POA manager.

15.3.2.7 get_id

```
string get_id();
```

This operation returns the **POAManage**r's unique identity. The **id** of the **POAManager** for the Root POA is "**RootPOAManager**."

15.3.3 POAManagerFactory Interface

POAManagers can be created implicitly, by passing a nil POAManager reference to the **create_POA** operation, or can be created explicitly using a POAManagerFactory. Explicit creation of a POAManager permits application control of the POAManager's identity, whereas implicit creation results in creation of a unique identity by the ORB run time. Explicit creation of a POAManager also permits the application to assign policies to the new POAManager.

15.3.3.1 create_POAManager

```
exception ManagerAlreadyExists {};

POAManager create_POAManager(
    in string                id,
    in CORBA::PolicyList  policies
    ) raises(ManagerAlreadyExists, CORBA::PolicyError);
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This operation creates a new POAManager with the given id. If a POAManager with the given id exists already within the ORB, the operation raises **ManagerAlreadyExists**. (Note that placing a POAManager into the inactive state does not necessarily result in destruction of the POAManager because destruction of a POAManager only occurs once the last of its POAs has been destroyed. **create_POAManager** succeeds in creation of a new POAManager with the same identity as a previous POAManager only once the previous POAManager's POAs are destroyed.)

The policies parameter permits an arbitrary number of policies to be passed; these policies can be used by an ORB implementation to influence the POAManager's behavior in some way; for example, an ORB may choose to use this mechanism to pass configuration information to the factory. The policies passed to **create_POAManager** are deep-copied during creation; modification of a policy sequence after creation has therefore no effect on already existing POAManagers. If one or more of the policies are invalid, **create_POAManager** raises **CORBA::PolicyError**.

The newly created POAManager is in the Holding state.

15.3.3.2 list

typedef sequence<POAManager> POAManagerSeq;
POAManagerSeq list();

The list operation returns all POAManagers (whether created implicitly or explicitly) that currently exist within the ORB.

15.3.3.3 find

POAManager find(in string id);

The find operation return the POAManager with the specified id. If no such POAManager exists, find returns a nil reference.

15.3.4 AdapterActivator Interface

Adapter activators are associated with POAs. An adapter activator supplies a POA with the ability to create child POAs on demand, as a side-effect of receiving a request that names the child POA (or one of its children), or when **find_POA** is called with an activate parameter value of TRUE. An application server that creates all its needed POAs at the beginning of execution does not need to use or provide an adapter activator; it is necessary only for the case in which POAs need to be created during request processing.

While a request from the POA to an adapter activator is in progress, all requests to objects managed by the new POA (or any descendant POAs) will be queued. This serialization allows the adapter activator to complete any initialization of the new POA before requests are delivered to that POA.

An **AdapterActivator** object must be local to the process containing the POA objects it is registered with. **AdapterActivator** is a local interface.

15.3.4.1 unknown_adapter

boolean unknown_adapter(in POA parent, in string name);

This operation is invoked when the ORB receives a request for an object reference that identifies a target POA that does not exist. The ORB invokes this operation once for each POA that must be created in order for the target POA to exist (starting with the ancestor POA closest to the root POA). The operation is invoked on the adapter activator associated with the POA that is the parent of the POA that needs to be created. That parent POA is passed as the **parent** parameter. The name of the POA to be created (relative to the parent) is passed as the **name** parameter.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The implementation of this operation should either create the specified POA and return TRUE, or it should return FALSE. If the operation returns TRUE, the ORB will proceed with processing the request. If the operation returns FALSE, the ORB will return OBJECT_NOT_EXIST with standard minor code 2 to the client. If multiple POAs need to be created, the ORB will invoke **unknown_adapter** once for each POA that needs to be created. If the parent of a nonexistent POA does not have an associated adapter activator, the ORB will return the OBJECT_NOT_EXIST system exception with standard minor code 2.

If **unknown_adapter** raises a system exception, the ORB will report an OBJ_ADAPTER system exception with standard minor code 1.

NOTE: It is possible for another thread to create the same **POA** the **AdapterActivator** is being asked to create if **AdapterActivator**s are used in conjunction with other threads calling **create_POA** with the same **POA** name. Applications should be prepared to deal with failures from either the manual or automatic (**AdapterActivator**) **POA** creation request. There can be no guarantee of the order of such calls.

For example, if the target object reference was created by a **POA** whose full name is "A," "B," "C," "D" and only **POA**s "A" and "B" currently exist, the **unknown_adapter** operation will be invoked on the adapter activator associated with **POA** "B" passing **POA** "B" as the parent parameter and "C" as the name of the missing **POA**. Assuming that the adapter activator creates POA "C" and returns TRUE, the ORB will then invoke **unknown_adapter** on the adapter activator associated with **POA** "C," passing **POA** "C" as the parent parameter and "D" as the name.

The **unknown_adapter** operation is also invoked when **find_POA** is called on the **POA** with which the **AdapterActivator** is associated, the specified child does not exist, and the **activate_it** parameter to **find_POA** is TRUE. If **unknown_adapter** creates the specified **POA** and returns TRUE, that POA is returned from **find_POA**. If **unknown_adapter** returns FALSE then **find_POA** raises AdapterNonExistent. If **unknown_adapter** raises any system exception then **find_POA** passes through the system exception it gets back from **unknown_adapter**.

NOTE: This allows the same code, the **unknown_adapter** implementation, to be used to initialize a **POA** whether that **POA** is created explicitly by the application or as a side-effect of processing a request. Furthermore, it makes this initialization atomic with respect to delivery of requests to the **POA**.

15.3.5 ServantManager Interface

Servant managers are associated with POAs. A servant manager supplies a POA with the ability to activate objects on demand when the POA receives a request targeted at an inactive object. A servant manager is registered with a POA as a callback object, to be invoked by the POA when necessary. An application server that activates all its needed objects at the beginning of execution does not need to use a servant manager; it is used only for the case in which an object must be activated during request processing.

The **ServantManager** interface is itself empty. It is inherited by two other interfaces, **ServantActivator** and **ServantLocator**.

The two types of servant managers correspond to the POA's **RETAIN** policy (**ServantActivator**) and to the **NON_RETAIN** policy (**ServantLocator**). The meaning of the policies and the operations that are available for POAs using each policy are listed under the two types of derived interfaces.

Each servant manager type contains two operations, the first called to find and return a servant and the second to deactivate a servant. The operations differ according to the amount of information usable for their situation.

ServantManager is a local interface. A **ServantManager** object must be local to the process containing the **POA** objects it is registered with.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

15.3.5.1 Common Information for Servant Manager Types

The two types of servant managers have certain semantics that are identical.

The **incarnate** and **preinvoke** operation may raise any system exception deemed appropriate (for example, OBJECT_NOT_EXIST if the object corresponding to the Object Id value has been destroyed).

NOTE: If a user-written routine (servant manager or method code) raises the OBJECT_NOT_EXIST exception, the POA does nothing but pass on that exception. It is the user's responsibility to deactivate the object if it had been previously activated.

The **incarnate** and **preinvoke** operation may also raise a ForwardRequest exception. If this occurs, the ORB is responsible for delivering the current request and subsequent requests to the object denoted in the **forward_reference** member of the exception. The behavior of this mechanism must be the functional equivalent of the GIOP location forwarding mechanism. If the current request was delivered via an implementation of the GIOP protocol (such as IIOP), the reference in the exception should be returned to the client in a reply message with **LOCATION_FORWARD** reply status. If some other protocol or delivery mechanism was used, the ORB is responsible for providing equivalent behavior, from the perspectives of the client and the object denoted by the new reference.

If the **ForwardRequest** exception is raised anywhere else, it is passed through the ORB as a normal user exception.

If a **ServantManager** returns a null servant (or the equivalent in a language mapping) as the result of an **incarnate** or **preinvoke** operation, the **POA** returns the **OBJ_ADAPTER** system exception with standard minor code 7 as the result of the request. If the ServantManager returns the wrong type of servant, it is indeterminate when that error is detected. An **ORB** that chooses to detect the error shall raise **OBJ_ADAPTER** with standard minor code 2; an **ORB** that does not explicitly check for this error condition likely raises BAD_OPERATION with standard minor code 2 or a MARSHAL exception (with unspecified minor code) at the time of method invocation.

15.3.6 ServantActivator Interface

When the POA has the **RETAIN** policy it uses servant managers that are **ServantActivator**s. When using such servant managers, the following statements apply for a given **ObjectId** used in the **incarnate** and **etherealize** operations:

- Servants incarnated by the servant manager will be placed in the Active Object Map with objects they have activated.
- Invocations of **incarnate** on the servant manager are serialized.
- Invocations of **etherealize** on the servant manager are serialized.
- Invocations of **incarnate** and **etherealize** on the servant manager are mutually exclusive.
- Incarnations of a particular object may not overlap; that is, **incarnate** shall not be invoked with a particular **ObjectId** while, within the same POA, that **ObjectId** is in use as the **ObjectId** of an activated object or as the argument of a call to incarnate or etherealize that has not completed.

It should be noted that there may be a period of time between an object's deactivation and the etherealization (during which outstanding requests are being processed) in which arriving requests on that object should not be passed to its servant. During this period, requests targeted for such an object act as if the POA were in *holding* state until **etherealize** completes. If **etherealize** is called as a consequence of a **deactivate** call with an **etherealize_objects** parameter of TRUE, incoming requests are rejected.

It should also be noted that a similar situation occurs with **incarnate**. There may be a period of time after the POA invokes **incarnate** and before that method returns in which arriving requests bound for that object should not be passed to the servant.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A single servant manager object may be concurrently registered with multiple POAs. Invocations of **incarnate** and **etherealize** on a servant manager in the context of different POAs are not necessarily serialized or mutually exclusive. There are no assumptions made about the thread in which **etherealize** is invoked.

15.3.6.1 incarnate

```
Servant incarnate (
    in ObjectId         oid,
    in POA              adapter)
        raises (ForwardRequest);
```

This operation is invoked by the POA whenever the POA receives a request for an object that is not currently active, assuming the POA has the **USE_SERVANT_MANAGER** and **RETAIN** policies.

The **oid** parameter contains the **ObjectId** value associated with the incoming request. The **adapter** is an object reference for the POA in which the object is being activated.

The user-supplied servant manager implementation is responsible for locating or creating an appropriate servant that corresponds to the **ObjectId** value if possible. **incarnate** returns a value of type **Servant**, which is the servant that will be used to process the incoming request (and potentially subsequent requests, since the POA has the **RETAIN** policy).

The POA enters the returned **Servant** value into the Active Object Map so that subsequent requests with the same **ObjectId** value will be delivered directly to that servant without invoking the servant manager.

If the **incarnate** operation returns a servant that is already active for a different Object Id and if the POA also has the **UNIQUE_ID** policy, the **incarnate** has violated the POA policy and is considered to be in error. The POA will raise an OBJ_ADAPTER system exception for the request. In this case, etherealize is not called by the POA because the servant was never added to the Active Object Map.

NOTE: If the same servant is used in two different POAs, it is legal for the POAs to use that servant even if the POAs have different Object Id uniqueness policies. The POAs do not interact with each other in this regard.

15.3.6.2 etherealize

```
void etherealize (
    in ObjectId         oid,
    in POA              adapter,
    in Servant          serv,
    in boolean          cleanup_in_progress,
    in boolean          remaining_activations);
```

This operation is invoked whenever a servant for an object is deactivated, assuming the POA has the **USE_SERVANT_MANAGER** and **RETAIN** policies. Note that an active servant may be deactivated by the servant manager via **etherealize** even if it was not incarnated by the servant manager.

The **oid** parameter contains the Object Id value of the object being deactivated. The **adapter** parameter is an object reference for the **POA** in whose scope the object was active. The **serv** parameter contains a reference to the servant that is associated with the object being deactivated. If the servant denoted by the **serv** parameter is associated with other objects in the **POA** denoted by the **adapter** parameter (that is, in the **POA**'s Active Object Map) at the time that **etherealize** is called, the **remaining_activations** parameter has the value **TRUE**. Otherwise, it has the value **FALSE**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

If the **cleanup_in_progress** parameter is **TRUE**, the reason for the **etherealize** operation is that either the **deactivate** or **destroy** operation was called with an **etherealize_objects** parameter of **TRUE**. If the parameter is **FALSE**, the **etherealize** operation is called for other reasons.

Deactivation occurs in the following circumstances:

- When an object is deactivated explicitly by an invocation of **POA::deactivate_object**.
- When the ORB or POA determines internally that an object must be deactivated. For example, an ORB implementation may provide policies that allow objects to be deactivated after some period of quiescence, or when the number of active objects reaches some limit.
- If **POAManager::deactivate** is invoked on a POA manager associated with a POA that has currently active objects.

Destroying a servant that is in the Active Object Map or is otherwise known to the POA can lead to undefined results.

In a multi-threaded environment, the **POA** makes certain guarantees that allow servant managers to safely destroy servants. Specifically, the servant's entry in the Active Object Map corresponding to the target object is removed before **etherealize** is called. Because calls to **incarnate** and **etherealize** are serialized, this prevents new requests for the target object from being invoked on the servant during etherealization. After removing the entry from the Active Object Map, if the **POA** determines before invoking **etherealize** that other requests for the same target object are already in progress on the servant, it delays the call to **etherealize** until all active methods for the target object have completed. Therefore, when **etherealize** is called, the servant manager can safely destroy the servant if it wants to, unless the **remaining_activations** argument is **TRUE**.

If the **etherealize** operation returns a system exception, the **POA** ignores the exception.

15.3.7 ServantLocator Interface

When the **POA** has the **NON_RETAIN** policy it uses servant managers that are **ServantLocators**. Because the **POA** knows that the servant returned by this servant manager will be used only for a single request, it can supply extra information to the servant manager's operations and the servant manager's pair of operations may be able to cooperate to do something different than a **ServantActivator**.

ServantLocator is a local interface. A **ServantLocator** object must be local to the process containing the **POA** objects it is registered with.

When the **POA** uses the **ServantLocator** interface, immediately after performing the operation invocation on the servant returned by **preinvoke**, the **POA** will invoke **postinvoke** on the servant manager, passing the **ObjectId** value and the **Servant** value as parameters (among others). The next request with this **ObjectId** value will then cause **preinvoke** to be invoked again. This feature may be used to force every request for objects associated with a **POA** to be mediated by the servant manager.

When using such a **ServantLocator**, the following statements apply for a given **ObjectId** used in the **preinvoke** and **postinvoke** operations:

- The servant returned by **preinvoke** is used only to process the single request that caused **preinvoke** to be invoked.
- No servant incarnated by the servant manager will be placed in the Active Object Map.
- When the invocation of the request on the servant is complete, **postinvoke** will be invoked for the object.
- No serialization of invocations of **preinvoke** or **postinvoke** may be assumed; there may be multiple concurrent invocations of **preinvoke** for the same **ObjectId**. (However, if the **SINGLE_THREAD_MODEL** policy is being used, that policy will serialize these calls.)

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- The same thread will be used to **preinvoke** the object, process the request, and **postinvoke** the object.
- If **preinvoke** raises an exception, **postinvoke** is not called. Otherwise the **preinvoke** and **postinvoke** operations are always called in pairs in response to any ORB activity. In particular, for a response to a **GIOP Locate** message a **GIOP**-conforming ORB may (or may not) call preinvoke to determine whether the object could be served at this location. If the ORB makes such a call, whatever the result, the ORB does not invoke a method, but does call postinvoke before responding to the **Locate** message.

NOTE: The **ServantActivator** interface does not behave similarly with respect to a **GIOP Locate** message since the etherealize operation is not associated with request processing.

15.3.7.1 preinvoke

```
Servant preinvoke(
    in ObjectId             oid,
    in POA                  adapter,
    in CORBA::Identifier    operation,
    out Cookie              the_cookie)
    raises (ForwardRequest
);
```

This operation is invoked by the POA whenever the POA receives a request for an object that is not currently active, assuming the POA has the **USE_SERVANT_MANAGER** and **NON_RETAIN** policies.

The **oid** parameter contains the **ObjectId** value associated with the incoming request. The **adapter** is an object reference for the POA in which the object is being activated.

The user-supplied servant manager implementation is responsible for locating or creating an appropriate servant that corresponds to the **ObjectId** value if possible. **preinvoke** returns a value of type **Servant**, which is the servant that will be used to process the incoming request.

The **Cookie** is a type opaque to the **POA** that can be set by the servant manager for use later by **postinvoke**. The operation is the name of the operation that will be called by the **POA** when the servant is returned.

15.3.7.2 postinvoke

```
void postinvoke(
    in ObjectId             oid,
    in POA                  adapter,
    in CORBA::Identifier    operation,
    in Cookie               the_cookie,
    in Servant              the_servant
);
```

This operation is invoked whenever a servant completes a request, assuming the POA has the **USE_SERVANT_MANAGER** and **NON_RETAIN** policies.

The **postinvoke** operation is considered to be part of a request on an object.That is, the request is not complete until postinvoke finishes. If the method finishes normally but postinvoke raises a system exception, the method's normal return is overridden; the request completes with the exception.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The **oid** parameter contains the Object Id value of the object on which the request was made. The **adapter** parameter is an object reference for the POA in whose scope the object was active. The **the_servant** parameter contains a reference to the servant that is associated with the object.

The **Cookie** is a type opaque to the **POA**; it contains any value that was set by the **preinvoke** operation. The operation is the name of the operation that was called by the **POA** for the request.

Destroying a servant that is known to the **POA** can lead to undefined results.

15.3.7.3 ServantLocator and Location Determination

Under certain circumstances, an ORB may need to determine the actual location of an object's implementation. For objects that are managed by a POA that is configured with a **ServantLocator**, it may invoke **preinvoke** and **postinvoke** or it may determine the object's location by some other means. If it invokes **preinvoke** and **postinvoke** under these circumstances it shall use the argument "**_locate**."

15.3.8 POA Policy Objects

Interfaces derived from **CORBA::Policy** are used with the **POA::create_POA** operation to specify policies that apply to a POA. Policy objects are created using factory operations on any pre-existing POA, such as the root POA, or by a call to **ORB::create_policy**. Policy objects are specified when a POA is created. Policies may not be changed on an existing POA. Policies are not inherited from the parent POA. All **Policy** interfaces defined in this sub clause are local interfaces.

The POA shall preserve Policies whose types have been registered via **PortableInterceptor::ORBInitInfo::register_policy_factory**, even if the POA itself does not know about those policies.

15.3.8.1 Thread Policy

Objects with the **ThreadPolicy** interface are obtained using the **POA::create_thread_policy** operation and passed to the **POA::create_POA** operation to specify the threading model used with the created POA. The value attribute of **ThreadPolicy** contains the value supplied to the **POA::create_thread_policy** operation from which it was obtained. The following values can be supplied.

- **ORB_CTRL_MODEL** - The ORB is responsible for assigning requests for an ORB- controlled POA to threads. In a multi-threaded environment, concurrent requests may be delivered using multiple threads.
- **SINGLE_THREAD_MODEL** - Requests for a single-threaded POA are processed sequentially. In a multi-threaded environment, all upcalls made by this POA to implementation code (servants and servant managers) are made in a manner that is safe for code that is multi-thread-unaware. The POA will still allow reentrant calls from an object implementation to itself, or to another object implementation managed by the same POA.
- **MAIN_THREAD_MODEL** - Requests for all main-thread POAs are processed sequentially. In a multi-threaded environment, all upcalls made by all POAs with this policy to servants are made in a manner that is safe for code that is multi-thread-unaware. If the environment has special requirements that some code must run on a distinguished "main" thread, servant upcalls will be processed on that thread.

If no **ThreadPolicy** object is passed to **create_POA**, the thread policy defaults to **ORB_CTRL_MODEL**.

NOTE: In some environments, calling multi-thread-unaware code safely (that is, using the **MAIN_THREAD_MODEL**) may mean that the POA will use only the main thread, in which case the application programmer is responsible to ensure that the main thread is given to the ORB, using **ORB::perform_work** or **ORB::run**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

POAs using the **SINGLE_THREAD_MODEL** may need to cooperate to ensure that calls are safe even when implementation code (such as a servant manager) is shared by multiple single-threaded POAs.

These models presume that the ORB and the application are using compatible threading primitives in a multi-threaded environment.

15.3.8.2 Lifespan Policy

Objects with the **LifespanPolicy** interface are obtained using the **POA::create_lifespan_policy** operation and passed to the **POA::create_POA** operation to specify the lifespan of the objects implemented in the created POA. The following values can be supplied.

- **TRANSIENT** - The objects implemented in the **POA** cannot outlive the **POA** instance in which they are first created. Once the POA's **POAManager** enters the deactivated state, any requests received by this **POA** will cause the **POA** to raise an **OBJECT_NOT_EXIST** system exception with standard minor code 4.
- **PERSISTENT** - The objects implemented in the **POA** can outlive the process in which they are first created.
  - Persistent objects have a **POA** associated with them (the **POA** that created them). When the ORB receives a request on a persistent object, it first searches for the matching **POA**, based on the names of the **POA** and all of its ancestors.
  - Administrative action beyond the scope of this specification may be necessary to inform the ORB's location service of the creation and eventual termination of existence of this **POA**, and optionally to arrange for on-demand activation of a process implementing this **POA**.
  - **POA** names must be unique within their enclosing scope (the parent **POA**). A portable program can assume that **POA** names used in other processes will not conflict with its own **POA** names. A conforming CORBA implementation will provide a method for ensuring this property.

If no **LifespanPolicy** object is passed to **create_POA**, the lifespan policy defaults to **TRANSIENT**.

15.3.8.3 Object Id Uniqueness Policy

Objects with the **IdUniquenessPolicy** interface are obtained using the **POA::create_id_uniqueness_policy** operation and passed to the **POA::create_POA** operation to specify whether the servants activated in the created **POA** must have unique object identities. The following values can be supplied.

- **UNIQUE_ID** - Servants activated with that POA support exactly one Object Id.
- **MULTIPLE_ID** - a servant activated with that POA may support one or more Object Ids.

If no **IdUniquenessPolicy** is specified at POA creation, the default is **UNIQUE_ID**.

NOTE: Use of **UNIQUE_ID** policy is meaningless in conjunction with **NON_RETAIN** policy. A conforming application should not use this policy combination. A conforming orb may, but need not, report an error during **create_POA** if this combination is used. If an orb permits this combination of policies to be used, the resulting **POA** shall not treat the use of the same servant for concurrent requests on different object ids as an error.

15.3.8.4 Id Assignment Policy

Objects with the **IdAssignmentPolicy** interface are obtained using the **POA::create_id_assignment_policy** operation and passed to the **POA::create_POA** operation to specify whether Object Ids in the created **POA** are generated by the application or by the ORB. The following values can be supplied.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- **USER_ID** - Objects created with that **POA** are assigned Object Ids only by the application.
- **SYSTEM_ID** - Objects created with that **POA** are assigned Object Ids only by the **POA**. If the **POA** also has the **PERSISTENT** policy, assigned Object Ids must be unique across all instantiations of the same POA.

If no **IdAssignmentPolicy** is specified at POA creation, the default is **SYSTEM_ID**.

15.3.8.5 Servant Retention Policy

Objects with the **ServantRetentionPolicy** interface are obtained using the **POA::create_servant_retention_policy** operation and passed to the **POA::create_POA** operation to specify whether the created **POA** retains active servants in an Active Object Map. The following values can be supplied.

- **RETAIN -** The POA will retain active servants in its Active Object Map.
- **NON_RETAIN -** Servants are not retained by the POA.

If no **ServantRetentionPolicy** is specified at POA creation, the default is **RETAIN**.

NOTE: The **NON_RETAIN** policy requires either the **USE_DEFAULT_SERVANT** or **USE_SERVANT_MANAGER** policies.

15.3.8.6 Request Processing Policy

Objects with the **RequestProcessingPolicy** interface are obtained using the **POA::create_request_processing_policy** operation and passed to the **POA::create_POA** operation to specify how requests are processed by the created **POA**. The following values can be supplied.

- **USE_ACTIVE_OBJECT_MAP_ONLY** - If the Object Id is not found in the Active Object Map, an OBJECT_NOT_EXIST system exception with standard minor code 2 is returned to the client. The **RETAIN** policy is also required.
- **USE_DEFAULT_SERVANT** - If the Object Id is not found in the Active Object Map or the **NON_RETAIN** policy is present, and a default servant has been registered with the **POA** using the **set_servant** operation, the request is dispatched to the default servant. If no default servant has been registered, an OBJ_ADAPTER system exception with standard minor code 3 is returned to the client. The **MULTIPLE_ID** policy is also required.
- **USE_SERVANT_MANAGER** - If the Object Id is not found in the Active Object Map or the **NON_RETAIN** policy is present, and a servant manager has been registered with the **POA** using the **set_servant_manager** operation, the servant manager is given the opportunity to locate a servant or raise an exception. If no servant manager has been registered, an OBJ_ADAPTER system exception with standard minor code 4 is returned to the client.

If no **RequestProcessingPolicy** is specified at **POA** creation, the default is **USE_ACTIVE_OBJECT_MAP_ONLY**.

By means of combining the **USE_ACTIVE_OBJECT_MAP_ONLY** / **USE_DEFAULT_SERVANT** / **USE_SERVANT_MANAGER** policies and the **RETAIN** / **NON_RETAIN** policies, the programmer is able to define a rich number of possible behaviors.

RETAIN and USE_ACTIVE_OBJECT_MAP_ONLY

This combination represents the situation where the **POA** does no automatic object activation (that is, the **POA** searches only the Active Object Map).

RETAIN and USE_SERVANT_MANAGER

This combination represents a very common situation, where there is an Active Object Map and a **ServantManager**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Because **RETAIN** is in effect, the application can call **activate_object** or **activate_object_with_id** to establish known servants in the Active Object Map for use in later requests.

If the **POA** doesn't find a servant in the Active Object Map for a given object, it tries to determine the servant by means of invoking incarnate in the **ServantManager** (specifically a **ServantActivator**) registered with the POA. If no **ServantManager** is available, the **POA** raises the **OBJ_ADAPTER** system exception with standard minor code 4.

RETAIN and USE_DEFAULT_SERVANT

This combination represents the situation where there is a default servant defined for all requests involving unknown objects.

Because **RETAIN** is in effect, the application can call **activate_object** or **activate_object_with_id** to establish known servants in the Active Object Map for use in later requests.

The **POA** first tries to find a servant in the Active Object Map for a given object. If it does not find such a servant, it uses the default servant. If no default servant is available, the **POA** raises the OBJ_ADAPTER system exception with standard minor code 3.

NON-RETAIN and USE_SERVANT_MANAGER

This combination represents the situation where one servant is used per method call.

The **POA** doesn't try to find a servant in the Active Object Map because the **ActiveObjectMap** does not exist. In every request, it will call preinvoke on the **ServantManager** (specifically a **ServantLocator**) registered with the **POA**. If no **ServantManager** is available, the **POA** will raise the OBJ_ADAPTER system exception.

NON-RETAIN and USE_DEFAULT_SERVANT

This combination represents the situation where there is one single servant defined for all CORBA objects.

The **POA** does not try to find a servant in the Active Object Map because the **ActiveObjectMap** doesn't exist. In every request, the **POA** will invoke the appropriate operation on the default servant registered with the **POA**. If no default servant is available, the **POA** will raise the OBJ_ADAPTER system exception.

15.3.8.7 Implicit Activation Policy

Objects with the **ImplicitActivationPolicy** interface are obtained using the **POA::create_implicit_activation_policy** operation and passed to the **POA::create_POA** operation to specify whether implicit activation of servants is supported in the created POA. The following values can be supplied.

- **IMPLICIT_ACTIVATION** - the POA will support implicit activation of servants. **IMPLICIT_ACTIVATION** also requires the **SYSTEM_ID** and **RETAIN** policies.
- **NO_IMPLICIT_ACTIVATION** - the POA will not support implicit activation of servants.

If no **ImplicitActivationPolicy** is specified at POA creation, the default is **NO_IMPLICIT_ACTIVATION**.

15.3.9 POA Interface

A POA object manages the implementation of a collection of objects. The POA supports a name space for the objects, which are identified by Object Ids.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A POA also provides a name space for POAs. A POA is created as a child of an existing POA, which forms a hierarchy starting with the root POA.

The **POA** interface is a local interface.

15.3.9.1 create_POA

```
POA create_POA(
    in string                adapter_name,
    in POAManager            a_POAManager,
    in CORBA::PolicyList     policies)
    raises (AdapterAlreadyExists, InvalidPolicy
);
```

This operation creates a new POA as a child of the target POA. The specified name identifies the new POA with respect to other POAs with the same parent POA. If the target POA already has a child POA with the specified name, the AdapterAlreadyExists exception is raised.

If the **a_POAManager** parameter is null, a new **POAManager** object is created and associated with the new POA. Otherwise, the specified **POAManager** object is associated with the new POA. The **POAManager** object can be obtained using the attribute name **the_POAManager**.

The specified policy objects are associated with the POA and used to control its behavior. The policy objects are effectively copied before this operation returns, so the application is free to destroy them while the POA is in use. Policies are *not* inherited from the parent POA.

The POA shall preserve Policies whose types have been registered via **PortableInterceptor::ORBInitInfo::register_policy_factory**, even if the POA itself does not know about those policies.

If any of the policy objects specified are not valid for the ORB implementation, if conflicting policy objects are specified, or if any of the specified policy objects require prior administrative action that has not been performed, an InvalidPolicy exception is raised containing the index in the policies parameter value of the first offending policy object.

NOTE: Creating a POA using a POA manager that is in the active state can lead to race conditions if the POA supports preexisting objects, because the new POA may receive a request before its adapter activator, servant manager, or default servant have been initialized. These problems do not occur if the POA is created by an adapter activator registered with a parent of the new POA, because requests are queued until the adapter activator returns. To avoid these problems when a POA must be explicitly initialized, the application can initialize the POA by invoking **find_POA** with a TRUE activate parameter.

15.3.9.2 find_POA

```
POA find_POA(
    in string        adapter_name,
    in boolean       activate_it)
    raises (AdapterNonExistent
);
```

If the target **POA** is the parent of a child **POA** with the specified name (relative to the target **POA**), that child **POA** is returned. If a child **POA** with the specified name does not exist and the value of the **activate_it** parameter is **TRUE**, the target **POA**'s **AdapterActivator**, if one exists, is invoked, and, if it successfully activates the child **POA**, that child **POA** is returned. Otherwise, the AdapterNonExistent exception is raised.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

If **find_POA** receives a system exception in response to a call to **unknown_adapter** on a **POA**, then **find_POA** passes through the system exception it received from **unknown_adapter**.

15.3.9.3 destroy

```
void destroy(
    in boolean      etherealize_objects,
    in boolean      wait_for_completion
);
```

This operation destroys the **POA** and all descendant **POAs**. All descendant **POA**s are destroyed (recursively) before the destruction of the containing **POA**. The **POA** so destroyed (that is, the **POA** with its name) may be re-created later in the same process. (This differs from the **POAManager::deactivate** operation that does not allow a re-creation of its associated **POA** in the same process. After a deactivate, re-creation is allowed only if the **POA** is later destroyed.)

When **destroy** is called the **POA** behaves as follows:

- The **POA** assumes the *discarding* state except when its **POAManager** is in the *inactive* state in which case the **POA** assumes the *inactive* state. Any further changes to the **POAManager**'s state do not affect this **POA**.
- The **POA** disables the **create_POA** operation. Subsequent calls to **create_POA** will result in a BAD_INV_ORDER system exception with standard minor code 17.
- The **POA** calls destroy on all of its immediate descendants.
- After all descendant **POA**s have been destroyed and their servants etherealized, the **POA** continues to process requests until there are no requests executing in the **POA**. At this point, apparent destruction of the **POA** has occurred.
- After destruction has become apparent, the **POA** may be re-created via either an AdapterActivator or a call to **create_POA**.
- If the **etherealize_objects** parameter is TRUE, the **POA** has the **RETAIN** policy, and a servant manager is registered with the **POA**, the **etherealize** operation on the servant manager is called for each *active* object in the *Active Object Map*. The apparent destruction of the **POA** occurs before any calls to **etherealize** are made. Thus, for example, an **etherealize** method that attempts to invoke operations on the **POA** receives the OBJECT_NOT_EXIST exception.
- If the **POA** has an **AdapterActivator** installed, any requests that would have caused **unknown_adapter** to be called cause a TRANSIENT exception with standard minor code 4 to be raised instead.

The **wait_for_completion** parameter is handled as follows:

- If **wait_for_completion** is **TRUE** and the current thread is not in an invocation context dispatched from some **POA** belonging to the same ORB as this **POA**, the destroy operation returns only after all active requests have completed and all invocations of **etherealize** have completed.
- If **wait_for_completion** is **TRUE** and the current thread is in an invocation context dispatched from some **POA** belonging to the same ORB as this **POA**, the BAD_INV_ORDER system exception with standard minor code 3 is raised and **POA** destruction does not occur.
- If **wait_for_completion** is **FALSE**, the **destroy** operation destroys the POA and its children but waits neither for active requests to complete nor for etherealization to occur. If **destroy** is called multiple times before destruction is complete (because there are active requests), the **etherealize_objects** parameter applies only to the first call of **destroy**. Subsequent calls with conflicting **etherealize_objects** settings use the value of **etherealize_objects** from the first call. The **wait_for_completion** parameter is handled as defined above for each individual call (some callers may choose to block, while others may not).

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

15.3.9.4 Policy Creation Operations

```
ThreadPolicy create_thread_policy(
        in ThreadPolicyValue value);
LifespanPolicy create_lifespan_policy(
        in LifespanPolicyValue value);
IdUniquenessPolicy create_id_uniqueness_policy(
        in IdUniquenessPolicyValue value);
IdAssignmentPolicy create_id_assignment_policy(
        in IdAssignmentPolicyValue value);
ImplicitActivationPolicy create_implicit_activation_policy(
        in ImplicitActivationPolicyValue value);
ServantRetentionPolicy create_servant_retention_policy(
        in ServantRetentionPolicyValue value);
RequestProcessingPolicy create_request_processing_policy(
        in RequestProcessingPolicyValue value);
```

These operations each return a reference to a policy object with the specified value. The application is responsible for calling the inherited **destroy** operation on the returned reference when it is no longer needed.

15.3.9.5 the_name

```
readonly attribute string the_name;
```

This attribute identifies the POA relative to its parent. This name is assigned when the POA is created. The name of the root POA is system-dependent and should not be relied upon by the application. In order to work properly with Portable Interceptors (see Adapter Names on page 392) the name of the root POA must be the sequence containing only the string "RootPOA."

15.3.9.6 the_parent

```
readonly attribute POA the_parent;
```

This attribute identifies the parent of the POA. The parent of the root POA is null.

15.3.9.7 the_children

```
readonly attribute POAList the_children;
```

This attribute identifies the current set of all child POAs of the POA. The set of child POAs includes only the POA's immediate children, and not their descendants.

15.3.9.8 the_POAManager

```
readonly attribute POAManager the_POAManager;
```

This attribute identifies the POA manager associated with the POA.

15.3.9.9 the_activator

```
attribute AdapterActivator the_activator;
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This attribute identifies the adapter activator associated with the POA. A newly created POA has no adapter activator (the attribute is null). It is system-dependent whether the root POA initially has an adapter activator; the application is free to assign its own adapter activator to the root POA.

15.3.9.10 the_POAManagerFactory

```
readonly attribute POAManagerFactory the_POAManagerFactory;
```

This attribute returns the **POAManagerFactory** that created the **POA**.

15.3.9.11 get_servant_manager

```
ServantManager get_servant_manager()
    raises(WrongPolicy);
```

This operation requires the **USE_SERVANT_MANAGER** policy; if not present, the WrongPolicy exception is raised.

This operation returns the servant manager associated with the POA. If no servant manager has been associated with the POA, it returns a null reference.

15.3.9.12 set_servant_manager

```
void set_servant_manager(
    in ServantManager imgr
) raises(WrongPolicy);
```

This operation requires the **USE_SERVANT_MANAGER** policy; if not present, the WrongPolicy exception is raised.

If the **ServantRetentionPolicy** of the **POA** is **RETAIN**, then the **ServantManager** argument (**imgr**) shall support the **ServantActivator** interface (e.g., in C++ `imgr` is narrowable to **ServantActivator**). If the **ServantRetentionPolicy** of the **POA** is **NON_RETAIN**, then the **ServantManager** argument shall support the **ServantLocator** interface. If the argument is **nil**, or does not support the required interface, then the OBJ_ADAPTER system exception with standard minor code 4 is raised.

This operation sets the default servant manager associated with the POA. This operation may only be invoked once after a POA has been created. Attempting to set the servant manager after one has already been set will result in the BAD_INV_ORDER system exception with standard minor code 6 being raised.

15.3.9.13 get_servant

```
Servant get_servant()
    raises(NoServant, WrongPolicy);
```

This operation requires the **USE_DEFAULT_SERVANT** policy; if not present, the WrongPolicy exception is raised.

This operation returns the default servant associated with the POA. If no servant has been associated with the POA, the NoServant exception is raised.

15.3.9.14 set_servant

```
void set_servant(
    in Servant p_servan
) raises(WrongPolicy);
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This operation requires the **USE_DEFAULT_SERVANT** policy; if not present, the WrongPolicy exception is raised.

This operation registers the specified servant with the POA as the default servant. This servant will be used for all requests for which no servant is found in the Active Object Map.

15.3.9.15 activate_object

```
ObjectId activate_object(
    in Servant p_servant
) raises (ServantAlreadyActive, WrongPolicy);
```

This operation requires the **SYSTEM_ID** and **RETAIN** policy; if not present, the WrongPolicy exception is raised.

If the POA has the **UNIQUE_ID** policy and the specified servant is already in the Active Object Map, the ServantAlreadyActive exception is raised. Otherwise, the **activate_object** operation generates an Object Id and enters the Object Id and the specified servant in the Active Object Map. The Object Id is returned.

15.3.9.16 activate_object_with_id

```
void activate_object_with_id(
    in          ObjectId oid,
    in Servant  p_servant
) raises (ObjectAlreadyActive, ServantAlreadyActive, WrongPolicy);
```

This operation requires the **RETAIN** policy; if not present, the WrongPolicy exception is raised.

If the CORBA object denoted by the Object Id value is already active in this POA (there is a servant bound to it in the Active Object Map), the ObjectAlreadyActive exception is raised. If the POA has the **UNIQUE_ID** policy and the servant is already in the Active Object Map, the ServantAlreadyActive exception is raised. Otherwise, the **activate_object_with_id** operation enters an association between the specified Object Id and the specified servant in the Active Object Map.

If the **POA** has the **SYSTEM_ID** policy and it detects that the Object Id value was not generated by the system or for this **POA**, the **activate_object_with_id** operation may raise the BAD_PARAM system exception. An ORB is not required to detect all such invalid Object Id values, but a portable application must not invoke **activate_object_with_id** on a **POA** that has the **SYSTEM_ID** policy with an Object Id value that was not previously generated by the system for that **POA**, or, if the **POA** also has the **PERSISTENT** policy, for a previous instantiation of the same **POA**. A **POA** is not required to raise the BAD_PARAM exception in this case because, in the general case, accurate rejection of an invalid Object Id requires unbounded persistent memory of all previously generated Id values.

15.3.9.17 deactivate_object

```
void deactivate_object(
    in ObjectId oid
) raises (ObjectNotActive, WrongPolicy);
```

This operation requires the **RETAIN** policy; if not present, the WrongPolicy exception is raised.

This operation causes the **ObjectId** specified in the **oid** parameter to be deactivated. An **ObjectId** that has been deactivated continues to process requests until there are no active requests for that **ObjectId**. Active requests are those requests that arrived before **deactivate_object** was called. A deactivated **ObjectId** is removed from the Active Object Map when all requests executing for that **ObjectId** have completed. If a servant manager is associated with the **POA**, **ServantActivator::etherealize** is invoked with the **oid** and the associated servant after the **ObjectId** has been removed

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

from the Active Object Map. Reactivation for the **ObjectId** blocks until etherealization (if necessary) is complete. This includes implicit activation (as described in etherealize) and explicit activation via **POA::activate_object_with_id**. Once an **ObjectId** has been removed from the Active Object Map and etherealized (if necessary) it may then be reactivated through the usual mechanisms. The operation does not wait for requests or etherealization to complete and always returns immediately after deactivating the **ObjectId**.

If the servant associated with the **oid** is serving multiple Object Ids, **ServantActivator::etherealize** may be invoked multiple times with the same servant when the other objects are deactivated. It is the responsibility of the object implementation to refrain from destroying the servant while it is active with any Id.

15.3.9.18 create_reference

```
Object create_reference (
    in CORBA::RepositoryId intf
) raises (WrongPolicy);
```

This operation requires the **SYSTEM_ID** policy; if not present, the WrongPolicy exception is raised.

This operation creates an object reference that encapsulates a POA-generated Object Id value and the specified interface repository id. The specified repository id, which may be a null string, will become the **type_id** of the generated object reference. A repository id that does not identify the most derived interface of the object or one of its base interfaces will result in undefined behavior.

This operation does not cause an activation to take place. The resulting reference may be passed to clients, so that subsequent requests on those references will cause the appropriate servant manager to be invoked, if one is available. The generated Object Id value may be obtained by invoking **POA::reference_to_id** with the created reference.

15.3.9.19 create_reference_with_id

```
Object create_reference_with_id (
    in ObjectId                 oid,
    in CORBA::RepositoryId      intf
);
```

This operation creates an object reference that encapsulates the specified Object Id and interface repository Id values. The specified repository id, which may be a null string, will become the **type_id** of the generated object reference. A repository id that does not identify the most derived interface of the object or one of its base interfaces will result in undefined behavior.

This operation does not cause an activation to take place. The resulting reference may be passed to clients, so that subsequent requests on those references will cause the object to be activated if necessary, or the default servant used, depending on the applicable policies.

If the **POA** has the **SYSTEM_ID** policy and it detects that the Object Id value was not generated by the system or for this POA, the **create_reference_with_id** operation may raise the BAD_PARAM system exception with standard minor code 14. An ORB is not required to detect all such invalid Object Id values, but a portable application must not invoke this operation on a POA that has the **SYSTEM_ID** policy with an Object Id value that was not previously generated by the system for that **POA**, or, if the **POA** also has the **PERSISTENT** policy, for a previous instantiation of the same **POA**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

15.3.9.20 servant_to_id

```
ObjectId servant_to_id(
    in Servant                  p_servant
) raises (ServantNotActive, WrongPolicy);
```

This operation requires the **USE_DEFAULT_SERVANT** policy or a combination of the **RETAIN** policy and either the **UNIQUE_ID** or **IMPLICIT_ACTIVATION** policies if invoked outside the context of an operation dispatched by this POA. If this operation is not invoked in the context of executing a request on the specified servant and the policies specified previously are not present, the WrongPolicy exception is raised.

This operation has four possible behaviors.

1. If the **POA** has both the **RETAIN** and the **UNIQUE_ID** policy and the specified servant is active, the Object Id associated with that servant is returned.
2. If the **POA** has both the **RETAIN** and the **IMPLICIT_ACTIVATION** policy and either the POA has the **MULTIPLE_ID** policy or the specified servant is not active, the servant is activated using a POA-generated Object Id and the Interface Id associated with the servant, and that Object Id is returned.
3. If the **POA** has the **USE_DEFAULT_SERVANT** policy, the servant specified is the default servant, and the operation is being invoked in the context of executing a request on the default servant, then the ObjectId associated with the current invocation is returned.
4. Otherwise, the ServantNotActive exception is raised.

15.3.9.21 servant_to_reference

```
Object servant_to_reference (
    in Servant                  p_servant
) raises (ServantNotActive, WrongPolicy);
```

This operation requires the **RETAIN** policy and either the **UNIQUE_ID** or **IMPLICIT_ACTIVATION** policies if invoked outside the context of an operation dispatched by this POA. If this operation is not invoked in the context of executing a request on the specified servant and the policies specified previously are not present the WrongPolicy exception is raised.

This operation has four possible behaviors.

1. If the POA has both the **RETAIN** and the **UNIQUE_ID** policy and the specified servant is active, an object reference encapsulating the information used to activate the servant is returned.
2. If the **POA** has both the **RETAIN** and the **IMPLICIT_ACTIVATION** policy and either the **POA** has the **MULTIPLE_ID** policy or the specified servant is not active, the servant is activated using a POA-generated Object Id and the Interface Id associated with the servant, and a corresponding object reference is returned.
3. If the operation was invoked in the context of executing a request on the specified servant, the reference associated with the current invocation is returned.
4. Otherwise, the ServantNotActive exception is raised.

NOTE: The allocation of an Object Id value and installation in the Active Object Map caused by implicit activation may actually be deferred until an attempt is made to externalize the reference. The real requirement here is that a reference is produced that will behave appropriately (that is, yield a consistent Object Id value when asked politely).

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

15.3.9.22 reference_to_servant

```
Servant reference_to_servant (
    in Object                   reference
) raises (ObjectNotActive, WrongAdapter, WrongPolicy);
```

The table below summarizes the behavior of this operation based on the **RETAIN** policy, the **USE_DEFAULT_POLICY** and the Object in question:

RETAIN	USE_DEFAULT_SERVANT	Object	Action
Present	Present	In AOM	Return Servant from AOM
Present	Absent	In AOM	Return Servant from AOM
Present	Present	Not in AOM	Return Default Servant
Present	Absent	Not in AOM	Raise ObjectNotActive
Absent	Present		Return Default Servant
Absent	Absent		Raise Wrong Policy

If the object reference was not created by this **POA**, the WrongAdapter exception is raised.

15.3.9.23 reference_to_id

```
ObjectId reference_to_id(
    in Object                   reference
) raises (WrongAdapter, WrongPolicy);
```

The WrongPolicy exception is declared to allow future extensions.

This operation returns the Object Id value encapsulated by the specified **reference**. This operation is valid only if the reference was created by the POA on which the operation is being performed. If the reference was not created by that POA, a WrongAdapter exception is raised. The object denoted by the reference does not have to be active for this operation to succeed.

15.3.9.24 id_to_servant

```
Servant id_to_servant(
    in ObjectId                 oid
) raises (ObjectNotActive, WrongPolicy);
```

This operation requires the **RETAIN** policy or the **USE_DEFAULT_SERVANT** policy. If neither policy is present, the WrongPolicy exception is raised.

If the POA has the **RETAIN** policy and the specified **ObjectId** is in the Active Object Map, this operation returns the servant associated with that object in the Active Object Map. Otherwise, if the POA has the **USE_DEFAULT_SERVANT** policy and a default servant has been registered with the POA, this operation returns the default servant. Otherwise the ObjectNotActive exception is raised.

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

15.3.9.25 id_to_reference

```
Object id_to_reference(
    in ObjectId                 oid
) raises (ObjectNotActive, WrongPolicy);
```

This operation requires the **RETAIN** policy; if not present, the WrongPolicy exception is raised. If an object with the specified Object Id value is currently active, a reference encapsulating the information used to activate the object is returned. If the Object Id value is not active in the POA, an ObjectNotActive exception is raised.

15.3.9.26 id

```
readonly attribute CORBA::OctetSeq id;
```

This returns the unique id of the POA in the process in which it is created. It is for use by portable interceptors. This id is guaranteed unique for the life span of the POA in the process. For persistent POAs, this means that if a POA is created in the same path with the same name as another POA, these POAs are identical and, therefore, have the same id. For transient POAs, each POA is unique.

15.3.10 Current Operations

The **PortableServer::Current** interface, derived from **CORBA::Current**, provides method implementations with access to the identity of the object on which the method was invoked. The **Current** interface is provided to support servants that implement multiple objects, but can be used within the context of POA-dispatched method invocations on any servant. To provide location transparency, ORBs are required to support use of **Current** in the context of both locally and remotely invoked operations.

An instance of **Current** can be obtained by the application by issuing the **CORBA::ORB::resolve_initial_references("POACurrent")** operation. Thereafter, it can be used within the context of a method dispatched by the **POA** to obtain the **POA** and **ObjectId** that identify the object on which that operation was invoked.

PortableServer::Current is a local interface.

15.3.10.1 get_POA

```
POA get_POA()
    raises (NoContext);
```

This operation returns a reference to the POA implementing the object in whose context it is called. If called outside the context of a POA-dispatched operation, a NoContext exception is raised.

15.3.10.2 get_object_id

```
ObjectId get_object_id()
    raises (NoContext);
```

This operation returns the **ObjectId** identifying the object in whose context it is called. If called outside the context of a POA-dispatched operation, a NoContext exception is raised.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

15.3.10.3 get_reference

```
Object get_reference()
    raises(NoContext);
```

This operation returns a locally manufactured reference to the object in the context of which it is called. If called outside the context of a POA dispatched operation, a NoContext exception is raised.

NOTE: This reference is not guaranteed to be identical to the original reference the client used to make the invocation, and calling the **Object::is_equivalent** operation to compare the two references may not necessarily return true.

15.3.10.4 get_servant

```
Servant get_servant()
    raises(NoContext);
```

This operation returns a reference to the servant that hosts the object in whose context it is called. If called outside the context of a POA dispatched operation, a NoContext exception is raised.

15.4 IDL for PortableServer Module

```
// IDL
// File: PortableServer.idl
#ifndef _PORTABLE_SERVER_IDL_
#define _PORTABLE_SERVER_IDL_

import ::CORBA;
module PortableServer {
    typeprefix PortableServer “org.omg”;
    local interface POA;        // forward declaration
    typedef sequence<POA> POAList;

    native Servant;

    typedef CORBA::OctetSeq ObjectId;

    exception ForwardRequest {
        Object forward_reference;
    };

    // Policy interfaces

    const CORBA::PolicyType THREAD_POLICY_ID = 16;
    const CORBA::PolicyType LIFESPAN_POLICY_ID = 17;
    const CORBA::PolicyType ID_UNIQUENESS_POLICY_ID = 18;
    const CORBA::PolicyType ID_ASSIGNMENT_POLICY_ID = 19;
    const CORBA::PolicyType IMPLICIT_ACTIVATION_POLICY_ID = 20;
    const CORBA::PolicyType SERVANT_RETENTION_POLICY_ID = 21;
    const CORBA::PolicyType REQUEST_PROCESSING_POLICY_ID = 22;

    enum ThreadPolicyValue {
```


© ISO/IEC 2008 - All rights reserved
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    ORB_CTRL_MODEL,
    SINGLE_THREAD_MODEL,
    MAIN_THREAD_MODEL
};

 local interface ThreadPolicy : CORBA::Policy {
    readonly attribute ThreadPolicyValue value;
};

enum LifespanPolicyValue {
    TRANSIENT,
    PERSISTENT
};

local interface LifespanPolicy : CORBA::Policy {
    readonly attribute LifespanPolicyValue value;
};

enum IdUniquenessPolicyValue {
    UNIQUE_ID,
    MULTIPLE_ID
};

local interface IdUniquenessPolicy : CORBA::Policy {
    readonly attribute IdUniquenessPolicyValue value;
};

enum IdAssignmentPolicyValue {
    USER_ID,
    SYSTEM_ID
};

local interface IdAssignmentPolicy : CORBA::Policy {
    readonly attribute IdAssignmentPolicyValue value;
};

enum ImplicitActivationPolicyValue {
    IMPLICIT_ACTIVATION,
    NO_IMPLICIT_ACTIVATION
};

local interface ImplicitActivationPolicy : CORBA::Policy {
    readonly attribute ImplicitActivationPolicyValue value;
};

enum ServantRetentionPolicyValue {
    RETAIN,
    NON_RETAIN
};

local interface ServantRetentionPolicy : CORBA::Policy {
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        readonly attribute ServantRetentionPolicyValue value;
    };

    enum RequestProcessingPolicyValue {
        USE_ACTIVE_OBJECT_MAP_ONLY,
        USE_DEFAULT_SERVANT,
        USE_SERVANT_MANAGER
    };

    local interface RequestProcessingPolicy : CORBA::Policy {
        readonly attribute RequestProcessingPolicyValue value;
    };

    // POAManager interface

    local interface POAManager {
        exception AdapterInactive{};

        enum State {HOLDING, ACTIVE, DISCARDING, INACTIVE};

        void activate()
            raises(AdapterInactive);
        void hold_requests(
            in boolean wait_for_completion)
            raises(AdapterInactive);
        void discard_requests(
            in boolean wait_for_completion)
            raises(AdapterInactive);

        void deactivate(
            in boolean etherealize_objects,
            in boolean wait_for_completion);
        State get_state();
        string get_id();
    };

    // PoaManagerFactory

    local interface POAManagerFactory {
         typedef sequence<POAManager> POAManagerSeq;

        exception ManagerAlreadyExists {};

        POAManager create_POAManager(
            in string                   id,
             in CORBA::PolicyList       policies
        ) raises(ManagerAlreadyExists, CORBA::PolicyError);

        POAManagerSeq list();
        POAManager find(  in string id);
    };
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
// AdapterActivator interface

local interface AdapterActivator {
    boolean unknown_adapter(
        in POA parent,
        in string name);
    };

// ServantManager interface

local interface ServantManager{ };

local interface ServantActivator : ServantManager {
    Servant incarnate (
        in ObjectId             oid,
        in POA                  adapter)
    raises (ForwardRequest);

    void etherealize (
        in ObjectId             oid,
        in POA                  adapter,
        in Servant              serv,
        in boolean              cleanup_in_progress,
        in boolean              remaining_activations);
    };

local interface ServantLocator : ServantManager {
    native Cookie;
    Servant preinvoke(
        in ObjectId             oid,
        in POA                  adapter,
        in CORBA::Identifier    operation,
        out Cookie              the_cookie)
    raises (ForwardRequest);

    void postinvoke(
        in ObjectId             oid,
        in POA                  adapter,
        in CORBA::Identifier    operation,
        in Cookie               the_cookie,
        in Servant              the_servant
    );
};

// POA interface

local interface POA {
    exception AdapterAlreadyExists {};
    exception AdapterNonExistent {};
    exception InvalidPolicy {unsigned short index;};
    exception NoServant {};
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
exception ObjectAlreadyActive {};
exception ObjectNotActive {};
exception ServantAlreadyActive {};
exception ServantNotActive {};
exception WrongAdapter {};
exception WrongPolicy {};

// POA creation and destruction

POA create_POA(
    in string adapter_name,
    in POAManager a_POAManager,
    in CORBA::PolicyList policies)
raises (AdapterAlreadyExists, InvalidPolicy);

POA find_POA(
    in string adapter_name,
    in boolean activate_it)
raises (AdapterNonExistent);

void destroy(
    in boolean etherealize_objects,
    in boolean wait_for_completion);

// Factories for Policy objects

ThreadPolicy create_thread_policy(
    in ThreadPolicyValue value);
LifespanPolicy create_lifespan_policy(
    in LifespanPolicyValue value);
IdUniquenessPolicy create_id_uniqueness_policy(
    in IdUniquenessPolicyValue value);
IdAssignmentPolicy create_id_assignment_policy(
    in IdAssignmentPolicyValue value);
ImplicitActivationPolicy create_implicit_activation_policy(
    in ImplicitActivationPolicyValue value);
ServantRetentionPolicy create_servant_retention_policy(
    in ServantRetentionPolicyValue value);
RequestProcessingPolicy create_request_processing_policy(
    in RequestProcessingPolicyValue value);

// POA attributes

readonly attribute string the_name;
readonly attribute POA the_parent;
readonly attribute POAList the_children;
readonly attribute POAManager the_POAManager;
attribute AdapterActivator the_activator;

// Servant Manager registration:
```


© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
ServantManager get_servant_manager()
raises (WrongPolicy);

void set_servant_manager(
    in ServantManager imgr)
raises (WrongPolicy);

// operations for the USE_DEFAULT_SERVANT policy

Servant get_servant()
raises (NoServant, WrongPolicy);

void set_servant(in Servant p_servant)
raises (WrongPolicy);

// object activation and deactivation

ObjectId activate_object(
    in Servant p_servant)
raises (ServantAlreadyActive, WrongPolicy);

void activate_object_with_id(
    in ObjectId id,
    in Servant p_servant)
raises (ServantAlreadyActive, ObjectAlreadyActive, WrongPolicy);

void deactivate_object(
    in ObjectId oid)
raises (ObjectNotActive, WrongPolicy);

// reference creation operations

Object create_reference (
    in CORBA::RepositoryId intf)
raises (WrongPolicy);

Object create_reference_with_id (
    in ObjectId oid,
    in CORBA::RepositoryId intf
);

// Identity mapping operations:

ObjectId servant_to_id(
    in Servant p_servant)
raises (ServantNotActive, WrongPolicy);

Object servant_to_reference(
    in Servant p_servant)
raises (ServantNotActive, WrongPolicy);
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        Servant reference_to_servant(
            in Object reference)
        raises(ObjectNotActive, WrongAdapter, WrongPolicy);

        ObjectId reference_to_id(
            in Object reference)
        raises (WrongAdapter, WrongPolicy);

        Servant id_to_servant(
            in ObjectId oid)
        raises (ObjectNotActive, WrongPolicy);

        Object id_to_reference(in ObjectId oid)
            raises (ObjectNotActive, WrongPolicy);

        readonly attribute CORBA::OctetSeq id;
        readonly attribute POAManagerFactory the_POAManagerFactory;
    };

    // Current interface

    local interface Current : CORBA::Current {
        exception NoContext { };

        POA get_POA()
        raises (NoContext);

        ObjectId get_object_id()
            raises (NoContext);

        Object get_reference()
            raises(NoContext);

        Servant get_servant()
            raises(NoContext);
    };
};
```

15.5 UML Description of PortableServer

The following diagrams were generated by an automated tool and then annotated with the cardinalities of the associations. They are intended to be an aid in comprehension to those who enjoy such representations. They are not normative.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

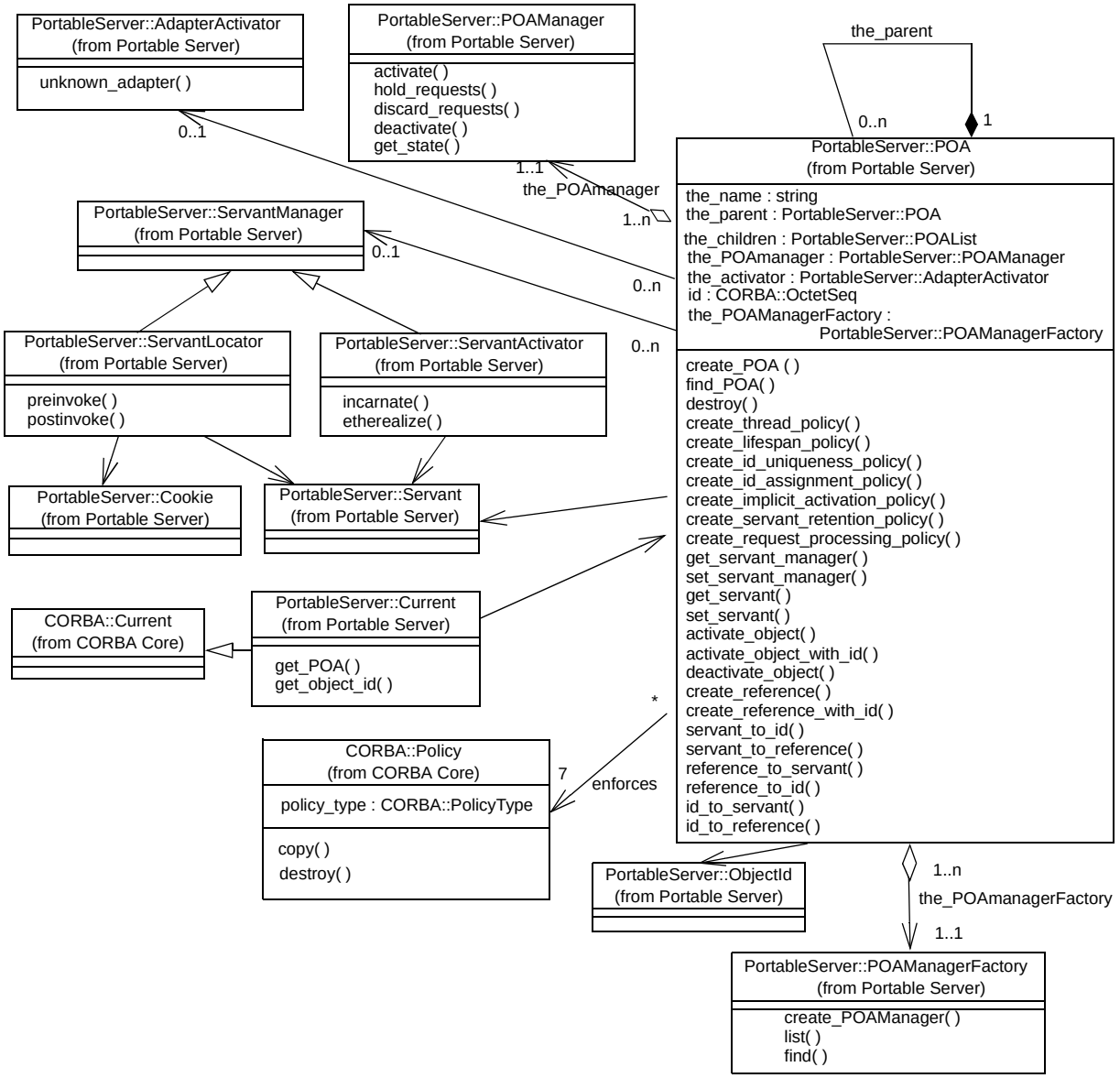


Figure 15.1 - UML for main part of PortableServer

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

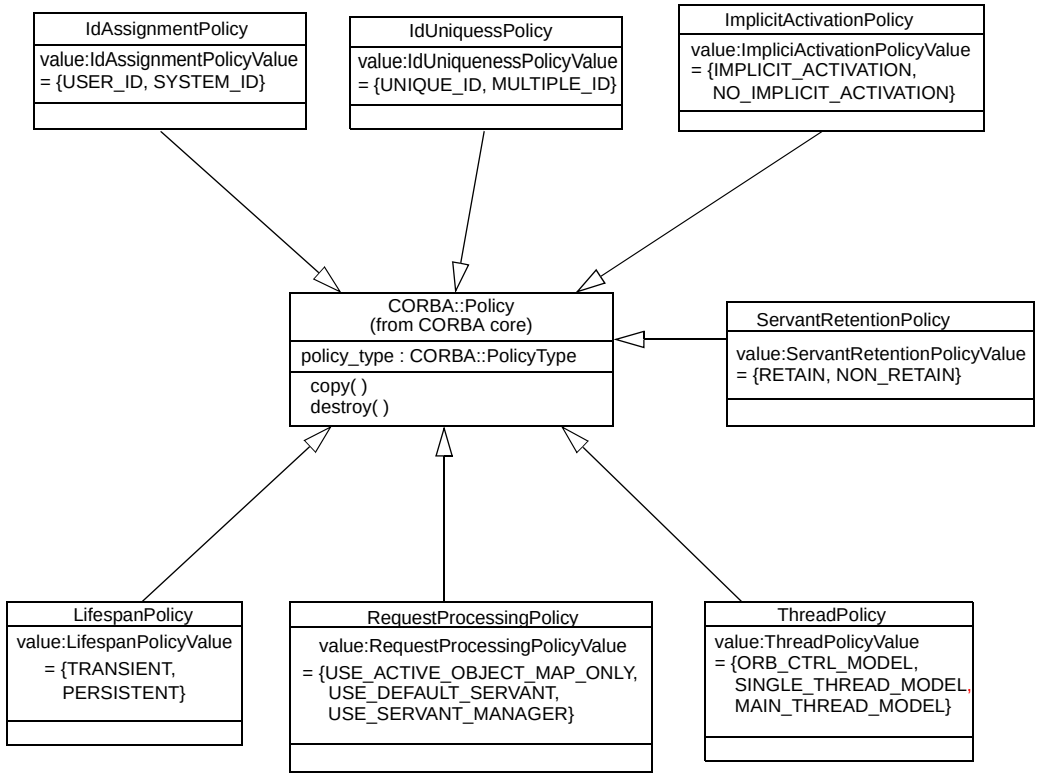


Figure 15.2 - UML for PortableServer Policies

15.6 Usage Scenarios

This sub clause illustrates how different capabilities of the POA may be used in applications.

NOTE: In some of the following C++ examples, PortableServer names are not explicitly scoped. It is assumed that all the examples have the C++ statement: **`using namespace PortableServer;`**

15.6.1 Getting the Root POA

All server applications must obtain a reference to the root POA, either to use it directly to manage objects, or to create new POA objects. The following example demonstrates how the application server can obtain a reference to the root POA.

```
// C++
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
CORBA::Object_ptr pfobj =
orb->resolve_initial_references("RootPOA");
PortableServer::POA_ptr rootPOA;
rootPOA = PortableServer::POA::narrow(pfobj);
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

15.6.2 Creating a POA

For a variety of reasons, a server application might want to create a new POA. The POA is created as a child of an existing POA. In this example, it is created as a child of the root POA.

```
// C++
CORBA::PolicyList policies(2);
policies.length(2);
policies[0] = rootPOA->create_thread_policy(
PortableServer::ThreadPolicy::ORB_CTRL_MODEL);
policies[1] = rootPOA->create_lifespan_policy(
PortableServer::LifespanPolicy::TRANSIENT);
PortableServer::POA_ptr poa =
rootPOA->create_POA("my_little_poa",
PortableServer::POAManager::_nil(), policies);
```

15.6.3 Explicit Activation with POA-assigned Object Ids

By specifying the **SYSTEM_ID** policy on a POA, objects may be explicitly activated through the POA without providing a user-specified identity value. Using this approach, objects are activated by performing the **activate_object** operation on the POA with the object in question. For this operation, the POA allocates, assigns, and returns a unique identity value for the object.

Generally this capability is most useful for transient objects, where the Object Id needs to be valid only as long as the servant is active in the server. The Object Ids can remain completely hidden and no servant manager need be provided. When this is the case, the identity and lifetime of the servant and the abstract object are essentially equivalent. When POA-assigned Object Ids are used with persistent objects or objects that are activated on demand, the application must be able to associate the generated Object Id value with its corresponding object state.

This example illustrates a simple implementation of transient objects using POA-assigned Object Ids. It presumes a POA that has the **SYSTEM_ID**, **USE_SERVANT_MANAGER**, and **RETAIN** policies.

Assume this interface:

```
// IDL
    interface Foo {
        long doit();
    };
```

This might result in the generation of the following skeleton:

```
class POA_Foo : public ServantBase
{
   public:
          ...
   virtual CORBA::Long doit() = 0;
}
```

Derive your implementation:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
class MyFooServant : public POA_Foo
{
   public:
         MyFooServant(POA_ptr poa, Long value)
         : my_poa(POA::_duplicate(poa)), my_value(value) {}
         ~MyFooServant() {CORBA::release(my_poa);}
         virtual POA_ptr _default_POA()
            {return POA::_duplicate(my_poa);}
         virtual Long doit() {return my_value;}
   protected:
         POA_ptr my_poa;
         Long my_value;
};
```

Now, somewhere in the program during initialization, probably in **main()**:

```
            MyFooServant* afoo = new MyFooServant(poa,27);
   PortableServer::ObjectId_var oid =
         poa->activate_object(afoo);
   Foo_var foo = afoo->_this();
   poa->the_POAManager()->activate();
   orb->run();
```

This object is activated with a generated Object Id.

15.6.4 Explicit Activation with User-assigned Object Ids

An object may be explicitly activated by a server using a user-assigned identity. This may be done for several reasons. For example, a programmer may know that certain objects are commonly used, or act as initial points of contact through which clients access other objects (for example, factories). The server could be implemented to create and explicitly activate these objects during initialization, avoiding the need for a servant manager.

If an implementation has a reasonably small number of servants, the server may be designed to keep them all active continuously (as long as the server is executing). If this is the case, the implementation need not provide a servant manager. When the server initializes, it could create all available servants, loading their state and identities from some persistent store. The POA supports an explicit activation operation, **activate_object_with_id**, that associates a servant with an Object Id. This operation would be used to activate all of the existing objects managed by the server during server initialization. Assuming the POA has the **USE_SERVANT_MANAGER** policy and no servant manager is associated with a POA, any request received by the POA for an Object Id value not present in the Active Object Map will result in an **OBJ_ADAPTER** exception.

In simple cases of well-known, long-lived objects, it may be sufficient to activate them with well-known Object Id values during server initialization, before activating the POA. This approach ensures that the objects are always available when the POA is active, and doesn't require writing a servant manager. It has severe practical limitations for a large number of objects, though.

This example illustrates the explicit activation of an object using a user-chosen Object Id. This example presumes a POA that has the **USER_ID**, **USE_SERVANT_MANAGER**, and **RETAIN** policies.

The code is like the previous example, but replace the last portion of the example shown above with the following code:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
// C++
MyFooServant* afoo = new MyFooServant(poa, 27);
PortableServer::ObjectId_var oid =
   PortableServer::string_to_ObjectId("myLittleFoo");
poa->activate_object_with_id(oid.in(), afoo);
Foo_var foo = afoo->_this();
```

15.6.5 Creating References before Activation

It is sometimes useful to create references for objects before activating them. This example extends the previous example to illustrate this option:

```
// C++
PortableServer::ObjectId_var oid =
PortableServer::string_to_ObjectId("myLittleFoo");
CORBA::Object_var obj = poa->create_reference_with_id(
         oid.in(), "IDL:Foo:1.0");
Foo_var foo = Foo::_narrow(obj);

// ...later...
MyFooServant* afoo = new MyFooServant(poa, 27);
poa->activate_object_with_id(oid.in(), afoo);
```

15.6.6 Servant Manager Definition and Creation

Servant managers are object implementations, and are required to satisfy all of the requirements of object implementations necessary for their intended function. Because servant managers are local objects, and their use is limited to a single narrow role, some simplifications in their implementation are possible. Note that these simplifications are suggestions, not normative requirements. They are intended as examples of ways to reduce the programming effort required to define servant managers.

A servant manager implementation must provide the following things:

- implementation code for either
  - **incarnate( )** and **etherealize( )**, or
  - **preinvoke( )** and **postinvoke( )**
- implementation code for the servant operations, as for all servants

The first two are obvious; their content is dictated by the requirements of the implementation that the servant manager is managing. For the third point, the default servant manager on the root POA already supplies this implementation code. User-written servant managers will have to provide this themselves.

Since servant managers are objects, they themselves must be activated. It is expected that most servant managers can be activated on the root POA with its default set of policies (see POA Creation on page 306). It is for this reason that the root POA has the **IMPLICIT_ACTIVATION** policy so that a servant manager can easily be activated. Users may choose to activate a servant manager on other POAs.

The following is an example servant manager to activate objects on demand. This example presumes a POA that has the **USER_ID**, **USE_SERVANT_MANAGER**, and **RETAIN** policies.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Since RETAIN is in effect, the type of servant manager used is a **ServantActivator**. The ORB supplies a servant activator skeleton class in a library:

```
// C++
namespace POA_PortableServer
{
   class ServantActivator : public virtual ServantManager
   {
         public:
            virtual ~ServantActivator();
            virtual Servant incarnate(
               const ObjectId& POA_ptr poa) = 0;
            virtual void etherealize(
               const ObjectId&, POA_ptr poa,
               Servant, Boolean remaining_activations) = 0;
   };
};
```

A **ServantActivator** servant manager might then look like:

```
// C++
class MyFooServantActivator : public
   POA_PortableServer::ServantActivator
{
   public:
         // ...
         Servant incarnate(
            const ObjectId& oid, POA_ptr poa)
         {
            String_var s = PortbleServer::ObjectId_to_string
                           (oid);
            if (strcmp(s, "myLittleFoo") == 0) {
               return new MyFooServant(poa, 27);
            else {
               throw CORBA::OBJECT_NOT_EXIST();
            }
         }

               void etherealize(
                     const ObjectId& oid,
                     POA_ptr poa,
                     Servant servant,
                     Boolean remaining_activations)
         {
            if (remaining_activations == 0)
               delete servant;
         }
         // ...
};
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

15.6.7 Object Activation on Demand

The precondition for this scenario is the existence of a client with a reference for an object with which no servant is associated at the time the client makes a request on the reference. It is the responsibility of the ORB, in collaboration with the POA and the server application to find or create an appropriate servant and perform the requested operation on it. Such an object is said to be *incarnated* (or *incarnation*) when it has an active servant. Note that the client had to obtain the reference in question previously from some source. From the client's perspective, the abstract object exists as long as it holds a reference, until it receives an OBJECT_NOT_EXIST system exception in a reply from an attempted request on the object. Incarnation state does not imply existence or non-existence of the abstract object.

NOTE: This specification does not address the issues of communication or server process activation, as they are immaterial to the POA interface and operation. It is assumed that the ORB activates the server if necessary, and can deliver the request to the appropriate POA.

To support object activation on demand, the server application must register a servant manager with the appropriate POA. Upon receiving the request, if the POA consults the Active Object Map and discovers that there is no active servant associated with the target Object Id, the POA invokes the **incarnate** operation on the servant manager.

NOTE: An implication that this model has for GIOP is that the object key in the request message must encapsulate the Object Id value. In addition, it may encapsulate other values as necessitated by the ORB implementation. For example, the server must be able to determine to which POA the request should be directed. It could assign a different communication endpoint to each POA so that the POA identity is implicit in the request, or it could use a single endpoint for the entire server and encapsulate POA identities in object key values. Note that this is not a concrete requirement; the object key may not actually contain any of those values. Whatever the concrete information is, the ORB and POA must be able to use it to find the servant manager, invoke activate if necessary (that requires the actual Object Id value), and/or find the active servant in some map.

The **incarnate** invocation passes the Object Id value to the servant manager. At this point, the servant manager may take any action necessary to produce a servant that it considers to be a valid incarnation of the object in question. The operation returns the servant to the POA, which invokes the operation on it. The **incarnate** operation may alternatively raise an OBJECT_NOT_EXIST system exception that will be returned to the invoking client. In this way, the user-supplied implementation is responsible for determining object existence and non-existence.

After activation, the POA maintains the association of the servant and the Object Id in the Active Object Map. (This example presumes the **RETAIN** and **USE_SERVANT_MANAGER** policies.)

As an obvious example of transparent activation, the Object Id value could contain a key for a record in a database that contains the object's state. The servant manager would retrieve the state from the database, construct a servant of the appropriate implementation class (assuming an object-oriented programming language), initialize it with the state from the database, and return it to the POA.

The example servant manager in the last sub clause (Servant Manager Definition and Creation on page 349) could be used for this scenario. Recall that the POA would have the **USER_ID**, **USE_SERVANT_MANAGER**, and **RETAIN** policies.

Given such a **ServantActivator**, all that remains is initialization code such as the following.

```
PortableServer::ObjectId_var oid =
   PortableServer::string_to_ObjectId("myLittleFoo");
CORBA::Object_var obj = poa->create_reference_with_id(
   oid, "IDL:foo:1.0");
MyFooServantActivator* fooIM = new MyFooServantActivator;
ServantActivator_var IMref = fooIM->_this();
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
poa->set_servant_manager(IMref);
poa->the_POAmanager()->activate();
orb->run();
```

15.6.8 Persistent Objects with POA-assigned Ids

It is possible to access the Object Id value assigned to an object by the POA, with the **POA::reference_to_id** operation. If the reference is for an object managed by the POA that is the operation's target, the operation will return the Object Id value, whether it was assigned by the POA or the user. By doing this, an implementation may provide a servant manager that associates the POA-allocated Object Id values with persistently stored state. It may also pass the POA-allocated Object Id values to POA operations such as **activate_object_with_id** and **create_reference_with_id**.

A POA with the **PERSISTENT** policy may be destroyed and later reinstantiated in the same or a different process. A POA with both the **SYSTEM_ID** and **PERSISTENT** policies generates Object Id values that are unique across all instantiations of the same POA.

15.6.9 Multiple Object Ids Mapping to a Single Servant

Each POA is created with a policy that indicates whether or not servants are allowed to support multiple object identities simultaneously. If a POA allows multiple identities per servant, the POA's treatment of the servants is affected in the following ways:

- Servants of the type may be explicitly activated multiple times with different identity values without raising an exception.
- A servant cannot be mapped onto or converted to an individual object reference using that POA, since the identity is potentially ambiguous.

15.6.10 One Servant for All Objects

By using the **USE_DEFAULT_SERVANT** policy, the developer can create a POA that will use a single servant to implement all of its objects. This approach is useful when there is very little data associated with each object, so little that the data can be encoded in the Object Id.

The following example illustrates this approach by using a single servant to incarnate all CORBA objects that export a given interface in the context of a server. This example presumes a POA that has the **USER_ID**, **NON_RETAIN,** and **USE_DEFAULT_SERVANT** policies.

Two interfaces are defined in IDL. The **FileDescriptor** interface is supported by objects that will encapsulate access to operations in a file associated with a file system. Global operations in a file system, such as the ones necessary to create **FileDescriptor** objects, are supported by objects that export the **FileSystem** interface.

```
// IDL
interface FileDescriptor {
    typedef sequence<octet> DataBuffer;

    long write (in DataBuffer buffer);
    DataBuffer read (
        in long num_bytes);
    void destroy ();
};
```


© ISO/IEC 2008 - All rights reserved
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
interface FileSystem {
    ...
    FileDescriptor open (
        in string file_name,
        in long flags);
    ...
};
```

Implementation of these two IDL interfaces may inherit from static skeleton classes generated by an IDL to C++ compiler as follows:

```
// C++
class FileDescriptorImpl : public POA_FileDescriptor
{
   public:
         FileDescriptorImpl(POA_ptr poa);
         ~FileDescriptorImpl();
         POA_ptr _default_POA();
         CORBA::Long write(
            const FileDescriptor::DataBuffer& buffer);
         FileDescriptor::DataBuffer* read(
            CORBA::Long num_bytes);
         void destroy();
   private:
         POA_ptr my_poa;
};

class FileSystemImpl : public POA_FileSystem
{
   public:
         FileSystemImpl(POA_ptr poa);
         ~FileSystemImpl();
         POA_ptr _default_POA();
         FileDescriptor_ptr open(
            const char* file_name, CORBA::Long flags);
   private:
         POA_ptr my_poa;
         FileDescriptorImpl* fd_servant;
};
```

A single servant may be used to serve all requests issued to all **FileDescriptor** objects created by a **FileSystem** object. The following fragment of code illustrates the steps to perform when a **FileSystem** servant is created.

```
// C++
FileSystemImpl::FileSystemImpl(POA_ptr poa)
   : my_poa(POA::_duplicate(poa))
{
   fd_servant = new FileDescriptorImpl(poa);
   poa->set_servant(fd_servant);
};
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The following fragment of code illustrates how **FileDescriptor** objects are created as a result of invoking an operation (**open**) exported by a **FileSystem** object. First, a local file descriptor is created using the appropriate operating system call. Then a CORBA object reference is created and returned to the client. The value of the local file descriptor will be used to distinguish the new **FileDescriptor** object from other **FileDescriptor** objects. Note that **FileDescriptor** objects in the example are transient, since they use the value of their file descriptors for their ObjectIds, and of course the file descriptors are only valid for the life of a process.

```
// C++
FileDescriptor_ptr
FileSystemImpl::open(
   const char* file_name, CORBA::Long flags)
{
   int fd = ::open(file_name, flags);
   ostrstream ostr;
   ostr << fd;
   PortableServer::ObjectId_var oid =
   PortableServer::string_to_ObjectId(ostr.str());
   Object_var obj = my_poa->create_reference_with_id(
         oid.in(),"IDL:FileDescriptor:1.0");
   return FileDescriptor::_narrow(obj);
};
```

Any request issued to a **FileDescriptor** object is handled by the same servant. In the context of a method invocation, the servant determines which particular object is being incarnated by invoking an operation that returns a reference to the target object and, after that, invoking **POA::reference_to_id**. In C++, the operation used to obtain a reference to the target object is **`_this()`**. Typically, the **ObjectId** value associated with the reference will be used to retrieve the state of the target object. However, in this example, such a step is not required since the only thing that is needed is the value for the local file descriptor and that value coincides with the **ObjectId** value associated with the reference.

Implementation of the **read** operation is rather simple. The servant determines which object it is incarnating, obtains the local file descriptor matching its identity, performs the appropriate operating system call, and returns the result in a **DataBuffer** sequence.

```
// C++
FileDescriptor::DataBuffer*
FileDescriptorImpl::read(CORBA::Long num_bytes)
{
   FileDescriptor_var me = _this();
   PortableServer::ObjectId_var oid =
         my_poa->reference_to_id(me.in());
   CORBA::String_var s =
         PortableServer::ObjectId_to_string(oid.in());
   istrstream is(s);
   int fd;
   is >> fd;
   CORBA::Octet* buffer = DataBuffer::alloc_buf(num_bytes);
   int len = ::read(fd, buffer, num_bytes);
   DataBuffer* result = new DataBuffer(len, len, buffer, 1);
   return result;
};
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Using a single servant per interface is useful in at least two situations.

- In one case, it may be appropriate for encapsulating access to legacy APIs that are not object-oriented (system calls in the Unix environment, as we have shown in the example).
- In another case, this technique is useful in handling scalability issues related to the number of CORBA objects that can be associated with a server. In the example above, there may be a million **FileDescriptor** objects in the same server and this would only require one entry in the ORB. Although there are operating system limitations in this respect (a Unix server is not able to open so many local file descriptors) the important point to take into account is that usage of CORBA doesn't introduce scalability problems but provides mechanisms to handle them.

15.6.11 Single Servant, Many Objects and Types, Using DSI

The ability to associate a single DSI servant with many CORBA objects is rather powerful in some scenarios. It can be the basis for development of gateways to legacy systems or software that mediates with external hardware, for example.

Usage of the DSI is illustrated in the following example. This example presumes a POA that supports the **USER_ID**, **USE_DEFAULT_SERVANT**, and **RETAIN** policies.

A single servant will be used to incarnate a huge number of CORBA objects, each of them representing a separate entry in a Database. There may be several types of entries in the Database, representing different entity types in the Database model. Each type of entry in the Database is associated with a separate interface that comprises operations supported by the Database on entries of that type. All these interfaces inherit from the **DatabaseEntry** interface. Finally, an object supporting the **DatabaseAgent** interface supports basic operations in the database such as creating a new entry, destroying an existing entry, etc.

```
// IDL
    interface DatabaseEntry {
        readonly attribute string name;
    };

    interface Employee : DatabaseEntry {
        attribute long id;
        attribute long salary;
    };
...

    interface DatabaseAgent {
        DatabaseEntry create_entry (
            in string key,
            in CORBA::Identifier entry_type,
            in NVPairSequence initial_attribute_values
        );

    void destroy_entry (
        in string key);
        ...
};
```

Implementation of the **DatabaseEntry** interface may inherit from the standard dynamic skeleton class as follows:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
// C++
class DatabaseEntryImpl :
   public PortableServer::DynamicImplementation
{
   public:
         DatabaseEntryImpl (DatabaseAccessPoint db);
         virtual void invoke (ServerRequest_ptr request);
         ~DatabaseEntryImpl ();

               virtual POA_ptr _default_POA()
         {
            return poa;
         }
};
```

On the other hand, implementation of the **DatabaseAgent** interface may inherit from a static skeleton class generated by an IDL to C++ compiler as follows:

```
// C++
class DatabaseAgentImpl :
   public DatabaseAgentImplBase
{
   protected:
         DatabaseAccessPoint mydb;
         DatabaseEntryImpl * common_servant;
   public:
         DatabaseAgentImpl ();
         virtual DatabaseEntry_ptr create_entry (
            const char * key,
            const char * entry_type,
            const NVPairSequence& initial_attribute_values
         );
         virtual void destroy_entry (const char * key);
         ~DatabaseAgentImpl ();
};
```

A single servant may be used to serve all requests issued to all **DatabaseEntry** objects created by a **DatabaseAgent** object. The following fragment of code illustrates the steps to perform when a **DatabaseAgent** servant is created. First, access to the database is initialized. As a result, some kind of descriptor (a **DatabaseAccessPoint** local object) used to operate on the database is obtained. Finally, a servant will be created and associated with the POA.

```
// C++
void DatabaseAgentImpl::DatabaseAgentImpl ()
{
   mydb = ...;
   common_servant = new DatabaseEntryImpl(mydb);
   poa->set_servant(common_servant);
};
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The code used to create **DatabaseEntry** objects representing entries in the database is similar to the one used for creating **FileDescriptor** objects in the example of the previous sub clause. In this case, a new entry is created in the database and the key associated with that entry will be used to represent the identity for the corresponding **DatabaseEntry** object. All requests issued to a **DatabaseEntry** object are handled by the same servant because references to this type of object are associated with a common POA created with the **USE_DEFAULT_SERVANT** policy.

```
// C++
DatabaseEntry_ptr DatabaseAgentImpl::create_entry (
   const char * key,
   const char * entry_type,
   const NVPairSequence& initial_attribute_values)

            // creates a new entry in the database:
   mydb->new_entry (key, ...);

            // creates a reference to the CORBA object used to
   // encapsulate access to the new entry in the database.
   // There is an interface for each entry type:
   CORBA::Object_ptr obj = poa->create_reference_with_id(
         string_to_ObjectId (key),
         identifierToRepositoryId (entry_type),
   );

            DatabaseEntry_ptr entry = DatabaseEntry::_narrow (obj);
         CORBA::release (obj);
   return entry;
};
```

Any request issued to a **DatabaseEntry** object is handled by the same servant. In the context of a method invocation, the servant determines which particular object it is incarnating, obtains the database key matching its identity, invokes the appropriate operation in the database, and returns the result as an output parameter in the **ServerRequest** object.

Sometimes, a program may need to determine the type of an entry in the database in order to invoke operations on the entry. If that is the case, the servant may obtain the type of an entry based on the interface supported by the **DatabaseEntry** object encapsulating access to that entry. This interface may be obtained by means of invoking the **get_interface** operation exported by the reference to the **DatabaseEntry** object.

```
// C++
void DatabaseEntryImpl::invoke (ServerRequest_ptr request)
{
   CORBA::Object_ptr current_obj = _this ();

            // The servant determines the key associated with
   // the database entry represented by current_obj:
   PortableServer::ObjectId oid =
         poa->reference_to_id (current_obj);
   char * key = ObjectId_to_string (oid);

            // The servant handles the incoming CORBA request. This
   // typically involves the following steps:
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
   // 1.    mapping the CORBA request into a database request
   //       using the key obtained previously
   // 2.    constructing output parameters to the CORBA request
   //       from the response to the database request
         ...
};
```

Note that in this example, we may have a billion **DatabaseEntry** objects in a server requiring only a single entry in map tables supported by the POA (that is, the ORB at the server). No permanent storage is required for references to **DatabaseEntry** objects at the server. Actually, references to **DatabaseEntry** objects will only occupy space:

- at clients, as long as those references are used; or
- at the server, only while a request is being served.

Scalability problems can be handled using this technique. There are many scenarios where this scalability causes no penalty in terms of performance (basically, when there is no need to restore the state of an object, each time a request to it is being served).

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16 Portable Interceptors

16.1 Introduction

Portable Interceptors are hooks into the ORB through which ORB services can intercept the normal flow of execution of the ORB. The following figures describe the programming model for which portable Interceptors were designed.

16.1.1 Object Creation

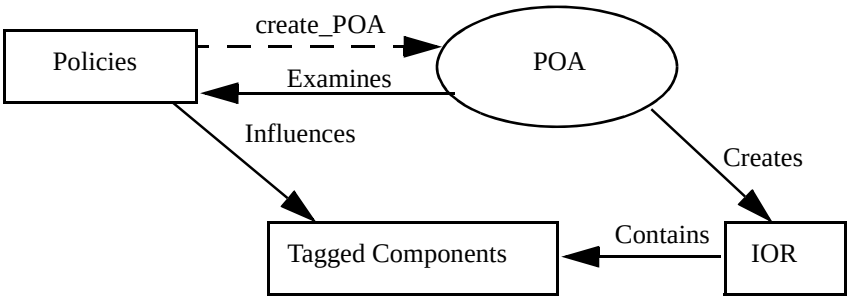


Figure 16.1 - Object Creation

Figure 16.1 shows the parts involved in the creation of an object. An object is represented by an IOR created by the POA. A set of policies is used to create a POA which influences the set of tagged components contained within the profiles of any IOR created by that POA. ORB services may have tagged components specific to their service, therefore they require a means to add tagged components to an IOR. ORB services may also introduce new policies; therefore, they require a means to create these new policies.

Requirement: Add tagged components

Satisfied by: IORInterceptor (see IOR Interceptor on page 392).

Requirement: Create policies

Satisfied by: PolicyFactory (see PolicyFactory on page 401).

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.1.2 Client Sends Request

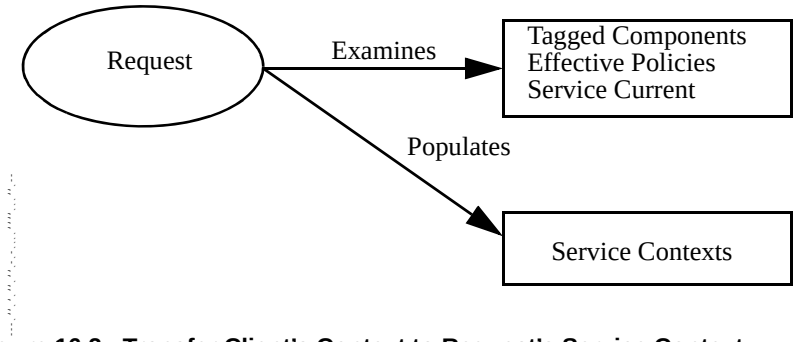


Figure 16.2 - Transfer Client's Context to Request's Service Context

Figure 16.2 shows what is needed to transfer a client's context to the service context. Service contexts are populated from information in a service's **Current** object, from the effective policies, and from information in the tagged components on an IOR's profile.

The processing of a request is an integral part of the ORB. Since each ORB service potentially creates its own service context, there must be a means by which each service can get the necessary information during request processing. Since service contexts are defined as a unique identifier and an octet sequence containing a CDR encapsulation there must be a portable method to create such an octet sequence.

Requirement:	Intercept request processing and access necessary data.
Satisfied by:	Request Interceptors (see Request Interceptors on page 363) and the PortableInterceptor::Current (see Portable Interceptor Current on page 386).

Requirement:	Convert types to octet sequences
Satisfied by:	Codec (see CORBA, Part 2: ORB Interoperability Architecture clause, Coder/Decoder Interfaces sub clause).

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.1.3 Server Receives Request

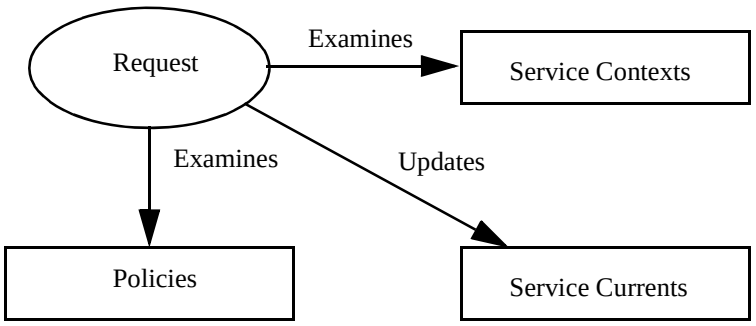


Figure 16.3 - Transfer Request's Service Context to Server's Context

On the client, the client's context is transferred to the request's service context. On the server, the opposite must occur: the information in the service context is transferred to the server's context which is then available to the server application. Figure 16.3 shows what is necessary to accomplish this.

The requirements that exist in Client Sends Request on page 360 also exist here.

16.1.4 Server Sends Reply

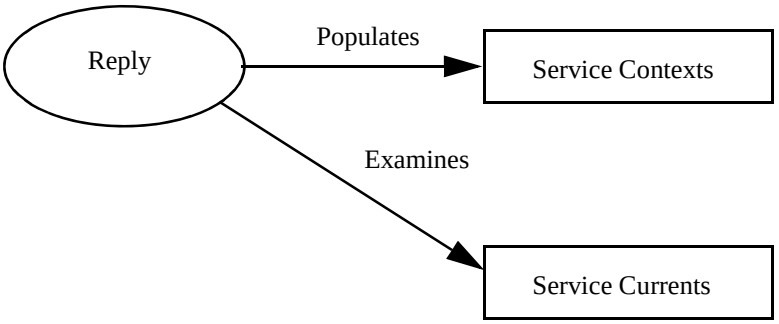


Figure 16.4 - Transfer Server's Context to Reply's Service Context

Figure 16.4 shows what is needed to transfer a server's context to a reply's service context. Service contexts are populated from information in a service's **Current** object.

The requirements which exist in Client Sends Request on page 360 also exist here.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.1.5 Client Receives Reply

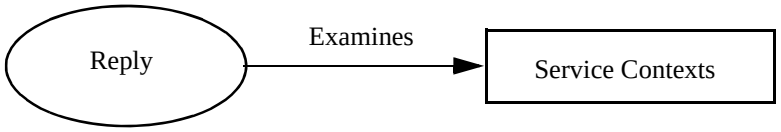


Figure 16.5 - View the Service Context on the Client Reply

When processing the client reply, although the client's context cannot be updated by the reply's service context, the service may still wish to query the service context information.

The client's context cannot be updated because such updates would be invalid on asynchronous calls. The client thread may be continually changing its context and if a reply also changed the context at any time, the state of the context at any given time would be indeterminate.

The requirements that exist in Client Sends Request on page 360 also exist here.

16.2 General Behavior of Local Objects

All local objects specified in this clause except for **Interceptor** and local interfaces derived from it, **PolicyFactory** and **ORBInitializer** override the default behavior of the **Object::get_orb** operation and return the **ORB** that the portable interceptor facility is associated with.

16.3 Interceptor Interface

Portable Interceptor interfaces and related type definitions reside in the module **PortableInterceptor**. All portable Interceptors inherit from the local interface **Interceptor**:

```
module PortableInterceptor {

    local interface Interceptor {
        readonly attribute string name;
        void destroy();
    };
};
```

Each Interceptor may have a name that may be used administratively to order the lists of Interceptors. Only one Interceptor of a given name can be registered with the ORB for each Interceptor type. An Interceptor may be anonymous; that is, have an empty string as the name attribute. Any number of anonymous Interceptors may be registered with the ORB.

Interceptor::destroy is called during **ORB::destroy**. When an application calls **ORB::destroy**, the ORB:

1. Waits for all requests in progress to complete.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

2. Calls the **Interceptor::destroy** operation for each interceptor.

3. Completes destruction of the ORB.

Method invocations from within **Interceptor::destroy** on object references for objects implemented on the ORB being destroyed result in undefined behavior. However, method invocations on objects implemented on an ORB other than the one being destroyed are permitted. (This means that the ORB being destroyed is still capable of acting as a client, but not as a server.)

16.4 Request Interceptors

A request Interceptor is designed to intercept the flow of a request/reply sequence through the ORB at specific points so that services can query the request information and manipulate the service contexts that are propagated between clients and servers.

The primary use of request Interceptors is to enable ORB services to transfer context information between clients and servers.

There are two types of request Interceptors: client-side (see Client-Side Interceptor on page 365) and server-side (see Server-Side Interceptor on page 370).

16.4.1 Design Principles

The following points are the principles followed in the design of the portable Interceptor architecture.

1. Interceptors are called on all ORB mediated invocations. The following implicit object operations may or may not be ORB mediated: **get_interface**, **is_a**, **non_existent**, **get_domain_managers, repository_id**, and **get_component**. When these are ORB mediated, Interceptors are called; when they are not ORB mediated, Interceptors are not called.

2. A request Interceptor can affect the outcome of a request by raising a system exception at any of the interception points. It can stop the request from even reaching the target by raising a system exception in the outbound path. It can alter an outcome specified by the target (exception or non-exception) by raising a system exception in the inbound path.

3. A request Interceptor can affect the outcome of a request by directing a request to a different location at any interception point other than a successful reply. That different location might include a location not otherwise reachable through the original request; that is, a location that might not be discovered by the ORB in the course of a locate request.

4. A request Interceptor cannot affect a request by changing a parameter specified by the client. That is, the Interceptor cannot modify “in” arguments.

5. A request Interceptor cannot affect a non-exception outcome by supplying the response itself. That is, the Interceptor cannot modify “out” arguments or the return value.

6. Request Interceptors are independent of other request Interceptors. That is, a request Interceptor won’t need to know, and won’t even be told, if there are request Interceptors executed before or after it. If a request Interceptor down the line (executed closer to the target than this one) affects the outcome of request, this request Interceptor will not be aware of that fact.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7. Corollary: request Interceptors can communicate between themselves to bypass this principle, but that's outside of the concerns of the model.
8. A request Interceptor may make object invocations itself before allowing the current request to execute.
9. There is no provision for making client implementations aware that any request Interceptor has been or will be called. Corollary: A client and a request Interceptor can communicate between themselves to bypass this principle, but that is outside of the concerns of the model.
10. There is no provision for making object implementations aware that any request Interceptor has been or will be called Corollary: An object implementation and a request Interceptor can communicate between themselves to bypass this principle, but that is outside of the concerns of the model.
11. To ensure the integrity of the effect of each request Interceptor, a set of general flow rules are specified that govern the flow of processing through a list of interceptors. See below.

16.4.2 General Flow Rules

Both client and server request Interceptors are registered with an ORB (see Registering Interceptors on page 401). The ORB logically maintains an ordered list of these Interceptors.

To accommodate both the client and server request Interceptors, and any future additions to the interception points list, the following general rules apply to the flow of execution of request interception points:

- There is a set of starting interception points. One and only one of these is called on any given request/reply sequence.
- There is a set of ending interception points. One and only one of these is called on any given request/reply sequence.
- There may be any number of intermediate interception points between the start and end interception points which run in sequence.
- On an exception, intermediate interception points may not be called.
- If and only if a starting interception point runs to completion is an ending interception point called.

See Client-Side Interception Point Flow on page 367 and Server-Side Interception Point Flow on page 372 for details of how these general flow rules apply specifically to the client-side and server-side Interceptors.

16.4.3 The Flow Stack Visual Model

To visualize the general flow rules, think of each Interceptor as being put on a Flow Stack when a starting interception point completes successfully. (An ORB need not implement the Flow Stack. It is presented simply as a visual cue.) An ending interception point is called for each Interceptor in the stack. If a starting interception point is called for all Interceptors, then all Interceptors will have an ending interception point called. If one of the Interceptors raises an exception during the invocation of its starting interception point, only those Interceptors on the stack at that point will be popped and have an ending interception point called.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.4.4 The Request Interceptor Points

Each request Interceptor is called at a number of interception points. Figure 16.6 shows the flow of control for a request/reply cycle that is subject to at least one request Interceptor. See Client-Side Interceptor on page 365 and Server-Side Interceptor on page 370 for descriptions of each of these interception points.

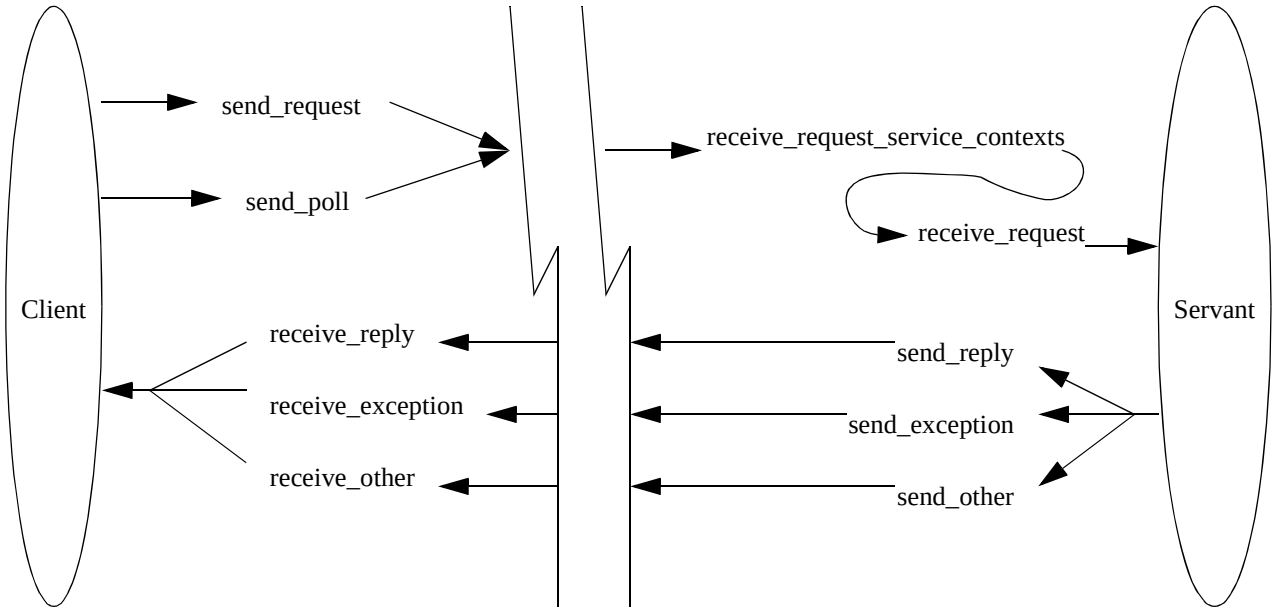


Figure 16.6 - Request Interception Points

16.4.5 Client-Side Interceptor

To write a client-side Interceptor, the **ClientRequestInterceptor** local interface shall be implemented.

```
local interface ClientRequestInterceptor : Interceptor {
    void send_request  (in ClientRequestInfo ri)
        raises (ForwardRequest);
    void send_poll (in ClientRequestInfo ri);
    void receive_reply (in ClientRequestInfo ri);
    void receive_exception (in ClientRequestInfo ri)
        raises (ForwardRequest);
    void receive_other (in ClientRequestInfo ri)
        raises (ForwardRequest);
};
```

16.4.6 Client-Side Interception Points

16.4.6.1 send_request

This interception point allows an Interceptor to query request information and modify the service context before the request is sent to the server.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This interception point may raise a system exception. If it does, no other Interceptors' **send_request** operations are called. Those Interceptors on the Flow Stack are popped and their **receive_exception** interception points are called.

This interception point may also raise a ForwardRequest exception (see ForwardRequest Exception on page 386 for details of this exception). If an Interceptor raises this exception, no other Interceptors' **send_request** operations are called. Those Interceptors on the Flow Stack are popped and their **receive_other** interception points are called.

Compliant Interceptors shall properly follow **completion_status** semantics if they raise a system exception from this interception point. The **completion_status** shall be **COMPLETED_NO**.

16.4.6.2 send_poll

This interception point allows an Interceptor to query information during a Time-Independent Invocation (TII) polling get reply sequence.

With TII, an application may poll for a response to a request sent previously by the polling client or some other client. This poll is reported to Interceptors through the **send_poll** interception point and the response is returned through the **receive_reply** or **receive_exception** interception points. If the response is not available before the poll time-out expires, the system exception TIMEOUT is raised and **receive_exception** is called with this exception.

This interception point may raise a system exception. If it does, no other Interceptors' **send_poll** operations are called. Those Interceptors on the Flow Stack are popped and their **receive_exception** interception points are called.

Compliant Interceptors shall properly follow **completion_status** semantics if they raise a system exception from this interception point. The **completion_status** shall be **COMPLETED_NO**.

16.4.6.3 receive_reply

This interception point allows an Interceptor to query the information on a reply after it is returned from the server and before control is returned to the client.

This interception point may raise a system exception. If it does, no other Interceptors' **receive_reply** operations are called. The remaining Interceptors in the Flow Stack shall have their **receive_exception** interception point called.

Compliant Interceptors shall properly follow **completion_status** semantics if they raise a system exception from this interception point. The **completion_status** shall be **COMPLETED_YES**.

16.4.6.4 receive_exception

When an exception occurs, this interception point is called. It allows an Interceptor to query the exception's information before it is raised to the client.

This interception point may raise a system exception. This has the effect of changing the exception, which successive Interceptors popped from the Flow Stack receive on their calls to **receive_exception**. The exception raised to the client will be the last exception raised by an Interceptor, or the original exception if no Interceptor changes the exception.

This interception point may also raise a ForwardRequest exception (see Section 16.4.15, "ForwardRequest Exception," on page 386 for details on this exception). If an Interceptor raises this exception, no other Interceptors' **receive_exception** operations are called. The remaining Interceptors in the Flow Stack are popped and have their **receive_other** interception point called.

If the **completion_status** of the exception is not **COMPLETED_NO**, then it is inappropriate for this interception point to raise a ForwardRequest exception. The request's at-most-once semantics would be lost.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Compliant Interceptors shall properly follow **completion_status** semantics if they raise a system exception from this interception point. If the original exception is a system exception, the **completion_status** of the new exception shall be the same as on the original. If the original exception is a user exception, then the **completion_status** of the new exception shall be **COMPLETED_YES**.

Under some conditions, depending on what policies are in effect, an exception (such as COMM_FAILURE) may result in a retry of the request. While this retry is a new request with respect to Interceptors, there is one point of correlation between the original request and the retry: because control has not returned to the client, the **PortableInterceptor::Current** for both the original request and the retrying request is the same (see Portable Interceptor Current on page 386).

16.4.6.5 receive_other

This interception point allows an Interceptor to query the information available when a request results in something other than a normal reply or an exception. For example, a request could result in a retry (for example, a GIOP Reply with a **LOCATION_FORWARD** status was received); or on asynchronous calls, the reply does not immediately follow the request, but control shall return to the client and an ending interception point shall be called.

For retries, depending on the policies in effect, a new request may or may not follow when a retry has been indicated. If a new request does follow, while this request is a new request with respect to Interceptors, there is one point of correlation between the original request and the retry. Because control has not returned to the client, the request scoped **PortableInterceptor::Current** for both the original request and the retrying request is the same (see Portable Interceptor Current on page 386).

This interception point may raise a system exception. If it does, no other Interceptors' **receive_other** operations are called. The remaining Interceptors in the Flow Stack are popped and have their **receive_exception** interception point called.

This interception point may also raise a ForwardRequest exception (see ForwardRequest Exception on page 386 for details on this exception). If an Interceptor raises this exception, successive Interceptors' **receive_other** operations are called with the new information provided by the ForwardRequest exception.

Compliant Interceptors shall properly follow **completion_status** semantics if they raise a system exception from this interception point. The **completion_status** shall be **COMPLETED_NO**. If the target invocation had completed, this interception point would not be called.

16.4.7 Client-Side Interception Point Flow

A **ClientRequestInterceptor** instance is registered with the ORB. The ORB logically maintains an ordered list of client-side Interceptors. The Interceptor list is traversed in order on the sending interception points and in reverse order on the receiving interception points.

16.4.7.1 Client-side Flow Rules

The client-side flow rules are derived from the general flow rules (see General Flow Rules on page 364):

- The set of starting interception points is: **send_request** and **send_poll**. One and only one of these is called on any given request/reply sequence.
- The set of ending interception points is: **receive_reply**, **receive_exception**, **receive_other**. One and only one of these is called on any given request/reply sequence.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- There are no intermediate exception points.
- If and only if **send_request** or **send_poll** runs to completion is an ending interception point called.

16.4.7.2 Additional Client-side Details

If, during request processing, a request is canceled because of an ORB shutdown, **receive_exception** is called with the system exception BAD_INV_ORDER with a minor code of 4 (ORB has shutdown).

If a request is canceled for any other reason (for example, a GIOP cancel message is sent by the ORB), **receive_exception** is called with the system exception TRANSIENT with a standard minor code of 2.

On oneway requests, returning control to the client may occur immediately or it may return after the target has performed the operation, or somewhere in-between depending on the SyncScope (see sync_scope on page 377). Regardless of the SyncScope, if there is no exception, **receive_other** is called before control is returned to the client.

Asynchronous requests are simply two separate requests. The first request receives no reply. The second receives a normal reply. So the normal (no exceptions) flow is: first request - **send_request** followed by **receive_other**; second request - **send_request** followed by **receive_reply**.

If during receive_reply the transaction contexts in the TSC and RSC do not match, then raise the system exception BAD_INV_ORDER with standard minor code 21.

16.4.7.3 Client-side Flow Examples

Given the client-side flow rules, here are some concrete examples:

- For successful invocations: **send_request** is followed by **receive_reply** - a start point is followed by an end point.
- For retries: **send_request** is followed by **receive_other** - a start point is followed by an end point.
- For a DII deferred synchronous invocation or AMI invocation using the polling model, **send_request** is followed by **receive_other** (when the invocation is successfully initiated) or **receive_exception** (if the invocation could not be initiated).
- For successful DII polls (using Request::get_response or ORB::get_next_response) or AMI polls (using valuetypes derived from Messaging::Poller), **send_poll** is followed by **receive_reply** - a start point is followed by an end point.
- For DII polls (using Request::get_response or ORB::get_next_response) or AMI polls (using valuetypes derived from Messaging::Poller), whose response is unavailable, **send_poll** is followed by **receive_exception** - a start point is followed by an end point.
- for AMI invocations using the callback model, **send_request** is followed by **receive_other** (when the invocation is succesfully initiated) or **receive_exception** (if the invocation could not be initiated). Any reply is treated as a separate invocation on the callback handler object.

For the following exception scenarios, assume we have Interceptors A, B, and C. On the send interception points they are called in the order A, B, C; on the receive interception points they are called in the order C, B, A.

Scenario

An exception arrives from the server:

- **A.send_request** is called;
- **B.send_request** is called;
- **C.send_request** is called;

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- **C.receive_exception** is called;
- **B.receive_exception** is called;
- **A.receive_exception** is called.

In this scenario you can see that the flow for each Interceptor follows the rules. They are all: **send_request** followed by **receive_exception** - a start point is followed by an end point.

Scenario

B.send_request raises an exception:

- **A.send_request** is called;
- **B.send_request** is called and raises an exception
- **A.receive_exception** is called.

In this scenario you can see that the flow for each Interceptor follows the rules:

- The flow for A is **send_request** followed by **receive_exception** - a start point is followed by an end point.
- The flow for B is **send_request** - a start point did not complete, so no end point was called; B raised the exception, so there is no need to tell it that the exception occurred.
- The flow for C is non-existent since the exception occurred before any of C's interception points was invoked - a start point was not called, so no end point is called.

Scenario

A reply returns successfully from the server, but **B.receive_reply** raises an exception:

- **A.send_request** is called;
- **B.send_request** is called;
- **C.send_request** is called;
- **C.receive_reply** is called;
- **B.receive_reply** is called and raises an exception;
- **A.receive_exception** is called.

In this scenario you can see that the flow for each Interceptor follows the rules:

- The flow for A is **send_request** followed by **receive_exception** - a start point is followed by an end point.
- The flow for B is **send_request** followed by **receive_reply** - a start point is followed by an end point.
- The flow for C is **send_request** followed by **receive_reply** - a start point is followed by an end point.

The scenario for B raising an exception at **receive_other** is similar to the scenario where B raises an exception at **receive_reply**.

Scenario

An exception X is returned by the server, but **B.receive_exception** changes the exception to Y:

- **A.send_request** is called;
- **B.send_request** is called;
- **C.send_request** is called;

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- **C.receive_exception** is called with X;
- **B.receive_exception** is called with X, raises Y;
- **A.receive_exception** is called with Y.

In this scenario, the flow for all Interceptors is **send_request** followed by **receive_exception** - a start point followed by an end point - Interceptor A is handed exception Y while the B and C are handed exception X.

16.4.8 Server-Side Interceptor

To write a server-side Interceptor, the **ServerRequestInterceptor** local interface shall be implemented.

```
local interface ServerRequestInterceptor : Interceptor {
    void receive_request_service_contexts (in ServerRequestInfo ri)
         raises (ForwardRequest);
    void receive_request (in ServerRequestInfo ri)
        raises (ForwardRequest);
    void send_reply (in ServerRequestInfo ri);
    void send_exception (in ServerRequestInfo ri)
        raises (ForwardRequest);
    void send_other (in ServerRequestInfo ri) raises (ForwardRequest);
};
```

16.4.9 Server-Side Interception Points

16.4.9.1 receive_request_service_contexts

At this interception point, Interceptors must get their service context information from the incoming request transfer it to **PortableInterceptor::Current**'s slots (see Portable Interceptor Current on page 386 for details on the relationship between **receive_request_service_contexts** and **PortableInterceptor::Current**).

This interception point is called before the servant manager is called. Operation parameters are not yet available at this point. This interception point may or may not execute in the same thread as the target invocation.

This interception point may raise a system exception. If it does, no other Interceptors' **receive_request_service_contexts** operations are called. Those Interceptors on the Flow Stack are popped and their **send_exception** interception points are called.

This interception point may also raise a ForwardRequest exception (see ForwardRequest Exception on page 386 for details on this exception). If an Interceptor raises this exception, no other Interceptors' **receive_request_service_contexts** operations are called. Those Interceptors on the Flow Stack are popped and their **send_other** interception points are called.

Compliant Interceptors shall properly follow **completion_status** semantics if they raise a system exception from this interception point. The **completion_status** shall be **COMPLETED_NO**.

16.4.9.2 receive_request

This interception point allows an Interceptor to query request information after all the information, including operation parameters, are available. This interception point shall execute in the same thread as the target invocation.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

In the DSI model, since the parameters are first available when the user code calls **arguments**, **receive_request** is called from within **arguments**. It is possible that **arguments** is not called in the DSI model. The target may call **set_exception** before calling **arguments**. The ORB shall guarantee that **receive_request** is called once, either through **arguments** or through **set_exception**. If it is called through **set_exception**, requesting the arguments will result in NO_RESOURCES being raised with a standard minor code of 1.

This interception point may raise a system exception. If it does, no other Interceptors' **receive_request** operations are called. Those Interceptors on the Flow Stack are popped and their **send_exception** interception points are called.

This interception point may also raise a ForwardRequest exception (see ForwardRequest Exception on page 386 for details on this exception). If an Interceptor raises this exception, no other Interceptors' **receive_request** operations are called. Those Interceptors on the Flow Stack are popped and their **send_other** interception points are called.

Compliant Interceptors shall properly follow **completion_status** semantics if they raise a system exception from this interception point. The **completion_status** shall be **COMPLETED_NO**.

16.4.9.3 send_reply

This interception point allows an Interceptor to query reply information and modify the reply service context after the target operation has been invoked and before the reply is returned to the client. This interception point shall execute in the same thread as the target invocation.

This interception point may raise a system exception. If it does, no other Interceptors' **send_reply** operations are called. The remaining Interceptors in the Flow Stack shall have their **send_exception** interception point called.

Compliant Interceptors shall properly follow **completion_status** semantics if they raise a system exception from this interception point. The **completion_status** shall be **COMPLETED_YES**.

16.4.9.4 send_exception

When an exception occurs, this interception point is called. It allows an Interceptor to query the exception information and modify the reply service context before the exception is raised to the client. This interception point shall execute in the same thread as the target invocation.

This interception point may raise a system exception. This has the effect of changing the exception that successive Interceptors popped from the Flow Stack receive on their calls to **send_exception**. The exception raised to the client will be the last exception raised by an Interceptor, or the original exception if no Interceptor changes the exception.

This interception point may also raise a ForwardRequest exception (see ForwardRequest Exception on page 386 for details on this exception). If an Interceptor raises this exception, no other Interceptors' **send_exception** operations are called. The remaining Interceptors in the Flow Stack shall have their **send_other** interception points called.

If the **completion_status** of the exception is not **COMPLETED_NO**, then it is inappropriate for this interception point to raise a ForwardRequest exception. The request's at-most-once semantics would be lost.

Compliant Interceptors shall properly follow **completion_status** semantics if they raise a system exception from this interception point. If the original exception is a system exception, the **completion_status** of the new exception shall be the same as on the original. If the original exception is a user exception, then the **completion_status** of the new exception shall be **COMPLETED_YES**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.4.9.5 send_other

This interception point allows an Interceptor to query the information available when a request results in something other than a normal reply or an exception. A request could result in a retry (for example, a GIOP Reply with a **LOCATION_FORWARD** status was received). This interception point shall execute in the same thread as the target invocation.

This interception point may raise a system exception. If it does, no other Interceptors' **send_other** operations are called. The remaining Interceptors in the Flow Stack shall have their **send_exception** interception points called.

This interception point may also raise a ForwardRequest exception (see ForwardRequest Exception on page 386 for details on this exception). If an Interceptor raises this exception, successive Interceptors' **send_other** operations are called with the new information provided by the ForwardRequest exception.

Compliant Interceptors shall properly follow **completion_status** semantics if they raise a system exception from this interception point. The **completion_status** shall be **COMPLETED_NO**.

16.4.10 Server-Side Interception Point Flow

A **ServerRequestInterceptor** instance is registered with the ORB (see Registering Interceptors on page 401). The ORB logically maintains an ordered list of server-side Interceptors. The Interceptor list is traversed in order on the receiving interception points and in reverse order on the sending interception points.

16.4.10.1 Server-side Flow Rules

The server-side flow rules are derived from the general flow rules (see General Flow Rules on page 364).

- The starting interception point is **receive_request_service_contexts**; this interception point is called on any given request/reply sequence.
- The set of ending interception points is **send_reply**, **send_exception**, **send_other**. One and only one of these is called on any given request/reply sequence.
- The intermediate interception point is **receive_request**, which is called after **receive_request_service_contexts** and before an ending interception point.
- On an exception, **receive_request** may not be called.
- If and only if **receive_request_service_contexts** runs to completion is an ending interception point called.

16.4.10.2 Additional Server-side Details

If, during request processing, a request is canceled because of an ORB shutdown, **send_exception** is called with the system exception BAD_INV_ORDER with a minor code of 4 (ORB has shutdown).

If a request is canceled for any other reason (for example, a GIOP cancel message has been received), **send_exception** is called with the system exception TRANSIENT with a standard minor code of 3.

The following statement is made about the GIOP close connection message (CORBA v2.3 15-45):

> "If the ORB sending the **CloseConnection** is a server, or bidirectional GIOP is in use, the sending ORB must not currently be processing any Requests from the other side."

With respect to portable Interceptors, "...processing any Requests..." means that **receive_request_service_contexts** has been called on any Interceptor and no ending interception point has yet been invoked.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

On oneway requests, there is no reply sent to the client; however, the target is called and the server can construct an empty reply. Since closure is necessary, this reply is tracked and **send_reply** is called (unless an exception occurs, in which case **send_exception** is called).

Asynchronous requests, from the server's point of view, are just normal synchronous requests. Normal interception point flows are followed.

If a POA and a servant locator are present, the order of their operations and interception points is:

1. **ServerRequestInterceptor.receive_request_service_contexts;**
2. **ServantLocator.preinvoke;**
3. **ServerRequestInterceptor.receive_request**
4. the operation
5. **ServantLocator.postinvoke;**
6. **ServerRequestInterceptor send_reply**, **send_exception**, or **send_other**.

preinvoke, the operation, and **postinvoke** are required to execute in the same thread (see ServantLocator Interface on page 323). Since **receive_request** occurs within this chain, **receive_request** shall also execute in the same thread.

postinvoke executes in the same thread as **preinvoke** in order for **postinvoke** to perform any necessary closure processing. Likewise, the sending interception points (**send_reply**, **send_exception**, or **send_other**) shall also execute in the same thread.

16.4.10.3 Server-side Flow Examples

Given the server-side flow rules, here are some concrete examples.

For successful invocations, the chain of interception points, in order, is: **receive_request_service_contexts**, **receive_request**, **send_reply** - a start point is followed by an intermediate point, which is followed by an end point.

For the following exception scenarios, assume we have Interceptors A, B, and C. On the receive interception points they are called in the order A, B, C; on the send interception points they are called in the order C, B, A.

Scenario

An exception is raised by the target:

- **A.receive_request_service_contexts** is called;
- **B.receive_request_service_contexts** is called;
- **C.receive_request_service_contexts** is called;
- **A.receive_request** is called;
- **B.receive_request** is called;
- **C.receive_request** is called;
- **C.send_exception** is called;
- **B.send_exception** is called;
- **A.send_exception** is called.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

In this scenario you can see that the flow for each Interceptor follows the rules. The chain for all is: **receive_request_service_contexts**, **receive_request**, **send_exception** - a start point is followed by an intermediate point that is followed by an end point.

Scenario

B.receive_request_service_contexts raises an exception:

- **A.receive_request_service_contexts** is called;
- **B.receive_request_service_contexts** is called and raises an exception;
- **A.send_exception** is called.;

In this scenario you can see that the flow for each Interceptor follows the rules:

- The flow for A is **receive_request_service_contexts** followed by **send_exception** - a start point followed by an end point, no intermediate points are called.
- The flow for B is **receive_request_service_contexts** - a start point did not complete, so no end point was called; B raised the exception, so there is no need to tell it that the exception occurred.
- The flow for C is non-existent since the exception occurred before any of C's interception points were invoked.

Scenario

B.receive_request raises an exception:

- **A.receive_request_service_contexts** is called;
- **B.receive_request_service_contexts** is called;
- **C.receive_request_service_contexts** is called;
- **A.receive_request** is called;
- **B.receive_request** is called and raises an exception;
- **C.send_exception** is called;
- **B.send_exception** is called;
- **A.send_exception** is called.

In this scenario you can see that the flow for each Interceptor follows the rules:

- Since the **receive_request_service_contexts** starting point ran to completion then, no matter what happens in intermediate points, a "terminating" interception point must be called for all interceptors.

16.4.10.4 Scenario

The target invocation returns successfully, but **B.send_reply** raises an exception:

- **A.receive_request_service_contexts** is called;
- **B.receive_request_service_contexts** is called;
- **C.receive_request_service_contexts** is called;
- **A.receive_request** is called;
- **B.receive_request** is called;
- **C.receive_request** is called;

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- **C.send_reply** is called;
- **B.send_reply** is called and raises an exception;
- **A.send_exception** is called.

In this scenario you can see that the flow for each Interceptor follows the rules:

- The flow for A is: **receive_request_service_contexts**, **receive_request**, **send_exception** - a start point is followed by an intermediate point that is followed by an end point.
- The flow for B is **receive_request_service_contexts**, **receive_request**, **send_reply** - a start point is followed by intermediate point, which is followed by an end point.
- The flow for C is: **receive_request_service_contexts**, **receive_request**, **send_reply** - a start point is followed by an intermediate point which is followed by an end point.

The scenario for B raising an exception at **send_other** is similar to the scenario where B raises an exception at **send_reply**.

Scenario

An exception X is raised by the target, but **B.send_exception** changes the exception to Y:

- **A.receive_request_service_contexts** is called;
- **B.receive_request_service_contexts** is called;
- **C.receive_request_service_contexts** is called;
- **A.receive_request** is called;
- **B.receive_request** is called;
- **C.receive_request** is called;
- **C.send_exception** is called with X;
- **B.send_exception** is called with X, raises Y;
- **A.send_exception** is called with Y.

In this scenario, the flow for all Interceptors is **receive_request_service_contexts**, **receive_request**, **send_exception** - a start point is followed by an intermediate point, which is followed by an end point; Interceptor A is handed exception Y while the B and C are handed exception X.

16.4.11 Request Information

Each interception point is given an object through which the Interceptor can access request information. Client-side and server-side interception points are concerned with different information, so there are two information objects: **ClientRequestInfo** is passed to the client-side interception points and **ServerRequestInfo** is passed to the server-side interception points. But there is information that is common to both, so they both inherit from a common interface: **RequestInfo**.

16.4.12 RequestInfo Interface

```
local interface RequestInfo {
    readonly attribute unsigned long request_id;
    readonly attribute string operation;
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        readonly attribute Dynamic::ParameterList arguments;
        readonly attribute Dynamic::ExceptionList exceptions;
        readonly attribute Dynamic::ContextList contexts;
        readonly attribute Dynamic::RequestContext operation_context;
        readonly attribute any result;
        readonly attribute boolean response_expected;
        readonly attribute Messaging::SyncScope sync_scope;
        readonly attribute ReplyStatus reply_status;
        readonly attribute Object forward_reference;
        any get_slot (in SlotId id) raises (InvalidSlot);
        IOP::ServiceContext get_request_service_context (
            in IOP::ServiceId id);
        IOP::ServiceContext get_reply_service_context (
            in IOP::ServiceId id);
    };
```

The details of the attributes and operations on **RequestInfo** follow. Some of these are not valid at all interception points. See Table 16.1 on page 380 and Table 16.2 on page 383.

16.4.12.1 request_id

This ID uniquely identifies an active request/reply sequence. Once a request/reply sequence is concluded this ID may be reused.

Note that this id is not the same as the GIOP **request_id**. If GIOP is the transport mechanism used, then these IDs may very well be the same, but this is not guaranteed nor required.

16.4.12.2 operation

This attribute is the name of the operation being invoked.

16.4.12.3 arguments

This attribute is a **Dynamic::ParameterList** containing the arguments on the operation being invoked (see NVList PIDL Represented by ParameterList IDL on page 410). If there are no arguments, this attribute will be a zero length sequence.

Not all environments provide access to the arguments. With the Java portable bindings, for example, the arguments are not available. In these environments, when this attribute is accessed, NO_RESOURCES will be raised with a standard minor code of 1.

16.4.12.4 exceptions

This attribute is a **Dynamic::ExceptionList** describing the **TypeCodes** of the user exceptions that this operation invocation may raise (see ExceptionList PIDL Represented by ExceptionList IDL on page 410). If there are no user exceptions, this attribute will be a zero length sequence.

Not all environments provide access to the exception list. With the Java portable bindings, for example, the exception list is not available. In these environments, when this attribute is accessed, NO_RESOURCES will be raised with a standard minor code of 1.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.4.12.5 contexts

This attribute is a **Dynamic::ContextList** describing the contexts that may be passed on this operation invocation (see ContextList PIDL Represented by ContextList IDL on page 410). If there are no contexts, this attribute will be a zero length sequence.

Not all environments provide access to the context list. With the Java portable bindings, for example, the context list is not available. In these environments, when this attribute is accessed, NO_RESOURCES will be raised with a standard minor code of 1.

16.4.12.6 operation_context

This attribute is a **Dynamic::RequestContext** containing the contexts being sent on the request (see Context PIDL Represented by RequestContext IDL on page 410).

Not all environments provide access to the context. With the Java portable bindings, for example, the context is not available. In these environments, when this attribute is accessed, NO_RESOURCES will be raised with standard minor code of 1.

16.4.12.7 result

This attribute is an **any** containing the result of the operation invocation.

If the operation return type is **void**, this attribute will be an **any** containing a type code with a **TCKind** value of **tk_void** and no value.

Not all environments provide access to the result. With the Java portable bindings, for example, the result is not available. In these environments, when this attribute is accessed, NO_RESOURCES will be raised with a standard minor code of 1.

16.4.12.8 response_expected

This boolean attribute indicates whether a response is expected.

On the client, a reply is not returned when **response_expected** is false, so **receive_reply** cannot be called. **receive_other** is called unless an exception occurs, in which case **receive_exception** is called.

On the client, within **send_poll**, this attribute is **true**.

16.4.12.9 sync_scope

This attribute, defined in the Messaging specification, is pertinent only when **response_expected** is **false**. If **response_expected** is **true**, the value of **sync_scope** is undefined. It defines how far the request shall progress before control is returned to the client. This attribute may have one of the following values:

Messaging::SYNC_NONE
Messaging::SYNC_WITH_TRANSPORT
Messaging::SYNC_WITH_SERVER
Messaging::SYNC_WITH_TARGET

On the server, for all scopes, a reply will be created from the return of the target operation call, but the reply will not return to the client. Although it does not return to the client, it does occur, so the normal server-side interception points are followed; that is, **receive_request_service_contexts**, **receive_request**, **send_reply**, or **send_exception**.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

For **SYNC_WITH_SERVER** the server does send an empty reply back to the client before the target is invoked. This reply is not intercepted by server-side Interceptors.

16.4.12.10 reply_status

This attribute describes the state of the result of the operation invocation. Its value can be one of the following:

PortableInterceptor::SUCCESSFUL
PortableInterceptor::SYSTEM_EXCEPTION
PortableInterceptor::USER_EXCEPTION
PortableInterceptor::LOCATION_FORWARD
PortableInterceptor::TRANSPORT_RETRY
PortableInterceptor::UNKNOWN

On the client:

- Within the **receive_reply** interception point, this attribute will only be **SUCCESSFUL**.
- Within the **receive_exception** interception point, this attribute will be either **SYSTEM_EXCEPTION** or **USER_EXCEPTION**.
- Within the receive_other interception point, this attribute will be any of: **SUCCESSFUL**, **LOCATION_FORWARD**, **TRANSPORT_RETRY**, or **UNKNOWN**. **SUCCESSFUL** means an asychronous request has been successfully initiated. **LOCATION_FORWARD** means that a reply came back with **LOCATION_FORWARD** as its status. **TRANSPORT_RETRY** means that the transport mechanism indicated a retry - a GIOP reply with a status of **NEEDS_ADDRESSING_MODE**, for instance. **UNKNOWN** means that the **ORB** was unable to determine the correct status. This can occur for example in the Java language mapping when the optimized path for a collocated call is used.

On the server:

- Within the **send_reply** interception point, this attribute will only be **SUCCESSFUL**.
- Within the **send_exception** interception point, this attribute will be either **SYSTEM_EXCEPTION** or **USER_EXCEPTION**.
- Within the send_other interception point, this attribute will be any of: **SUCCESSFUL**, **LOCATION_FORWARD**, or **UNKNOWN**. SUCCESSFUL means an asynchronous request returned successfully. **LOCATION_FORWARD** means that a reply came back with **LOCATION_FORWARD** as its status. **UNKNOWN** means that the ORB was unable to determine the correct status. This can occur for example in the Java language mapping when the optimized path for a collocated call is used.

16.4.12.11 forward_reference

If the **reply_status** attribute is **LOCATION_FORWARD**, then this attribute will contain the object to which the request will be forwarded. It is indeterminate whether a forwarded request will actually occur.

16.4.12.12 get_slot

This operation returns the data from the given slot of the **PortableInterceptor::Current** that is in the scope of the request.

If the given slot has not been set, then an **any** containing a type code with a **TCKind** value of **tk_null** is returned.

If the ID does not define an allocated slot, **InvalidSlot** is raised.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

See Portable Interceptor Current on page 386 for an explanation of slots and the **PortableInterceptor::Current**.

Parameter(s)

- **id**
  The SlotId of the slot that is to be returned.

Return Value

The slot data, in the form of an any, obtained with the given identifier.

16.4.12.13 get_request_service_context

This operation returns a copy of the service context with the given ID that is associated with the request.

If the request's service context does not contain an entry for that ID, BAD_PARAM with a standard minor code of 26 is raised.

Parameter(s)

- **id**
  The IOP::ServiceId of the service context that is to be returned.

Return Value

The IOP::ServiceContext obtained with the given identifier.

16.4.12.14 get_reply_service_context

This operation returns a copy of the service context with the given ID that is associated with the reply.

If the request's service context does not contain an entry for that ID, BAD_PARAM with a standard minor code of 26 is raised.

Parameter(s)

- **id**
  The IOP::ServiceId of the service context that is to be returned.

Return Value

The IOP::ServiceContext obtained with the given identifier.

16.4.13 ClientRequestInfo Interface

```
local interface ClientRequestInfo : RequestInfo {
    readonly attribute Object target;
    readonly attribute Object effective_target;
    readonly attribute IOP::TaggedProfile effective_profile;
    readonly attribute any received_exception;
    readonly attribute CORBA::RepositoryId received_exception_id;
    IOR::TaggedComponent get_effective_component (
        in IOP::ComponentId id);
    IOP::TaggedComponentSeq get_effective_components (
        in IOP::ComponentId id);
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        CORBA::Policy get_request_policy (in CORBA::PolicyType type);
        void add_request_service_context (
            in IOP::ServiceContext service_context,
            in boolean replace);
    };
```

Some attributes and operations on **ClientRequestInfo** are not valid at all interception points. Table 16.1 shows the validity of each attribute or operation. If it is not valid, attempting to access it will result in a BAD_INV_ORDER being raised with a standard minor code of 14.

Table 16.1

	send_ request	**send_poll**	**receive_ reply**	**receive_ exception**	**receive_ other**
request_id	yes	yes	yes	yes	yes
operation	yes	yes	yes	yes	yes
arguments	yes_1	no	yes	no	no
exceptions	yes	no	yes	yes	yes
contexts	yes	no	yes	yes	yes
operation_context	yes	no	yes	yes	yes
result	no	no	yes	no	no
response_expected	yes	yes	yes	yes	yes
sync_scope	yes	no	yes	yes	yes
reply_status	no	no	yes	yes	yes
forward_reference	no	no	no	no	yes_2
get_slot	yes	yes	yes	yes	yes
get_request_service_context	yes	no	yes	yes	yes
get_reply_service_context	no	no	yes	yes	yes
target	yes	yes	yes	yes	yes
effective_target	yes	yes	yes	yes	yes
effective_profile	yes	yes	yes	yes	yes
received_exception	no	no	no	yes	no
received_exception_id	no	no	no	yes	no
get_effective_component	yes	no	yes	yes	yes
get_effective_components	yes	no	yes	yes	yes
get_request_policy	yes	no	yes	yes	yes
add_request_service_context	yes	no	no	no	no

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

1 When **ClientRequestInfo** is passed to **send_request**, there is an entry in the list for every argument, whether in, inout, or out. But only the in and inout arguments will be available.

2 If the **reply_status** attribute is not **LOCATION_FORWARD**, accessing this attribute will raise BAD_INV_ORDER with a standard minor code of 14.

16.4.13.1 target

This attribute is the object that the client called to perform the operation. See effective_target on page 381.

16.4.13.2 effective_target

This attribute is the actual object on which the operation will be invoked. If the **reply_status** is **LOCATION_FORWARD**, then on subsequent requests, **effective_target** will contain the forwarded IOR while **target** will remain unchanged.

16.4.13.3 effective_profile

This attribute is the profile that will be used to send the request. If a location forward has occurred for this operation's object and that object's profile changed accordingly, then this profile will be that located profile.

16.4.13.4 received_exception

This attribute is an **any** that contains the exception to be returned to the client.

If the exception is a user exception that cannot be inserted into an any (for example, it is unknown or the bindings don't provide the **TypeCode**), then this attribute will be an any containing the system exception UNKNOWN with a standard minor code of 1. However, the **RepositoryId** of the exception is available in the **received_exception_id** attribute.

16.4.13.5 received_exception_id

This attribute is the **CORBA::RepositoryId** of the exception to be returned to the client.

16.4.13.6 get_effective_component

This operation returns the **IOP::TaggedComponent** with the given ID from the profile selected for this request.

If there is more than one component for a given component ID, it is undefined which component this operation returns. If there is more than one component for a given component ID, **get_effective_components** should be called instead.

If no component exists for the given component ID, this operation will raise BAD_PARAM with a standard minor code of 28.

Parameter(s)

- **id**

  The IOP:: ComponentId of the component that is to be returned.

Return Value

The IOP::TaggedComponent obtained with the given identifier.

16.4.13.7 get_effective_components

This operation returns all the tagged components with the given ID from the profile selected for this request. This sequence is in the form of an **IOP::TaggedComponentSeq**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

If no component exists for the given component ID, this operation will raise BAD_PARAM with a standard minor code of 28.

Parameter(s)

- **id**
  The IOP:: ComponentId of the components that are to be returned.

Return Value

The IOP::TaggedComponentSeq, each component of which contains the given identifier.

16.4.13.8 get_request_policy

This operation returns the given policy in effect for this operation.

If the policy type is not valid either because the specified type is not supported by this ORB or because a policy object of that type is not associated with this Object, INV_POLICY with a standard minor code of 2 is raised.

Parameter(s)

- **id**
  The CORBA::PolicyType that specifies the policy to be returned.

Return Value

The CORBA::Policy obtained with the given type.

16.4.13.9 add_request_service_context

This operation allows Interceptors to add service contexts to the request. There is no declaration of the order of the service contexts. They may or may not appear in the order that they are added.

Parameter(s)

- **service_context**
  The IOP::ServiceContext to be added to the request.
- **replace**
  Indicates the behavior of this operation when a service context already exists with the given ID. If false, then BAD_INV_ORDER with a standard minor code of 15 is raised. If true, then the existing service context is replaced by the new one.

16.4.14 ServerRequestInfo Interface

```
local interface ServerRequestInfo : RequestInfo {
    readonly attribute any sending_exception;
    readonly attribute CORBA::OctetSeq object_id;
    readonly attribute CORBA::OctetSeq adapter_id;
    readonly attribute ServerId server_id ;
    readonly attribute ORBId orb_id ;
    readonly attribute AdapterName adapter_name;
    readonly attribute CORBA::RepositoryId
        target_most_derived_interface;
    CORBA::Policy get_server_policy (in CORBA::PolicyType type);
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        void set_slot (in SlotId id, in any data) raises (InvalidSlot);
        boolean target_is_a (in CORBA::RepositoryId id);
        void add_reply_service_context (
            in IOP::ServiceContext service_context,
            in boolean replace);
    };
```

Some attributes and operations on **ServerRequestInfo** are not valid at all interception points. Table 16.2 shows the validity of each attribute or operation. If it is not valid, attempting to access it will result in a BAD_INV_ORDER being raised with a standard minor code of 14.

Table 16.2

	receive_request_ service_contexts	**receive_ request**	**send_reply**	**send_ exception**	**send_ other**
request_id	yes	yes	yes	yes	yes
operation	yes	yes	yes	yes	yes
arguments	no	yes_1	yes	no_2	no_2
exceptions	no	yes	yes	yes	yes
contexts	no	yes	yes	yes	yes
operation_context	no	yes	yes	no	no
result	no	no	yes	no	no
response_expected	yes	yes	yes	yes	yes
sync_scope	yes	yes	yes	yes	yes
reply_status	no	no	yes	yes	yes
forward_reference	no	no	no	no	yes_2
get_slot	yes	yes	yes	yes	yes
get_request_service_context	yes	yes	yes	yes	yes
get_reply_service_context	no	no	yes	yes	yes
sending_exception	no	no	no	yes	no
object_id	no	yes	yes	yes_3	yes_3
adapter_id	no	yes	yes	yes_3	yes_3
server_id	no	yes	yes	yes	yes
orb_id	no	yes	yes	yes	yes
adapter_name	no	yes	yes	yes	yes
target_most_derived_interface	no	yes	no_4	no_4	no_4
get_server_policy	yes	yes	yes	yes	yes

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 16.2

	receive_request_ service_contexts	receive_ request	send_reply	send_ exception	send_ other
set_slot	yes	yes	yes	yes	yes
target_is_a	no	yes	no[4]	no[4]	no[4]
add_reply_service_context	yes	yes	yes	yes	yes

1 When **ServerRequestInfo** is passed to **receive_request**, there is an entry in the list for every argument, whether in, inout, or out. But only the in and inout arguments will be available.

2 If the **reply_status** attribute is not **LOCATION_FORWARD**, accessing this attribute will raise BAD_INV_ORDER with a standard minor code of 14.

3 If the servant locator caused a location forward, or raised an exception, this attribute/operation may not be available in this interception point. **NO_RESOURCES** with a standard minor code of 1 will be raised if it is not available.

4 The operation is not available in this interception point because the necessary information requires access to the target object's servant, which may no longer be available to the ORB. For example, if the object's adapter is a POA that uses a **ServantLocator**, then the ORB invokes the interception point after it calls **ServantLocator::postinvoke()**.

16.4.14.1 sending_exception

This attribute is an **any** that contains the exception to be returned to the client.

If the exception is a user exception that cannot be inserted into an any (for example, it is unknown or the bindings don't provide the **TypeCode**), then this attribute will be an any containing the system exception UNKNOWN with a standard minor code of 1.

16.4.14.2 object_id

This attribute is the opaque **object_id** describing the target of the operation invocation.

16.4.14.3 adapter_id

This attribute is the opaque identifier for the object adapter.

16.4.14.4 server_id

The value of the **server_id** attribute is the value that was passed into the **ORB::init** call (see Server ID on page 117) using the **-ORBServerId** argument when the ORB was created.

16.4.14.5 orb_id

The value of the **orb_id** attribute is the value that was passed into the **ORB::init** call.

In Java, this is accomplished using the **-ORBid** argument in the `ORB.init` call that created the ORB containing the object adapter that created this template. What happens if the same ORBid is used on multiple `ORB::init` calls in the same server is currently undefined.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

16.4.14.6 adapter_name

The **adapter_name** attribute defines a name for the object adapter that services requests for the invoked object. In the case of the POA, the adapter_name is the sequence of names from the root POA to the POA that services the request. The name of the root POA is the sequence containing only the string “RootPOA.”

16.4.14.7 target_most_derived_interface

This attribute is the **RepositoryID** for the most derived interface of the servant.

16.4.14.8 get_server_policy

This operation returns the policy in effect for this operation for the given policy type. The returned **CORBA::Policy** object shall only be a policy whose type was registered via **register_policy_factory** (see register_policy_factory on page 406).

If a policy for the given type was not registered via **register_policy_factory**, this operation will raise INV_POLICY with a standard minor code of 3.

Parameter(s)

- **type**
  The CORBA::PolicyType that specifies the policy to be returned.

Return Value

The CORBA::Policy obtained with the given policy type.

16.4.14.9 set_slot

This operation allows an Interceptor to set a slot in the **PortableInterceptor::Current** that is in the scope of the request. If data already exists in that slot, it will be overwritten.

If the ID does not define an allocated slot, InvalidSlot is raised.

See Portable Interceptor Current on page 386 for an explanation of slots and **PortableInterceptor::Current.**

Parameter(s)

- **id**
  The SlotId of the slot.
- **data**
  the data, in the form of an any, to store in that slot.

16.4.14.10 target_is_a

This operation returns **true** if the servant is the given **RepositoryId**, **false** if it is not.

Parameter(s)

- id
  The caller wants to know if the servant is this CORBA::RepositoryId.

Return Value

Is the servant the given RepositoryId?

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.4.14.11 add_reply_service_context

This operation allows Interceptors to add service contexts to the request. There is no declaration of the order of the service contexts. They may or may not appear in the order that they are added.

Parameter(s)

- **service_context**
  The IOP::ServiceContext to add to the reply.
- **replace**
  Indicates the behavior of this operation when a service context already exists with the given ID. If false, then BAD_INV_ORDER with a standard minor code of 11 is raised. If true, then the existing service context is replaced by the new one

16.4.15 ForwardRequest Exception

```
exception ForwardRequest {
    Object forward;
};
```

The ForwardRequest exception is the means by which an Interceptor can indicate to the ORB that a retry of the request should occur with the new object given in the exception. This behavior of causing a retry only occurs if the ORB receives a ForwardRequest from an interceptor. If ForwardRequest is raised anywhere else, it is passed through the ORB as is normal for a user exception.

If an Interceptor raises a ForwardRequest exception in response to a call of an interceptor, no other Interceptors are called for that interception point. The remaining Interceptors in the Flow Stack shall have their appropriate ending interception point called: **receive_other** on the client, or **send_other** on the server. The **reply_status** in the **receive_other** or **send_other** shall be **LOCATION_FORWARD**.

16.5 Portable Interceptor Current

16.5.1 Overview

The **PortableInterceptor::Current** object (hereafter referred to as **PICurrent**) is a **Current** object that is used specifically by portable Interceptors to transfer thread context information to a request context. Portable Interceptors are not required to use **PICurrent**. But if information from a client's thread context is required at an Interceptor's interception points, then **PICurrent** can be used to propagate that information. **PICurrent** allows portable service code to be written regardless of an ORB's threading model.

On the client side, this information includes, but is not limited to, thread context information that shall be propagated to the server via a service context.

On the server side, this information includes, but is not limited to, service context information received from the client which is propagated to the target's thread context.

16.5.2 Obtaining the Portable Interceptor Current

Before an invocation is made, **PICurrent** is obtained via a call to **ORB::resolve_initial_references ("PICurrent")**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

From within the interception points, the data on **PICurrent** that has moved from the thread scope to the request scope is available via the **get_slot** operation on the **RequestInfo** object. A **PICurrent** can still be obtained via **resolve_initial_references**, but that is the Interceptor's thread scope **PICurrent**. See Request Scope vs Thread Scope on page 390 for a detailed discussion of the scope of **PICurrent**.

16.5.3 Portable Interceptor Current Interface

```
module PortableInterceptor {
    typedef unsigned long SlotId;
    exception InvalidSlot {};
    local interface Current : CORBA::Current {

        any get_slot (in SlotId id) raises (InvalidSlot);
        void set_slot (in SlotId id, in any data) raises (InvalidSlot);
    };
};
```

PICurrent is merely a slot table, the slots of which are used by each service to transfer their context data between their context and the request's or reply's service context. Each service that wishes to use **PICurrent** reserves a slot or slots at initialization time (see allocate_slot_id on page 406) and uses those slots during the processing of requests and replies.

16.5.3.1 get_slot

A service can get the slot data it set in **PICurrent** via **get_slot**. The data is in the form of an **any**.

- If the given slot has not been set, an **any** containing a type code with a **TCKind** value of **tk_null** and no value is returned.
- If **get_slot** is called on a slot that has not been allocated, InvalidSlot is raised.
- If **get_slot** is called from within an ORB initializer (see Registering Interceptors on page 401), BAD_INV_ORDER with a minor code of 10 shall be raised

Parameter(s)

- **id**
  The SlotId of the slot from which the data will be returned

Return Value

The data, in the form of an any, of the given slot identifier.

16.5.3.2 set_slot

A service sets data in a slot with **set_slot**. The data shall be in the form of an **any**.

- If data already exists in that slot, it is overridden.
- If **set_slot** is called on a slot that has not been allocated, InvalidSlot is raised
- If **set_slot** is called from within an ORB initializer (see Registering Interceptors on page 401), BAD_INV_ORDER with a minor code of 10 shall be raised.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Parameter(s)

- **id**
  The SlotId of the slot to which the data will be set.
- **data**
  The data, in the form of an any, which will be set to the identified slot.

16.5.4 Use of Portable Interceptor Current

16.5.4.1 Client-side use of PICurrent

PICurrent is merely a slot table. Before a request, a service's **Current** can store its context specific data into a slot in **PICurrent**. When a request begins, **PICurrent**'s context transitions from a thread context to a request context. (That is, the ORB logically makes a copy of the current **PICurrent** and places that copy on the request. Note that this could be a lazy copy. A copy would only be necessary if **PICurrent** were modified. Since a copy may never actually be made, the term "logical copy" is used in this sub clause.) Each service's Interceptor now has access to the data that its **Current** put into **PICurrent**'s slot table. In other words, each service's Interceptor now has access to the data within the calling client's thread context even though the request processing may be in a different thread.

For example, see the following pseudo-code. Within its **ORBInitializer** (see ORBInitializer Interface on page 401), the transaction service allocates a slot:

```
PortableInterceptor::SlotId mySlotId =
   orb_init_info.allocate_slot_id ();
```

When a transaction begins, the Transaction's **Current** is called, which can place its context information in a slot on **PICurrent**:

```
any myData = ...; // get data from Transaction's Current
PortableInterceptor::Current pic =
   orb.resolve_initial_references ("PICurrent");
pic.set_slot (mySlotId, myData);
```

When an operation invocation begins, the ORB logically copies **PICurrent** from the thread context to the request context and the slots are available to Interceptors via the **ClientRequestInfo** object. So the transaction service's Interceptor could look like:

```
any myData = info.get_slot (mySlotId);
IOP::ServiceContext sc = ...;// convert myData to a SC
info.add_request_service_context (sc);
```

The request scope **PICurrent** slots are read-only on the client. There is no **set_slot** on the **ClientRequestInfo** object.

16.5.4.2 Example of PICurrent to Handle Client-side Recursion

If an Interceptor itself makes an operation invocation, it shall have some means of breaking infinite recursion. For example: the client calls operation X; **send_request** is called, which calls operation Y; **send_request** is called, which again calls operation Y; and so on unless the implementation of **send_request** breaks the recursion.

Recursion can be broken using **PICurrent**. If an Interceptor knows it will recurse, it allocates a slot in **PICurrent** in its **ORBInitializer** (see ORBInitializer Interface on page 401) that it will use for recursion:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
PortableInterceptor::SlotId recurseId =
   orb_init_info.allocate_slot_id ();
```

At the point at which it recurses, say in **send_request**, it does so in a manner similar to the following:

```
any recurse = info.get_slot (recurseId);

// if we haven't yet recursed, then the slot will be empty.
if (recurse.type () == tk_null)
{
   // Fill in the recurse slot before making
   // the recursive call.
   any recurseFlag = new any;
   recurseFlag.insert_boolean (true);
   PortableInterceptor::Current pic =
         orb.resolve_initial_references ("PICurrent");
   pic.set_slot (recurseId, recurseFlag);
   // Now make the recursive call.
   someObject.someOperation ();
}
```

When a client calls operation X, **send_request** is invoked for operation X. The recurse slot is empty, so the **if** block is executed: the recurse slot is set to **true** for this thread's **PICurrent** and the recursive call to **someOperation** is made. **send_request** is again invoked, this time for **someOperation**. This time the recurse slot is not empty, so the **if** block is not executed and the recursive call is not made, thus breaking the recursion.

16.5.4.3 Server-side use of PICurrent

The service contexts associated with the request may be propagated, using **PICurrent**, to the context of the thread that will execute the operation. The request's **PICurrent** is read and written via the **get_slot** and **set_slot** operations on **ServerRequestInfo**.

receive_request_service_contexts shall populate the slots of the request scope **PICurrent**. The ORB logically copies this **PICurrent** to the thread scope after processing the **receive_request_service_contexts** list.

When the operation invocation completes, the send interception points still have read/write access to the request scope **PICurrent**.

For example, within its **ORBInitializer** (see ORBInitializer Interface on page 401), the transaction service allocates a slot:

```
PortableInterceptor::SlotId mySlotId =
   orb_init_info.allocate_slot_id ();
```

The Transaction Interceptor can move the transaction information from the service context list to **PICurrent**:

```
IOP::ServiceContext sc =
   info.get_request_service_context (transactionId);
any myData = // convert SC to an any
info.set_slot (mySlotId, myData);
```

Within a server thread, the Transaction service can transfer its information from **PICurrent** to the **TransactionCurrent**:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
PortableInterceptor::Current pic =
   orb.resolve_initial_references ("PICurrent");
any myData = pic.get_slot (mySlotId);
// Copy myData into the current context.
```

16.5.4.4 Request Scope vs Thread Scope

The thread scope **PICurrent** is the **PICurrent** that exists within a thread's context. A request scope **PICurrent** is the **PICurrent** associated with the request. On the client-side, the thread scope **PICurrent** is logically copied to the request scope **PICurrent** from the thread's context when a request begins and is attached to the **ClientRequestInfo** object. On the server-side, the request scope **PICurrent** is attached to the **ServerRequestInfo** and follows the request processing. It is logically copied to the thread scope **PICurrent** after the list of **receive_request_service_contexts** interception points are processed.

16.5.4.5 Flow of PICurrent between Scopes

For the following, TSC means Thread Scope **PICurrent**; and RSC means Request Scope **PICurrent**. Refer to Figure 16.1 on page 391 for a graphical representation of the following discussion. The numbered points below correspond to the numbers in Figure 16.1.

Before operation invocation, the client thread may read and write the TSC. On a synchronous operation invocation, the flow proceeds as follows:

1. The invocation proceeds to the ORB.
2. Before the sending interception points are called, a TSC is logically copied to the request scope.
3. The sending interception points are called. They have read-only access to this RSC. They may add entries to the service context list based on the slot data in the RSC.
4. On the server, an empty RSC is created. Interceptors shall populate this RSC from the service context list in **receive_request_service_contexts**.
5. The ORB logically copies the RSC to the server-side TSC after the **receive_request_service_contexts** points are processed and before the servant manager is called. This TSC is within the context for the **receive_request** points, the invocation of the servant manager, and the invocation of the target operation. The **receive_request** points may modify the RSC, but this no longer affects the TSC. The **receive_request** points are called. These points have access to the RSC - though modifying the RSC at this point has no affect on the TSC. Since these points execute in the same thread as the target operation invocation, these points may modify the server-side TSC.
6. After the **receive_request** points are called, control transfers to the server threads that may also read and write this server-side TSC.
7. The target operation invocation completes and control returns to the ORB.
8. The TSC from the thread on which the ORB invoked the target operation is copied back to the RSC, overwriting the slots in the RSC.
9. The send interception points have access to this RSC from which they may populate the reply service context list. After the invocation result is sent back to the client, the server-side RSC is logically destroyed.
10. The client receives the reply. The Interceptors may read the service contexts associated with the reply. They also have readonly access to the RSC was seen by the send interception points.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11. The invocation returns to the client. When the request completes, the client-side RSC is logically destroyed.

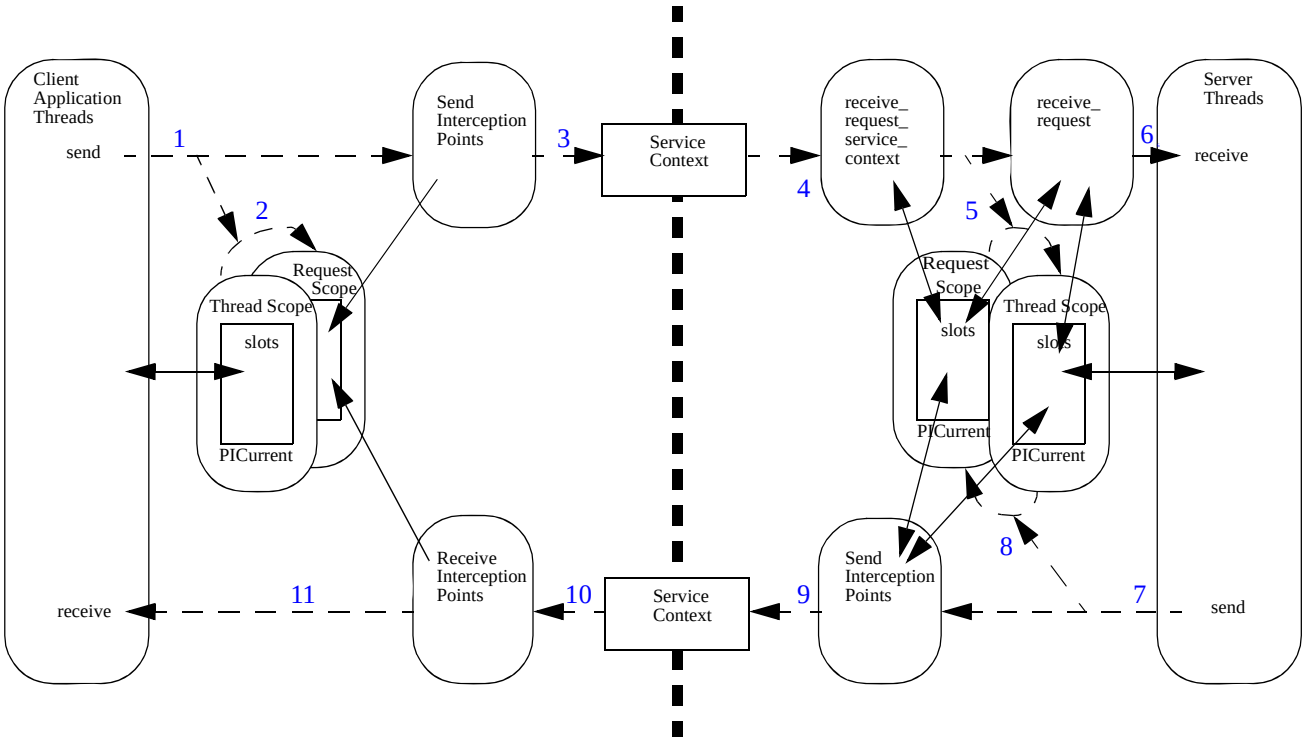


Figure 16.1 - Thread Scope vs Request Scope

Figure 16.1 Legend

Dotted Line	Flow of control (between the thread scopes and the request scopes, the dotted arrows indicate a logical copy).
Solid Line	Access; single arrow is readonly, double arrow is read/write.
Thick Dotted Line	Boundary between client and server.

16.5.4.6 Notes on PICurrent and Scopes

Since an Interceptor is running in a thread, it is running with a thread context and there is a **PICurrent** on that context. If the Interceptor calls **ORB::resolve_initial_references ("PICurrent")**, it gets the **PICurrent** within its thread scope. This **PICurrent** is different than the request scope **PICurrent** that the Interceptor obtains via calls to the **Client-** or **Server- RequestInfo** object. So if an Interceptor makes an operation call, it is the Interceptor's thread scope **PICurrent** that will be logically copied to the request scope of that operation, not the **PICurrent** from the original operation invocation.

Even if a client-side Interceptor happens to be running in the same thread from which the invocation was made (this is vendor dependent), the request scope **PICurrent** and the thread scope **PICurrent** are still different. The request scope **PICurrent** is a copy of the thread scope **PICurrent** at the point when the invocation began. So even if an Interceptor changed the data in its thread scope **PICurrent**, that does not change the request scope **PICurrent**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Interceptors shall assume that each client-side interception point logically runs in its own TSC thread, with no context relationship between it and any other thread. Each point's logical TSC thread is shared by all registered **ClientRequestInterceptors** executing in that point. While an ORB implementation may not actually behave in this manner, it is up to the ORB implementation to treat **PICurrent** as if it did.

Interceptors shall assume that all server-side interception points except **receive_request_service_contexts** run in the same thread as the target operation invocation, thereby sharing thread context information. **receive_request_service_contexts**, like all client-side interception points, logically runs in its own TSC thread, with no context relationship between it and any other thread. The **receive_request_service_contexts** interception point logical TSC thread is shared by all registered **ServerRequestInterceptors** executing in that point. While an ORB implementation may not actually behave in this manner, it is up to the ORB implementation to treat **PICurrent** as if it did.

16.6 IOR Interceptor

16.6.1 Overview

In some cases, a portable ORB service implementation may need to add information describing the server's or object's ORB service related capabilities to object references in order to enable the ORB service implementation in the client to function properly.

This is supported through the **IORInterceptor** and **IORInfo** interfaces.

The IOR Interceptor is used to establish tagged components in the profiles within an IOR.

16.6.2 An Abstract Model for Object Adapters

Using the **IORInterceptor** to support the object reference template imposes certain requirements on Object Adapters. While the **POA** is the only (current) standard object adapter, it is deemed inappropriate to impose the **POA** architecture on all possible proprietary object adapters. Consequently only the abstract properties that are required, and how these map to the particular case of the POA, are presented here.

Object Adapters have the following requirements:

- They have a unique name so that different instances of a particular object adapter may be identified.
- Object adapters typically have some kind of request processing state to indicate whether the adapter is currently accepting, rejecting, or performing some other kind of action on incoming requests. There is some representation of adapter instance state so that a server activation framework built on the object reference template can correctly process requests as the adapter instances states change.
- If an object adapter supports large numbers of adapter instances, reporting state changes that affect a number of adapter instances simultaneously could be expensive in the amount of data required. The POA has the concept of an adapter manager (the **POAManager**) that controls the state of a number of **POA** instances. They must have an abstract adapter manager that can be used for reporting relevant state changes.

16.6.2.1 Adapter Names

If an Object Adapter supports multiple adapter instances, there is a need for some kind of adapter name to distinguish the instances. For this purpose, an adapter name is defined as a sequence of strings. Several interpretations of an adapter name are possible:

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- If the Object Adapter supports only a single instance, a fixed name can be used.
- If the namespace for an Object Adapter is flat, sequences of length 1 can be used.
- If the namespace is hierarchical (e.g., the POA), a more complex name sequence can be used.

In the case of the POA, the adapter name shall be the sequence of names starting with the root POA that is required to reach the POA using the **find_POA** call. The name of the root POA is the sequence containing only the string "RootPOA."

16.6.2.2 Adapter States

Object adapters may be in one of several states that describe how the adapter behaves when a new request is dispatched to the adapter:

HOLDING	The request is held off temporarily in response to a transient resource limit or a application program request. An IMR could either choose to forward the request to the server and let the server hold it off, or else to hold off the request at the IMR until the state changes.
ACTIVE	The request is dispatched to the servant and processed. An IMR should forward the request to the server in this case.
DISCARDING	The request is discarded. This is indicated to the client with some kind of error. An IMR could either forward the request to the server, or else reject the request directly. The POA specification requires that a TRANSIENT/1 system exception be returned to the client in this case.
INACTIVE	The request is discarded. The adapter is in the process of shutting down, and will eventually end up in the **NON_EXISTENT** state. An IMR could reject the request directly, typically with an OBJ_ADAPTER/1 error.
NON_EXISTENT	The adapter has been destroyed. The IMR should attempt to reactivate the server and adapter as necessary to satisfy the request. The IMR should hold off the request until the adapter becomes active again.

In the case of the **POA**, **HOLDING**, **ACTIVE**, **DISCARDING**, and **INACTIVE** map to the same named states of the POAManager. **NON_EXISTENT** does not map directly to a particular **POAManager** state, but is used to indicate that a **POA** has been destroyed. A **POA** whose state is **INACTIVE** will transition to state **NON_EXISTENT** after the destruction process has completed.

- While non-POA adapters may have different detailed states than the POA, it should be possible to map other adapter's states onto a subset of the above states.

16.6.2.3 Adapter Managers

Some object adapters have a concept of a group of adapters that undergo state transitions together. In such cases it is useful to capture the grouping abstractly. We define the *adapter manager* to represent this grouping. The only standard attribute of the adapter manager is the adapter manager id, which is an opaque id. This ID serves to distinguish different adapter manager instances, and to associate an adapter manager instance with its adapter instances. The adapter manager id is only locally significant within the ORB instance that defines the adapter manager. The id is transient, and can be compared for equality within the defining ORB instance. All adapter instances that share the same adapter manager must have the same adapter manager id.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Use of an adapter manager allows state transitions for all adapters managed by the same adapter manager to be efficiently reported. The only assumption made about the semantics of an adapter manager is that a state change reported for an adapter manager is reflected in all adapter instances managed by the adapter manager.

In the case of the **POA**, the **POAManager** is an adapter manager.

16.6.2.4 Adapter State Changes

Some adapters may support mechanisms independent of the adapter manager for changing states. In such cases, a means needs to be provided for reporting the state changes.

In the case of the POA, a subtree of POAs may all transition to the **NON_EXISTENT** state as a result of the **POA::destroy** call.

16.6.3 Object Reference Template

16.6.3.1 Definition

The Object Reference Template is defined in IDL as an abstract valuetype.

An object reference template is associated with an object adapter. Typically the template is created when the object adapter is created, used within the adapter to create object references, and destroyed when the adapter is destroyed. Different adapters may support very different styles of object creation.

The object reference template is defined as follows:

```
module PortableInterceptor {
    typedef string ServerId ;
    typedef string ORBId ;
    typedef CORBA::StringSeq AdapterName ;
    typedef CORBA::OctetSeq ObjectId ;

    abstract valuetype ObjectReferenceFactory {
        boolean equals( in ObjectReferenceFactory other ) ;
        Object make_object( in string repositoryId, in ObjectId id ) ;
        IOP::TaggedProfileSeq make_profiles(
            in string repository_id,
            in ObjectId id ) ;
    };

    abstract valuetype ObjectReferenceTemplate :
        ObjectReferenceFactory {
        readonly attribute ServerId server_id ;
        readonly attribute ORBId orb_id ;
        readonly attribute AdapterName adapter_name;
    };

    typedef sequence<ObjectReferenceTemplate>
        ObjectReferenceTemplateSeq;
};
```


© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The **ObjectReferenceFactory** valuetype provides the capability to create new object references, while the **ObjectReferenceTemplate** valuetype extends the factory capability with the identity of the template. This division is convenient because the **current_factory** attribute in IORInfo (see IORInfo Interface on page 397) only requires the capability to create an object reference, while the **adapter_template** attribute (also in IORInfo Interface on page 397) also requires identity information.

Concrete definitions and implementations of **ObjectReferenceTemplate** and **ObjectReferenceFactory** are ORB implementation specific and are not defined as they are not expected to be exchanged between ORB implementations.

16.6.3.2 The ObjectReferenceFactory abstract valuetype

The **ObjectReferenceFactory** provides only the capability to create an object reference. Note that a factory is immutable: after it has been created, it cannot be modified.

Also, note that it is possible to create a concrete valuetype (unknown to the ORB implementation) that subclasses the **ObjectReferenceFactory** valuetype, and to use this factory in the IOR interceptor as **current_factory** (see current_factory on page 399). In such cases, the implementation must either be immutable after it is created, or the implementation must not change the behavior of **make_object**. Failure to observe this requirement may result in undefined behavior.

16.6.3.3 make_object

make_object creates an Object Reference from this factory using the given repository ID and object ID.

16.6.3.4 make_profiles

make_profiles returns the sequence of tagged profiles for the IOR that corresponds to the object reference that would be created by a call to **make_object** with the same arguments.

16.6.3.5 equals

equals satisfies the usual reflexive, symmetric, and transitive properties that equality normally respects. That is, for any **ObjectReferenceFactories X**, **Y**, and **Z**:

1. **X**.**equals**(**X**) = **TRUE**
2. **X**.**equals**(**Y**) = **Y**.**equals**(**X**)
3. if **X**.**equals**(**Y**) = **TRUE** and **Y**.**equals**(**Z**) = **TRUE**, then **X**.**equals**(**Z**) = **TRUE**

If **X** and **Y** are different object adapters, and **Xinfo** and **Yinfo** are the **IORInfo** objects passed to the **IORInterceptor**, then **Xinfo**.**adapter_template**().**equals**(**Yinfo**.**adapter_template**()) = **FALSE.**

An **equals** method on a user defined **ObjectReferenceFactory** must return **FALSE** when passed the value of an **IORInfo**.**adapter_template** attribute, unless the user defined **make_profiles** method returns the same **ProfileSeq** as the **adapter_template make_profiles** method when invoked with the same arguments, in which case the user defined **ObjectReferenceFactory equals** method may return **TRUE**.

16.6.3.6 The ObjectReferenceTemplate abstract valuetype

The **ObjectReferenceTemplate** extends the **ObjectReferenceFactory** with the identity of the object adapter. Note that the template, like the factory, is immutable: after it has been created, it cannot be modified.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.6.3.7 server_id

The value of the **server_id** attribute is the value that was passed into the ORB::init call (see Server ID on page 117) using the **-ORBServerId** argument when the ORB was created.

16.6.3.8 orb_id

The value of the **orb_id** attribute is the value that was passed into the **ORB::init** call.

In Java, this is accomplished using the **-ORBid** argument in the **`ORB.init`** call that created the ORB containing the object adapter that created this template. What happens if the same ORBid is used on multiple **`ORB::init`** calls in the same server is currently undefined.

16.6.3.9 adapter_name

The **adapter_name** attribute defines a name for the object adapter that services requests for the invoked object.

16.6.4 IORInterceptor Interface

```
local interface IORInterceptor : Interceptor {
    void establish_components (in IORInfo info);
};

local interface IORInterceptor_3_0 : IORInterceptor {
    void components_established( in IORInfo info ) ;

    void adapter_manager_state_changed( in AdapterManagerId id,
                                        in AdapterState state );

    void adapter_state_changed( in ObjectReferenceTemplateSeq
        templates, in AdapterState state ) ;
};
```

16.6.4.1 establish_components

A server side ORB calls the **establish_components** operation on all registered **IORInterceptor** instances when it is assembling the list of components that will be included in the profile or profiles of an object reference. This operation is not necessarily called for each individual object reference. In the case of the POA, these calls are made each time **POA::create_POA** is called. In other adapters, these calls would typically be made when the adapter is initialized. The adapter template is not available at this stage since information (the components) needed in the adapter template is being constructed.

An implementation of **establish_components** must not throw exceptions. If it does, the ORB shall ignore the exception and proceed to call the next IOR Interceptor's **establish_components** operation.

Parameter(s)

- **info**
  The IORInfo instance used by the ORB service to query applicable policies and add components to be included in the generated IORs.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.6.4.2 components_established

After all of the **establish_components** methods have been called, the **components_established** methods are called on all registered IORInterceptor_3_0 instances. The adapter template is available at this stage. The **current_factory** attribute may be get or set at this stage.

Any exception that occurs in **components_established** is returned to the caller of **components_established**. In the case of the POA, this causes the **create_POA** call to fail, and an **OBJ_ADAPTER** exception with a standard minor code of 6 is returned to the invoker of **create_POA**.

16.6.4.3 adapter_manager_state_changed

Any time the state of an adapter manager changes, the **adapter_manager_state_changed** method is invoked on all registered IORInterceptor_3_0 instances.

If a state change is reported through **adapter_manager_state_changed**, it is not reported through **adapter_state_changed**.

16.6.4.4 adapter_state_changed

Adapter state changes unrelated to adapter manager state changes are reported by invoking the adapter_state_changed method on all registered IORInterceptor_3_0 instances. The templates argument identifies the object adapters that have changed state by the template ID information. The sequence contains the adapter templates for all object adapters that have made the state transition being reported.

16.6.5 IORInfo Interface

The **IORInfo** interface provides the server-side ORB service with access to the applicable policies during IOR construction and the ability to add components. The ORB passes an instance of its implementation of this interface as a parameter to **IORInterceptor::establish_components**.

```
typedef string AdapterManagerId;

typedef short AdapterState ;
const AdapterState              HOLDING         = 0 ;
const AdapterState              ACTIVE          = 1 ;
const AdapterState              DISCARDING      = 2 ;
const AdapterState              INACTIVE        = 3 ;
const AdapterState              NON_EXISTENT    = 4 ;

    local interface IORInfo {
        CORBA::Policy get_effective_policy (in CORBA::PolicyType type);
        void add_ior_component
            (in IOP::TaggedComponent a_component);
        void add_ior_component_to_profile (
            in IOP::TaggedComponent a_component,
            in IOP::ProfileId profile_id);
                        readonly attribute AdapterManagerId manager_id;
                        readonly attribute AdapterState state;
                        readonly attribute ObjectReferenceTemplate adapter_template ;
                        attribute ObjectReferenceFactory current_factory ;
    };
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

All object adapter implementations provide some mechanism for creating object references. The construction of the object reference is influenced by all of the applicable server-side policies, which are used while assembling the tagged components required for the object reference. The IOR interceptors also influence the tagged components through the **IORInfo::add_component** and **IORInfo:add_component_to_profile** methods. After all of this construction has completed, the adapter conceptually has a template that can be used to create object references. We will refer to this template as the *adapter template*.

For example, in the POA, after **POA::create_POA** method has completed, there is a complete template in the POA that will be used to create individual object references when **create_reference** or any other method is called that needs to create an object reference.

16.6.5.1 get_effective_policy

An ORB service implementation may determine what server side policy of a particular type is in effect for an IOR being constructed by calling the **get_effective_policy** operation. When the IOR being constructed is for an object implemented using a POA, all Policy objects passed to the **PortableServer::POA::create_POA** call that created that POA are accessible via **get_effective_policy**.

If a policy for the given type is not known to the ORB, then this operation will raise INV_POLICY with a standard minor code of 3.

Parameter(s)

- **type**
  The CORBA::PolicyType specifying the type of policy to return.

Return Value
The effective CORBA::Policy object of the requested type. If the given policy type is known, but no policy of that type is in effect, then this operation will return a nil object reference.

16.6.5.2 add_ior_component

A portable ORB service implementation calls **add_ior_component** from its implementation of **establish_components** to add a tagged component to the set that will be included when constructing IORs. The components in this set will be included in all profiles.

Any number of components may exist with the same component ID.

Parameter(s)

- **a_component**
  The IOP::TaggedComponent to add.

16.6.5.3 add_ior_component_to_profile

A portable ORB service implementation calls **add_ior_component_to_profile** from its implementation of **establish_components** to add a tagged component to the set that will be included when constructing IORs. The components in this set will be included in the specified profile.

Any number of components may exist with the same component ID.

If the given profile ID does not define a known profile or it is impossible to add components to that profile, BAD_PARAM is raised with a standard minor code of 29.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Parameter(s)

- **a_component**
  The IOP::TaggedComponent to add.
- **profile_id**
  The IOP::ProfileId of the profile to which this component will be added.

16.6.5.4 manager_id

The **manager_id** attribute provides an opaque handle to the manager of the adapter. This is used for reporting state changes in adapters managed by the same adapter manager.

16.6.5.5 state

The **state** attribute returns the current state of the adapter. This must be one of HOLDING, ACTIVE, DISCARDING, INACTIVE, NON_EXISTENT.

16.6.5.6 adapter_template

The **adapter_template** attribute provides a means to obtain an object reference template whenever an ior interceptor is invoked. There is no standard way to directly create an object reference template. The value of **adapter_template** is the template created for the adapter policies and IOR interceptor calls to **add_component** and **add_component_to_profile**. The value of the **adapter_template** attribute is never changed for the lifetime of the object adapter.

16.6.5.7 current_factory

The **current_factory** attribute provides access to the factory that will be used by the adapter to create object references. **current_factory** initially has the same value as the **adapter_template** attribute, but this can be changed by setting **current_factory** to another factory. All object references created by the object adapter must be created by calling the **make_object** method on **current_factory**.

The value of the **current_factory** attribute that is used by the adapter can only be set during the call to the **components_established** method.

16.6.5.8 Method Validity

The following table defines the validity of each attribute or operation in **IORInfo** in the methods defined in the **IORInterceptor**.

	establish_components	**components_established**
get_effective_policy	yes	yes
add_component	yes	no
add_component_to_profile	yes	no
read manager_id	yes	yes
read state	yes	yes
read adapter_template	no	yes

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

read current_factory	no	yes
write current_factory	no	yes

If an illegal call is made to an attribute or operation in **IORInfo**, the BAD_INV_ORDER system exception is raised with a standard minor code value of 14.

16.7 Interceptor Policy Objects

An Interceptor's behavior may itself be modified by one or more Interceptor Policies. These **Policy** objects are created using a call to **ORB::create_policy** and are associated with an **Interceptor** during registration. (All Policy interfaces defined in this sub clause are local.) The ORB can be accessed via the implicit **get_orb** operation of **ORBInitInfo**.

16.7.1 ProcessingMode Policy

Request interceptor performance may be improved by applying a ProcessingMode policy to limit the conditions under which the interceptor shall be invoked.

The following values can be supplied.

- **LOCAL_AND_REMOTE** - Request interceptors with this policy are invoked whether the method is executed locally or remotely. This is the default behavior if no **ProcessingMode** Policy is associated with a request Interceptor.
- **REMOTE_ONLY** - Request interceptors with this policy are not invoked when the method is executed using the optimized collocated path.
- **LOCAL_ONLY** - Request interceptors with this policy are only invoked when the method is executed using the optimized collocated path.

```
module PortableInterceptor {

    typedef short ProcessingMode;
    const ProcessingMode LOCAL_AND_REMOTE = 0;
    const ProcessingMode REMOTE_ONLY      = 1;
    const ProcessingMode LOCAL_ONLY       = 2;
    // ProcessingMode Policy (default = LOCAL_AND_REMOTE)

    const CORBA::PolicyType
        PROCESSING_MODE_POLICY_TYPE = 63;

    local interface ProcessingModePolicy : CORBA::Policy {
        readonly attribute ProcessingMode processing_mode;
    };
};
```


© ISO/IEC 2008 - All rights reserved
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.8 PolicyFactory

16.8.1 PolicyFactory Interface

A portable ORB service implementation registers an instance of the **PolicyFactory** interface during ORB initialization (see register_policy_factory on page 406) in order to enable its policy types to be constructed using **CORBA::ORB::create_policy**. The POA is required to preserve any policy that is registered with **ORBInitInfo** in this manner.

```
module PortableInterceptor
{
    local interface PolicyFactory {
        CORBA::Policy create_policy (
            in CORBA::PolicyType type,
            in any value)
            raises (CORBA::PolicyError);
    };
};
```

16.8.1.1 create_policy

The ORB calls **create_policy** on a registered **PolicyFactory** instance when **CORBA::ORB::create_policy** is called for the **PolicyType** under which the **PolicyFactory** has been registered. The **create_policy** operation then returns an instance of the appropriate interface derived from **CORBA::Policy** whose value corresponds to the specified **any**. If it cannot, it shall raise an exception as described for **CORBA::ORB::create_policy**.

Parameter(s)

- **type**
  A CORBA::PolicyType specifying the type of policy being created.
- **value**
  An any containing data with which to construct the CORBA::Policy.

Return Value

A CORBA::Policy object of the specified type and value.

16.9 Registering Interceptors

Interceptors are intended to be a means by which ORB services gain access to ORB processing, effectively becoming part of the ORB. Since Interceptors are part of the ORB, when **ORB_init** returns an ORB, the Interceptors shall have been registered. Interceptors cannot be registered on an ORB after it has been returned by a call to **ORB_init**.

16.9.1 ORBInitializer Interface

An Interceptor is registered by registering an associated **ORBInitializer** object that implements the **ORBInitializer** interface. When an ORB is initializing, it shall call each registered **ORBInitializer**, passing it an **ORBInitInfo** object, which is used to register its Interceptor. Any exceptional return from the invocation of any operation of the

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

ORBInitializer interface other than those resulting from the failure to instantiate a portable interceptor object shall result in the abandonment of the ORB initialization and destruction of the ORB. Any **ORBInitializer** implementation that needs the ORB to ignore any thrown exceptions can simply catch and discard them itself.

```
module PortableInterceptor {
    local interface ORBInitializer {
        void pre_init (in ORBInitInfo info);
        void post_init (in ORBInitInfo info);
    };
};
```

16.9.1.1 pre_init

This operation is called during ORB initialization. If it is expected that initial services registered by an interceptor will be used by other interceptors, then those initial services shall be registered at this point via calls to **ORBInitInfo::register_initial_reference**.

Parameter(s)

- **info**
  See below. This object provides initialization attributes and operations by which Interceptors can be registered.

16.9.1.2 post_init

This operation is called during ORB initialization. If a service must resolve initial references as part of its initialization, it can assume that all initial references will be available at this point.

Calling the **post_init** operations is not the final task of ORB initialization. The final task, following the **post_init** calls, is attaching the lists of registered interceptors to the ORB. Therefore, the ORB does not contain the interceptors during calls to **post_init**. If an ORB-mediated call is made from within **post_init**, no request interceptors will be invoked on that call. Likewise, if an operation is performed that causes an IOR to be created, no IOR interceptors will be invoked.

Parameter(s)

- **info**
  See below. This object provides initialization attributes and operations by which Interceptors can be registered.

During a call to **post_init**, invoking the **ORBInitInfo** methods: **add_client_request_interceptor**, **add_server_request_interceptor**, **allocate_slot_id**, or **add_ior_interceptor** will raise the BAD_INV_ORDER system exception with standard minor code 26.

16.9.2 ORBInitInfo Interface

```
module PortableInterceptor {
    local interface ORBInitInfo {
        typedef string ObjectId;
        exception DuplicateName {
            string name;
        };
        exception InvalidName {};

        readonly attribute CORBA::StringSeq arguments;
        readonly attribute string orb_id;
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        readonly attribute IOP::CodecFactory codec_factory;

        void register_initial_reference (in ObjectId id, in Object obj)
            raises (InvalidName);
        Object resolve_initial_references (
            in ObjectId id) raises (InvalidName);
        void add_client_request_interceptor (
            in ClientRequestInterceptor interceptor)
            raises (DuplicateName);
        void add_server_request_interceptor (
            in ServerRequestInterceptor interceptor)
            raises (DuplicateName);
        void add_ior_interceptor (in IORInterceptor interceptor)
            raises (DuplicateName);
        SlotId allocate_slot_id ();
        void register_policy_factory (
            in CORBA::PolicyType type,
            in PolicyFactory policy_factory);
    };

    local interface ORBInitInfo_3_1 : ORBInitInfo {
        void add_client_request_interceptor_with_policy(
            in ClientRequestInterceptor interceptor,
            in CORBA::PolicyList policies)
            raises (DuplicateName, CORBA::PolicyError);
        void add_server_request_interceptor_with_policy(
            in ServerRequestInterceptor interceptor,
            in CORBA::PolicyList policies)
            raises (DuplicateName, CORBA::PolicyError);
        void add_ior_interceptor_with_policy(
            in IORInterceptor interceptor,
            in CORBA::PolicyList policies)
            raises (DuplicateName, CORBA::PolicyError);
    };
};
```

16.9.2.1 DuplicateName Exception

Only one Interceptor of a given name can be registered with the ORB for each Interceptor type. If an attempt is made to register a second Interceptor with the same name, DuplicateName is raised.

An Interceptor may be anonymous; that is, have an empty string as the name attribute. Any number of anonymous Interceptors may be registered with the ORB so, if the Interceptor being registered is anonymous, the registration operation will not raise DuplicateName.

16.9.2.2 InvalidName Exception

This exception is raised by **register_initial_reference** and **resolve_initial_references**.

register_initial_reference raises InvalidName if:

- this operation is called with an empty string id; or

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- this operation is called with an id that is already registered, including the default names defined by OMG.

resolve_initial_references raises InvalidName if the name to be resolved is invalid.

16.9.2.3 arguments

This attribute returns the original *argv* parameters as they were passed to **ORB_init**.

16.9.2.4 orb_id

This attribute is the ID of the ORB being initialized.

16.9.2.5 codec_factory

This attribute is the **IOP::CodecFactory**. The **CodecFactory** is normally obtained via a call to **ORB::resolve_initial_references ("CodecFactory")**, but since the ORB is not yet available and Interceptors, particularly when processing service contexts, will require a **Codec**, a means of obtaining a **Codec** is necessary during ORB initialization.

16.9.2.6 register_initial_reference

This operation is identical to **ORB::register_initial_reference** described there. This same functionality exists here because the ORB, not yet fully initialized, is not yet available but initial references may need to be registered as part of Interceptor registration. The only difference is that the version of this operation on the ORB uses PIDL (**CORBA::ORB::ObjectId** and **CORBA::ORB::InvalidName**) whereas the version in this interface uses IDL defined in this interface; the semantics are identical.

16.9.2.7 resolve_initial_references

See Registering Interceptors on page 401. This operation is only valid during **post_init**. It is identical to **ORB::resolve_initial_references**. This same functionality exists here because the ORB, not yet fully initialized, is not yet available but initial references may be required from the ORB as part of Interceptor registration. The only difference is that the version of this operation on the ORB uses PIDL (**CORBA::ORB::ObjectId** and **CORBA::ORB::InvalidName**) whereas the version in this interface uses IDL defined in this interface; the semantics are identical.

16.9.2.8 add_client_request_interceptor

This operation is used to add a client-side request Interceptor to the list of client-side request Interceptors. If a client-side request Interceptor has already been registered with this Interceptor's name, DuplicateName is raised.

Parameter(s)

- **interceptor**
  The ClientRequestInterceptor to be added.

16.9.2.9 add_server_request_interceptor

This operation is used to add a server-side request Interceptor to the list of server-side request Interceptors. If a server-side request Interceptor has already been registered with this Interceptor's name, DuplicateName is raised.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Parameter(s)

- **interceptor**
  The ServerRequestInterceptor to be added.

16.9.2.10 add_ior_interceptor

This operation is used to add an IOR Interceptor to the list of IOR Interceptors. If an IOR Interceptor has already been registered with this Interceptor's name, DuplicateName is raise

Parameter(s)

- **interceptor**
  The IORInterceptor to be added.

16.9.2.11 add_client_request_interceptor_with_policy

This form of registration allows interceptor behavior to be modified by one or more Policies. The policy objects are effectively copied before the operation returns, so the caller is free to destroy them while the Interceptor is in use.

CORBA::PolicyError is raised if one or more of the policies is invalid. If a server-side request Interceptor has already been registered with this Interceptor's name, **DuplicateName** is raised.

Parameter(s)

- **interceptor**
  The client request interceptor to be added.
- **policies**
  A sequence of interceptor policies to be used to control the behavior of the interceptor being registered.

16.9.2.12 add_server_request_interceptor_with_policy

This form of registration allows interceptor behavior to be modified by one or more Policies. The policy objects are effectively copied before the operation returns, so the caller is free to destroy them while the Interceptor is in use. **CORBA::PolicyError** is raised if one or more of the policies is invalid. If a server-side request Interceptor has already been registered with this Interceptor's name, **DuplicateName** is raised.

Parameter(s)

- **interceptor**
  The server request interceptor to be added.
- **policies**
  A sequence of interceptor policies to be used to control the behavior of the interceptor being registered.

16.9.2.13 add_ior_interceptor_with_policy

This form of registration allows interceptor behavior to be modified by one or more Policies. The policy objects are effectively copied before the operation returns, so the caller is free to destroy them while the Interceptor is in use. **CORBA::PolicyError** is raised if one or more of the policies is invalid. If a server-side request Interceptor has already been registered with this Interceptor's name, **DuplicateName** is raised.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Parameter(s)

- **interceptor**
  The ior request interceptor to be added.
- **policies**
  A sequence of interceptor policies to be used to control the behavior of the interceptor being registered

16.9.2.14 allocate_slot_id

A service calls **allocate_slot_id** to allocate a slot on **PortableInterceptor::Current**.

Note that while slot ids can be allocated within an ORB initializer, the slots themselves cannot be initialized. Calling **set_slot** or **get_slot** on the **PICurrent** (see Portable Interceptor Current on page 386) within an ORB initializer shall raise a BAD_INV_ORDER with a minor code of 14.

Return Value

The index to the slot that has been allocated.

16.9.2.15 register_policy_factory

Register a **PolicyFactory** for the given **PolicyType**.

If a **PolicyFactory** already exists for the given **PolicyType**, BAD_INV_ORDER is raised with a standard minor code of 16.

Parameter(s)

- **type**
  The CORBA::PolicyType that the given PolicyFactory serves.
- **policy_factory**
  The factory for the given CORBA::PolicyType.

16.9.3 register_orb_initializer Operation

To register an **ORBInitializer**, a new operation is provided: **register_orb_initializer**. This operation, like **ORB_init**, is PIDL and is not part of any interface. It resides in the **PortableInterceptor** module.

```
void register_orb_initializer (in ORBInitializer init);
```

Each service that implements Interceptors will provide an instance of **ORBInitializer**. To use a service, an application would first call **register_orb_initializer**, passing in the service's **ORBInitializer**. After this is complete, the application would make an instantiating **ORB_init** call. (An instantiating **ORB_init** call is one that produces a new ORB. In other words, one that is not passed the ID of an existing ORB.) This instantiating **ORB_init** call calls each registered **ORBInitializer**. The returned ORB will contain any Interceptors that the given service requires.

register_orb_initializer is a global operation. An **ORBInitializer** registered at a given point in time will be called by all instantiating **ORB_init** calls that occur after that point in time. No ORB instantiated before that point in time will be affected by that **ORBInitializer**. Moreover, if **register_orb_initializer** is called from within an initializer, the initializer registered by that call will not be called for the ORB currently being initialized. That initializer will only be invoked on an ORB instantiated at a later time.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.9.3.1 Mappings of register_orb_initializer

C++

The **register_orb_initializer** method is defined in the **PortableInterceptor** name space as:

```
namespace PortableInterceptor {
   static void register_orb_initializer (
         PortableInterceptor::ORBInitializer_ptr init);
};
```

Java

The **register_orb_initializer** operation, since it is global, would break applet security with respect to the ORB. So, in Java, instead of registering **ORBInitializers** via **register_orb_initializer**, **ORBInitializers** are registered via Java ORB properties.

New Property Set

The new property names are of the form:

org.omg.PortableInterceptor.ORBInitializerClass.<Service>

where <**Service**> is the string name of a class, which implements

org.omg.PortableInterceptor.ORBInitializer.

To avoid name collisions, the reverse DNS name convention should be used. For example, if company X has three initializers, it could define the following properties:

```
org.omg.PortableInterceptor.ORBInitializerClass.com.x.Init1
org.omg.PortableInterceptor.ORBInitializerClass.com.x.Init2
org.omg.PortableInterceptor.ORBInitializerClass.com.x.Init3
```

During **ORB.init**, these ORB properties that begin with **org.omg.PortableInterceptor.ORBInitializerClass** shall be collected, the <**Service**> portion of each property shall be extracted, an object shall be instantiated with the <**Service**> string as its class name, and the **pre_init** and **post_init** methods shall be called on that object. If the attempt to instantiate an interceptor object fails the ORB shall ignore the failure and continue execution. For any other exceptions returned by pre_init or post_init, the ORB shall discontinue initialization and destroy itself, and the original exception returned by the **ORBInitializer** shall be returned by **ORB_init**.

Example

A client-side logging service written by company X, for example, may have the following **ORBInitializer** implementation:

```
package com.x.logging;

import org.omg.PortableInterceptor.Interceptor;
import org.omg.PortableInterceptor.ORBInitializer;
import org.omg.PortableInterceptor.ORBInitInfo;
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
public class LoggingService implements ORBInitializer
{
   void pre_init (ORBInitInfo info)
   {
         // Instantiate the Logging Service's Interceptor.
         Interceptor interceptor = new LoggingInterceptor ();

               // Register the Logging Service's Interceptor.
         info.add_client_request_interceptor (interceptor);
   }

            void post_init (ORBInitInfo info)
   {
         // This service does not need two init points.
   }
}
```

To run a program called **MyApp** using this logging service, the user could type:

```
java
-Dorg.omg.PortableInterceptor.ORBInitializerClass.com.x.
Logging.LoggingService MyApp
```

Ada

For the Ada mapping, a new child library procedure is defined to register **ORBInitializers**:

```
procedure PortableInterceptor.ORBinitializer.Register
   (Init: in PortableInterceptor.ORBinitializer.Local_Ref);
```

16.9.4 Notes about Registering Interceptors

Request Interceptors are registered on a per-ORB basis.

To achieve virtual per-object Interceptors, query the policies on the target from within the interception points to determine whether they should do any work.

To achieve virtual per-POA Interceptors, instantiate each POA with a different ORB.

While Interceptors may be ordered administratively, there is no concept of order with respect to the registration of Interceptors. Request Interceptors are concerned with service contexts. Service contexts have no order, so there is no purpose for request Interceptors to have an order. IOR Interceptors are concerned with tagged components. Tagged components also have no order, so there is no purpose for IOR Interceptors to have an order.

Registration code should avoid using the ORB; that is, calling **ORB_init** with the provided **orb_id**. Since registration occurs during ORB initialization, results of invocations on this ORB while it is in this state are undefined.

The **ORBInitInfo** object is only valid during **ORB_init**. If a service keeps a reference to its **ORBInitInfo** object and tries to use it after **ORB_init** returns, the object no longer exists and an OBJECT_NOT_EXIST exception shall be raised.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.10 Dynamic Initial References

There are a set number of objects that a call to **ORB::resolve_initial_references** is able to return. However, vendors and applications may wish to add additional initial references. The lifecycle of these additional references coincides with the lifecycle of the ORB.

16.10.1 register_initial_reference

An operation is available in the ORB interface:

```
void register_initial_reference (in ObjectId id, in Object obj)
    raises (InvalidName);
```

If this operation is called with an id, “Y”, and an object, YY, then a subsequent call to **ORB::resolve_initial_references (“Y”)** will return object YY.

This operation can be used to replace the object reference corresponding to any of the OMG specified Ids. For example:

```
register_initial_reference ("NameService", Z)
```

will cause Z to be substituted as the object reference that will be used to get to the Name Service instead of the ORB vendor supplied built in Name Service. This facility should be used with care since subsitution of certain OMG specified ids is unlikely to work at all.

Implementations are allowed to restrict substitutability of references corresponding to the following **ObjectIds**:

RootPOA, POACurrent, DynAnyFactory, ORBPolicyManager, PolicyCurrent, CodecFactory, and PICurrent

When substitutability is restricted it shall be clearly documented. InvalidName exception is raised when any of these restricted **ObjectIds** are passed in as a parameter to **resolve_initial_reference**.

InvalidName is raised if this operation is called with an empty string id.

- this operation is called with an empty string id; or
- this operation is called with an id that is already registered, including the default names defined by OMG.

If the **Object** parameter is null, BAD_PARAM will be raised with a standard minor code of 27.

Parameter(s)

- **id**
  The ID by which the initial reference will be known.
- **obj**
  The initial reference itself.

See also register_initial_reference on page 404.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.11 Module Dynamic

In order to keep the portable Interceptor IDL from becoming PIDL, we provide IDL types that correspond to PIDL types for that subset of PIDL that the portable Interceptors use. We have chosen to place these new types in a module called **Dynamic** since it is the dynamic interface sub clauses that define the PIDL that the portable Interceptors use.

16.11.1 NVList PIDL Represented by ParameterList IDL

```
struct Parameter {
    any argument;
    CORBA::ParameterMode mode;
                              };
typedef sequence<Parameter> ParameterList;
```

16.11.2 ContextList PIDL Represented by ContextList IDL

```
typedef CORBA::StringSeq ContextList;
```

16.11.3 ExceptionList PIDL Represented by ExceptionList IDL

```
typedef sequence<CORBA::TypeCode> ExceptionList;
```

16.11.4 Context PIDL Represented by RequestContext IDL

Context objects are encoded as **sequence<string>**. The strings occur in pairs. The first string in each pair is the context property name and the second string in each pair is the associated value.

```
typedef CORBA::StringSeq RequestContext;
```

16.12 Consolidated IDL

16.12.1 Dynamic

```
// IDL
// File: Dynamic.idl
#ifndef _DYNAMIC_IDL_
#define _DYNAMIC_IDL_

import ::CORBA;
module Dynamic {
    typeprefix Dynamic “omg.org”;

    struct Parameter {
        any argument;
        CORBA::ParameterMode mode;
    };
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    typedef sequence<Parameter> ParameterList;

    typedef CORBA::StringSeq ContextList;

    typedef sequence<CORBA::TypeCode> ExceptionList;

    typedef CORBA::StringSeq RequestContext;
};
#endif _DYNAMIC_IDL_
```

16.12.2 Portions of IOP Relevant to Portable Interceptor

```
import ::CORBA;

module IOP{
    typeprefix IOP "omg.org";
    typedef sequence<IOP::TaggedComponent> TaggedComponentSeq;

    local interface Codec {
        exception InvalidTypeForEncoding {};
        exception FormatMismatch {};
        exception TypeMismatch {};

        CORBA::OctetSeq encode (in any data)
            raises (InvalidTypeForEncoding);
        any decode (in CORBA::OctetSeq data)
            raises (FormatMismatch);
        CORBA::OctetSeq encode_value (in any data)
            raises (InvalidTypeForEncoding);
        any decode_value (
            in CORBA::OctetSeq data,
            in CORBA::TypeCode tc)
            raises (FormatMismatch, TypeMismatch);
    };

    typedef short EncodingFormat;
    const EncodingFormat ENCODING_CDR_ENCAPS = 0;

    struct Encoding {
        EncodingFormat format;
        octet major_version;
        octet minor_version;
    };

    local interface CodecFactory {
        exception UnknownEncoding {};

        Codec create_codec (in Encoding enc) raises (UnknownEncoding);
    };
};
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.12.3 PortableInterceptor

```
// IDL
// File: PortableInterceptor.idl
#ifndef _PORTABLE_INTERCEPTOR_IDL_
#define _PORTABLE_INTERCEPTOR_IDL_

import ::CORBA;
import ::IOP;
import ::Messaging;
import ::Dynamic;

module PortableInterceptor {
    typeprefix PortableInterceptor “omg.org”;
    local interface Interceptor {
         readonly attribute string name;
        void destroy();
    };

    exception ForwardRequest {
        Object forward;
    };

    typedef short ReplyStatus;

    // Valid reply_status values:
        const ReplyStatus SUCCESSFUL = 0;
        const ReplyStatus SYSTEM_EXCEPTION = 1;
        const ReplyStatus USER_EXCEPTION = 2;
        const ReplyStatus LOCATION_FORWARD = 3;
        const ReplyStatus TRANSPORT_RETRY = 4;
        const ReplyStatus UNKNOWN = 5;

    typedef unsigned long SlotId;

    exception InvalidSlot {};

    typedef short ProcessingMode;
        const ProcessingMode LOCAL_AND_REMOTE = 0;
        const ProcessingMode REMOTE_ONLY      = 1;
        const ProcessingMode LOCAL_ONLY       = 2;
    // ProcessingMode Policy (default = LOCAL_AND_REMOTE)

    const CORBA::PolicyType
        PROCESSING_MODE_POLICY_TYPE = 63;

    local interface ProcessingModePolicy : CORBA::Policy {
        readonly attribute ProcessingMode processing_mode;
    };
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
local interface Current : CORBA::Current {
    any get_slot (in SlotId id) raises (InvalidSlot);
    void set_slot (in SlotId id, in any data) raises (InvalidSlot);
};

local interface RequestInfo {
    readonly attribute unsigned long request_id;
    readonly attribute string operation;
    readonly attribute Dynamic::ParameterList arguments;
    readonly attribute Dynamic::ExceptionList exceptions;
    readonly attribute Dynamic::ContextList contexts;
    readonly attribute Dynamic::RequestContext operation_context;
    readonly attribute any result;
    readonly attribute boolean response_expected;
    readonly attribute Messaging::SyncScope sync_scope;
    readonly attribute ReplyStatus reply_status;
    readonly attribute Object forward_reference;
    any get_slot (in SlotId id) raises (InvalidSlot);
    IOP::ServiceContext get_request_service_context (
        in IOP::ServiceId id);
    IOP::ServiceContext get_reply_service_context (
        in IOP::ServiceId id);
};

local interface ClientRequestInfo : RequestInfo {
    readonly attribute Object target;
    readonly attribute Object effective_target;
    readonly attribute IOP::TaggedProfile effective_profile;
    readonly attribute any received_exception;
    readonly attribute CORBA::RepositoryId received_exception_id;
    IOP::TaggedComponent get_effective_component (
        in IOP::ComponentId id);
    IOP::TaggedComponentSeq get_effective_components (
        in IOP::ComponentId id);
    CORBA::Policy get_request_policy (in CORBA::PolicyType type);
    void add_request_service_context (
        in IOP::ServiceContext service_context,
        in boolean replace);
};

typedef string ServerId ;
typedef string ORBId ;
typedef CORBA::StringSeq AdapterName ;
typedef CORBA::OctetSeq ObjectId;

local interface ServerRequestInfo : RequestInfo {
    readonly attribute any sending_exception;
                        readonly attribute ServerId server_id ;
                        readonly attribute ORBId orb_id ;
                        readonly attribute AdapterName adapter_name ;
    readonly attribute ObjectId object_id;
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        readonly attribute CORBA::OctetSeq adapter_id;
        readonly attribute CORBA::RepositoryId
            target_most_derived_interface;
        CORBA::Policy get_server_policy (in CORBA::PolicyType type);
        void set_slot (in SlotId id, in any data) raises (InvalidSlot);
        boolean target_is_a (in CORBA::RepositoryId id);
        void add_reply_service_context (
            in IOP::ServiceContext service_context,
            in boolean replace);
    };

    local interface ClientRequestInterceptor : Interceptor {
        void send_request  (in ClientRequestInfo ri)
            raises (ForwardRequest);
        void send_poll (in ClientRequestInfo ri);
        void receive_reply (in ClientRequestInfo ri);
        void receive_exception (in ClientRequestInfo ri)
            raises (ForwardRequest);
        void receive_other (in ClientRequestInfo ri)
            raises (ForwardRequest);
    };

    local interface ServerRequestInterceptor : Interceptor {
        void receive_request_service_contexts (in ServerRequestInfo ri)
            raises (ForwardRequest);
        void receive_request (in ServerRequestInfo ri)
             raises (ForwardRequest);
        void send_reply (in ServerRequestInfo ri);
        void send_exception (in ServerRequestInfo ri)
            raises (ForwardRequest);
        void send_other (in ServerRequestInfo ri)
            raises (ForwardRequest);
    };

    abstract valuetype ObjectReferenceFactory {
        boolean equals( in ObjectReferenceFactory other ) ;
        Object make_object( in string repositoryId, in ObjectId id ) ;
        IOP::TaggedProfileSeq make_profiles(
            in string repository_id,
            in ObjectId id ) ;
    };

    abstract valuetype ObjectReferenceTemplate :
        ObjectReferenceFactory {
        readonly attribute ServerId server_id ;
        readonly attribute ORBId orb_id ;
        readonly attribute AdapterName adapter_name ;
    } ;

    typedef sequence<ObjectReferenceTemplate>
        ObjectReferenceTemplateSeq;
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
typedef string AdapterManagerId;

typedef short AdapterState ;

const AdapterState      HOLDING         = 0 ;
const AdapterState      ACTIVE          = 1 ;
const AdapterState      DISCARDING      = 2 ;
const AdapterState      INACTIVE        = 3 ;
const AdapterState      NON_EXISTENT    = 4 ;

local interface IORInfo {
    CORBA::Policy get_effective_policy (in CORBA::PolicyType type);
    void add_ior_component (
        in IOP::TaggedComponent a_component);
    void add_ior_component_to_profile (
        in IOP::TaggedComponent a_component,
        in IOP::ProfileId profile_id);
};

local interface IORInterceptor : Interceptor {
    void establish_components (in IORInfo info);
};

local interface IORInterceptor_3_0 : IORInterceptor {
    void components_established( in IORInfo info ) ;
    void adapter_manager_state_changed(
        in AdapterManagerId id, in AdapterState state ) ;
    void adapter_state_changed(
        in ObjectReferenceTemplateSeq templates,
        in AdapterState state ) ;
};

local interface PolicyFactory {
    CORBA::Policy create_policy (
        in CORBA::PolicyType type,
        in any value)
        raises (CORBA::PolicyError);
};

local interface ORBInitInfo {
    typedef string ObjectId;
    exception DuplicateName {
        string name;
    };
    exception InvalidName {};

    readonly attribute CORBA::StringSeq arguments;
    readonly attribute string orb_id;
    readonly attribute IOP::CodecFactory codec_factory;

    void register_initial_reference (in ObjectId id, in Object obj)
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            raises (InvalidName);
        Object resolve_initial_references (
            in ObjectId id) raises (InvalidName);
        void add_client_request_interceptor (
            in ClientRequestInterceptor interceptor)
            raises (DuplicateName);
        void add_server_request_interceptor (
            in ServerRequestInterceptor interceptor)
            raises (DuplicateName);
        void add_ior_interceptor (in IORInterceptor interceptor)
            raises (DuplicateName);
        SlotId allocate_slot_id ();
        void register_policy_factory (
            in CORBA::PolicyType type,
            in PolicyFactory policy_factory);
    };

    local interface ORBInitInfo_3_1 : ORBInitInfo {
        void add_client_request_interceptor_with_policy(
            in ClientRequestInterceptor interceptor,
            in CORBA::PolicyList policies)
            raises (DuplicateName, CORBA::PolicyError);
        void add_server_request_interceptor_with_policy(
            in ServerRequestInterceptor interceptor,
            in CORBA::PolicyList policies)
            raises (DuplicateName, CORBA::PolicyError);
        void add_ior_interceptor_with_policy(
            in IORInterceptor interceptor,
            in CORBA::PolicyList policies)
            raises (DuplicateName, CORBA::PolicyError);
    };

    local interface ORBInitializer {
        void pre_init (in ORBInitInfo info);
        void post_init (in ORBInitInfo info);
    };
};
#endif _PORTABLE_INTERCEPTOR_IDL_
```


© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17 CORBA Messaging

This clause covers three general topics: Quality of Service, Asynchronous Method Invocations (including Time-Independent or "Persistent" Requests), and the specification of interoperable Routing interfaces to support the transport of requests asynchronously from the handling of their replies.

Section I - Quality of Service

Messaging requires clients and servers to have the ability to set the required and supported qualities of service with respect to requests. This specification provides generalized APIs through which such qualities are set in clients and servers. In addition, the set of Messaging-related qualities and the rules for reconciling and using these qualities are defined. Finally, the Messaging-specific IOR Profile Component and Service Context are defined for propagation of QoS information.

17.1 Section I - Introduction

This sub clause describes a standard Quality of Service (QoS) framework within which CORBA Services specifications should define their service-specific qualities. In this framework, all QoS settings are interfaces derived from **CORBA::Policy**.

The details of the Policy Management Framework are to be found in the *ORB Interface* clause.

17.2 Messaging Quality of Service

The Messaging module contains the IDL that the programmer uses to define Qualities of Service specific to CORBA messaging.

NOTE: Except where defaults are noted, this specification does not state required default values for the following Qualities of Service. Application code must explicitly set its ORB-level Quality of Service to ensure portability across ORB products.

```
module Messaging {

    typedef short RebindMode;
    const RebindMode TRANSPARENT =              0;
    const RebindMode NO_REBIND =                1;
    const RebindMode NO_RECONNECT =             2;

    typedef short SyncScope;
    const SyncScope SYNC_NONE =                 0;
    const SyncScope SYNC_WITH_TRANSPORT =       1;
    const SyncScope SYNC_WITH_SERVER =          2;
    const SyncScope SYNC_WITH_TARGET =          3;

    typedef short RoutingType;
    const RoutingType ROUTE_NONE =              0;
    const RoutingType ROUTE_FORWARD =           1;
    const RoutingType ROUTE_STORE_AND_FORWARD =2;
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
typedef short Priority;

typedef unsigned short Ordering;
const Ordering ORDER_ANY =              0x01;
const Ordering ORDER_TEMPORAL =         0x02;
const Ordering ORDER_PRIORITY =         0x04;
const Ordering ORDER_DEADLINE =         0x08;

// Rebind Policy (default = TRANSPARENT)
const CORBA::PolicyType REBIND_POLICY_TYPE = 23;
local interface RebindPolicy : CORBA::Policy {
    readonly attribute RebindMode     rebind_mode;
};

// Synchronization Policy (default = SYNC_WITH_TRANSPORT)
const CORBA::PolicyType SYNC_SCOPE_POLICY_TYPE = 24;
    local interface SyncScopePolicy : CORBA::Policy {
    readonly attribute SyncScope      synchronization;
};

// Priority Policies
const CORBA::PolicyType REQUEST_PRIORITY_POLICY_TYPE = 25;
struct PriorityRange {
    Priority min;
    Priority max;
};

local interface RequestPriorityPolicy : CORBA::Policy {
    readonly attribute PriorityRange   priority_range;
};

const CORBA::PolicyType REPLY_PRIORITY_POLICY_TYPE = 26;
local interface ReplyPriorityPolicy : CORBA::Policy {
    readonly attribute PriorityRange   priority_range;
};

// Timeout Policies
const CORBA::PolicyType
                        REQUEST_START_TIME_POLICY_TYPE = 27;
local interface RequestStartTimePolicy : CORBA::Policy {
    readonly attribute TimeBase::UtcT start_time;
};
const CORBA::PolicyType REQUEST_END_TIME_POLICY_TYPE = 28;
local interface RequestEndTimePolicy : CORBA::Policy {
    readonly attribute TimeBase::UtcT end_time;
};

const CORBA::PolicyType REPLY_START_TIME_POLICY_TYPE = 29;
local interface ReplyStartTimePolicy : CORBA::Policy {
    readonly attribute TimeBase::UtcT start_time;
};
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    const CORBA::PolicyType REPLY_END_TIME_POLICY_TYPE = 30;
    local interface ReplyEndTimePolicy : CORBA::Policy {
        readonly attribute TimeBase::UtcT end_time;
    };

    const CORBA::PolicyType
                RELATIVE_REQ_TIMEOUT_POLICY_TYPE = 31;
    local interface RelativeRequestTimeoutPolicy : CORBA::Policy {
        readonly attribute TimeBase::TimeT relative_expiry;
    };

    const CORBA::PolicyType
                RELATIVE_RT_TIMEOUT_POLICY_TYPE = 32;
    local interface RelativeRoundtripTimeoutPolicy : CORBA::Policy {
        readonly attribute TimeBase::TimeT relative_expiry;
    };

    const CORBA::PolicyType ROUTING_POLICY_TYPE = 33;
    struct RoutingTypeRange {
        RoutingType min;
        RoutingType max;
    };
    local interface RoutingPolicy : CORBA::Policy {
        readonly attribute RoutingTypeRange routing_range;
    };

    const CORBA::PolicyType MAX_HOPS_POLICY_TYPE = 34;
    local interface MaxHopsPolicy : CORBA::Policy {
        readonly attribute unsigned short max_hops;
    };

    // Router Delivery-ordering Policy (default = ORDER_TEMPORAL)
    const CORBA::PolicyType QUEUE_ORDER_POLICY_TYPE = 35;
    local interface QueueOrderPolicy : CORBA::Policy {
        readonly attribute Ordering              allowed_orders;
    };
};
```

17.2.1 Rebind Support

Rebind support discussed in this sub clause refers to the act of rebinding an object reference that has already been bound once. The policies discussed here do not affect the initial binding of an object reference.

17.2.1.1 typedef short RebindMode

Describes the level of transparent rebinding that may occur during the course of an invocation on an Object. Values of type **RebindMode** are used in conjunction with a **RebindPolicy**, as described in interface RebindPolicy on page 420. All non-negative values are reserved for use in OMG specifications. Any negative value of **RebindMode** is considered a vendor extension.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- **TRANSPARENT** - allows the ORB to silently handle object-forwarding and necessary reconnection during the course of making a remote request. This is equivalent to the only defined *CORBA* ORB behavior.
- **NO_REBIND** - allows the ORB to silently handle reopening of closed connections while making a remote request, but prevents any transparent object-forwarding that would cause a change in client-visible effective QoS policies. When this policy is in effect, only explicit rebinding (through **CORBA::Object::validate_connection**) is allowed.
- **NO_RECONNECT** - prevents the ORB from silently handling object-forwards or the reopening of closed connections. When this policy is in effect, only explicit rebinding and reconnection (through **CORBA::Object::validate_connection**) is allowed.

17.2.1.2 interface RebindPolicy

This interface is a local object derived from **CORBA::Policy**. It is used to indicate whether the ORB may transparently rebind once successfully *bound* to a target. For GIOP-based protocols an object reference is considered bound once it is in a state where a **LocateRequest** message would result in a **LocateReply** message with status **OBJECT_HERE**. If the effective Policy of this type has a **rebind_mode** value of **TRANSPARENT** (always the default and the only valid value in *CORBA*), the ORB will silently handle any subsequent **LocateReply** messages with **OBJECT_FORWARD** status or Reply messages with **LOCATION_FORWARD** status. The effective policies of other types for this object reference may change from invocation to invocation. If the effective Policy of this type has a **rebind_mode** value of **NO_REBIND**, the ORB will raise a REBIND system exception if any rebind handling would cause a client-visible change in policies. This could happen under the following circumstances:

- The client receives a **LocateReply** message with an **OBJECT_FORWARD** status and a new IOR that has policy requirements incompatible with the effective policies currently in use.
- The client receives a Reply message with **LOCATION_FORWARD** status and a new IOR that has policy requirements incompatible with the effective policies currently in use.

If the effective Policy of this type has a **rebind_mode** value of **NO_RECONNECT**, the ORB will raise a REBIND system exception if any rebind handling would cause a client-visible change in policies, or if a new connection must be opened. This includes the reopening of previously closed connections as well as the opening of new connections if the target address changes (for example, due to a **LOCATION_FORWARD** reply). For connectionless protocols, the meaning of this effective policy must be specified, or it must be defined that **NO_RECONNECT** is an equivalent to **NO_REBIND**. Regardless of the effective **RebindPolicy**, rebind or reconnect can always be explicitly requested through an invocation of **CORBA::Object::validate_connection**. When instances of **RebindPolicy** are created, a value of type **RebindMode** is passed to **CORBA::ORB::create_policy**. This policy is only applicable as a client-side override. When an instance of **RebindPolicy** is propagated within a **PolicyValue** in an **INVOCATION_POLICIES** Service Context, the **ptype** has value **REBIND_POLICY_TYPE** and the **pvalue** is a CDR encapsulation containing a **RebindMode**.

17.2.2 Synchronization Scope

17.2.2.1 typedef short SyncScope

Describes the level of synchronization for a request with respect to the target. Values of type **SyncScope** are used in conjunction with a **SyncScopePolicy**, as described in interface SyncScopePolicy on page 421, to control the behavior of oneway operations. All non-negative values are reserved for use in OMG specifications. Any negative value of **SyncScope** is considered a vendor extension.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- **SYNC_NONE** - equivalent to one allowable interpretation of *CORBA* oneway operations. The ORB returns control to the client (e.g., returns from the method invocation) before passing the request message to the transport protocol. The client is guaranteed not to block. Since no reply is returned from the server, no location-forwarding can be done with this level of synchronization.
- **SYNC_WITH_TRANSPORT** - equivalent to one allowable interpretation of *CORBA* oneway operations. The ORB returns control to the client only after the transport has accepted the request message. This in itself gives no guarantee that the request will be delivered, but in conjunction with knowledge of the characteristics of the transport may provide the client with a useful degree of assurance. For example, for a direct message over TCP, **SYNC_WITH_TRANSPORT** is not a stronger guarantee than **SYNC_NONE**. However, for a store-and-forward transport, this QoS provides a high level of reliability. Since no reply is returned from the server, no location-forwarding can be done with this level of synchronization.
- **SYNC_WITH_SERVER** - the server-side ORB sends a reply before invoking the target implementation. If a reply of **NO_EXCEPTION** is sent, any necessary location-forwarding has already occurred. Upon receipt of this reply, the client-side ORB returns control to the client application. This form of guarantee is useful where the reliability of the network is substantially lower than that of the server. The client blocks until all location-forwarding has been completed. For a server using a POA, the reply would be sent after invoking any ServantManager, but before delivering the request to the target Servant.
- **SYNC_WITH_TARGET** - equivalent to a synchronous, non-oneway operation in *CORBA*. The server-side ORB shall only send the reply message after the target has completed the invoked operation. Note that any **LOCATION_FORWARD** reply will already have been sent prior to invoking the target and that a **SYSTEM_EXCEPTION** reply may be sent at anytime (depending on the semantics of the exception). Even though it was declared oneway, the operation actually has the behavior of a synchronous operation. This form of synchronization guarantees that the client knows that the target has seen and acted upon a request. As with *CORBA*, only with this highest level of synchronization can the OTS be used. Any operations invoked with lesser synchronization precludes the target from participating in the client's current transaction.

17.2.2.2 interface SyncScopePolicy

This interface is a local object derived from **CORBA::Policy**. It is applied to oneway operations to indicate the synchronization scope with respect to the target of that operation request. It is ignored when any non-oneway operation is invoked. This policy is also applied when the DII is used with a flag of **INV_NO_RESPONSE** since the implementation of the DII is not required to consult an interface definition to determine if an operation is declared oneway. The default value of this Policy is not defined. Applications must explicitly set an ORB-level **SyncScopePolicy** to ensure portability across ORB implementations. When instances of **SyncScopePolicy** are created, a value of type **Messaging::SyncScope** is passed to **CORBA::ORB::create_policy**. This policy is only applicable as a client-side override. The client's **SyncScopePolicy** is propagated within a request in the RequestHeader's **response_flags** as described in GIOP Request Header.

17.2.3 Request and Reply Priority

17.2.3.1 struct PriorityRange

This structure describes a range of priorities. A **PriorityRange** with minimum Priority greater than maximum Priority is invalid.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.2.3.2 interface RequestPriorityPolicy

This interface is a local object derived from **CORBA::Policy**. It is used to indicate the valid range of priorities, which may be associated with an operation request. This value is used by Routers when the effective **QueueOrderPolicy** has the value **ORDER_PRIORITY**. Higher Priority values indicate a higher priority. When instances of **RequestPriorityPolicy** are created, a value of type **Messaging::PriorityRange** is passed to **CORBA::ORB::create_policy**. An instance of **RequestPriorityPolicy** may be specified when creating a POA (and therefore may be represented in Object references). In addition, an Object reference's **RequestPriorityPolicy** may be overridden by the client. If set on both the client and server, reconciliation is performed by intersecting the server-specified **RequestPriorityPolicy** range with the range of the client's effective override. When an instance of **RequestPriorityPolicy** is propagated within a **PolicyValue** in a **TAG_POLICIES** Profile Component or **INVOCATION_POLICIES** Service Context, the **ptype** has value **REQUEST_PRIORITY_POLICY_TYPE** and the **pvalue** is a CDR encapsulation containing a **Messaging::PriorityRange**.

17.2.3.3 interface ReplyPriorityPolicy

This interface is a local object derived from **CORBA::Policy**. It is used to indicate the valid range of priorities, which may be associated with the reply to an operation request. This value is used by Routers when the effective **QueueOrderPolicy** has the value **ORDER_PRIORITY**. Higher Priority values indicate a higher priority. When instances of **ReplyPriorityPolicy** are created, a value of type **Messaging::PriorityRange** is passed to **CORBA::ORB::create_policy**. An instance of **ReplyPriorityPolicy** may be specified when creating a POA (and therefore may be represented in Object references). In addition, an Object reference's **ReplyPriorityPolicy** may be overridden by the client. If set on both the client and server, reconciliation is performed by intersecting the server-specified **ReplyPriorityPolicy** range with the range of the client's effective override. When an instance of **ReplyPriorityPolicy** is propagated within a **PolicyValue** in a **TAG_POLICIES** Profile Component or **INVOCATION_POLICIES** Service Context, the **ptype** has value **REPLY_PRIORITY_POLICY_TYPE** and the **pvalue** is a CDR encapsulation containing a **Messaging::PriorityRange**.

17.2.4 Request and Reply Timeout

This specification describes the lifetime of requests and replies in terms of the structured type from the CORBA Time Service Specification. This describes time as a 64-bit value, which is the number of 100 nano-seconds from 15 October 1582 00:00, along with inaccuracy and time zone information.

17.2.4.1 interface RequestStartTimePolicy

This interface is a local object derived from **CORBA::Policy**. It is used to indicate the valid start time after which a request may be delivered to its target, and is applied to both synchronous and asynchronous invocations. When instances of **RequestStartTimePolicy** are created, a value of type **TimeBase::UtcT** containing an absolute time is passed to **CORBA::ORB::create_policy**. This policy is only applicable as a client-side override. When an instance of **RequestStartTimePolicy** is propagated within a **PolicyValue** in an **INVOCATION_POLICIES** Service Context, the **ptype** has value **REQUEST_START_TIME_POLICY_TYPE** and the **pvalue** is a CDR encapsulation containing a **TimeBase::UtcT**.

If the effective **RoutingPolicy** is **NONE**, the client ORB shall refrain from transmitting the request to the target until after the specified start time. Otherwise, the client ORB and all but the last hop router are free to transmit the request immediately, and the last hop router shall delay the request until the specified start time.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.2.4.2 interface RequestEndTimePolicy

This interface is a local object derived from **CORBA::Policy**. It is used to indicate the time after which a request may no longer be delivered to its target. This policy is applied to both synchronous and asynchronous invocations. When instances of **RequestEndTimePolicy** are created, a value of type **TimeBase::UtcT** is containing an absolute time passed to **CORBA::ORB::create_policy**. This policy is only applicable as a client-side override. When an instance of **RequestEndTimePolicy** is propagated within a **PolicyValue** in an **INVOCATION_POLICIES** Service Context, the **ptype** has value **REQUEST_END_TIME_POLICY_TYPE** and the **pvalue** is a CDR encapsulation containing a **TimeBase::UtcT**.

The client ORB, all routers and the target ORB shall check to see if the end time specified in the **RequestEndTimePolicy** associated with a request has expired and the request is yet to be delivered to the target. If so, it shall discard the request and return the system exception TIMEOUT with standard minor code 2.

17.2.4.3 interface ReplyStartTimePolicy

This interface is a local object derived from **CORBA::Policy**. It is used to indicate the valid start time after which a reply may be delivered to the client. This policy is applied to both synchronous and asynchronous invocations. When instances of **ReplyStartTimePolicy** are created, a value of type **TimeBase::UtcT** containing an absolute time is passed to **CORBA::ORB::create_policy**. This policy is only applicable as a client-side override. When an instance of **ReplyStartTimePolicy** is propagated within a **PolicyValue** in an **INVOCATION_POLICIES** Service Context, the **ptype** has value **REPLY_START_TIME_POLICY_TYPE** and the **pvalue** is a CDR encapsulation containing a **TimeBase::UtcT**.

If the **RoutePolicy** is **ROUTE_NONE**, the client ORB shall delay delivering the reply until the start time has been reached. Otherwise, the target ORB and all but the last hop router are free to transmit the reply immediately, and the last hop router shall delay transmission of the reply to the client until the start time has been reached.

17.2.4.4 interface ReplyEndTimePolicy

This interface is a local object derived from **CORBA::Policy**. It is used to indicate the time after which a reply may no longer be obtained or returned to the client. This policy is applied to both synchronous and asynchronous invocations. When instances of **ReplyEndTimePolicy** are created, a value of type **TimeBase::UtcT** containing an absolute time is passed to **CORBA::ORB::create_policy**. This policy is only applicable as a client-side override. When an instance of **ReplyEndTimePolicy** is propagated within a **PolicyValue** in an **INVOCATION_POLICIES** Service Context, the **ptype** has value **REPLY_END_TIME_POLICY_TYPE** and the **pvalue** is a CDR encapsulation containing a **TimeBase::UtcT**.

The client ORB, all routers and the target ORB shall check to see if the end time specified in the **ReplyEndTimePolicy** associated with a request has expired and a reply has not yet been delivered to the client. If so, it shall discard the reply and return the system exception TIMEOUT with standard minor code 3.

17.2.4.5 interface RelativeRequestTimeoutPolicy

This interface is a local object derived from **CORBA::Policy**. It is used to indicate the relative amount of time for which a Request may be delivered. After this amount of time the Request is cancelled. This policy is applied to both synchronous and asynchronous invocations. If asynchronous invocation is used, this policy only limits the amount of time during which the request may be processed. Assuming the request completes within the specified timeout, the reply will never be discarded due to timeout. When instances of **RelativeRequestTimeoutPolicy** are created, a value of type **TimeBase::TimeT** containing a relative time is passed to **CORBA::ORB::create_policy**. This policy is only applicable as a client-side override. When an instance of **RelativeRequestTimeoutPolicy** is propagated within a

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

PolicyValue in an **INVOCATION_POLICIES** Service Context, the **ptype** has value **REQUEST_END_TIME_POLICY_TYPE** and the **pvalue** is a CDR encapsulation containing the **relative_expiry** converted into a **TimeBase::UtcT** end time (as in the case of **RequestEndTimePolicy**).

Since a **RelativeRequestTimeoutPolicy** is converted to a **RequestEndTimePolicy** before transmitting the request to the target ORB, see interface RequestEndTimePolicy on page 423 for the required behavior of an ORB or router when the timeout expires.

17.2.4.6 interface RelativeRoundtripTimeoutPolicy

This interface is a local object derived from **CORBA::Policy**. It is used to indicate the relative amount of time for which a Request or its corresponding Reply may be delivered. After this amount of time, the Request is cancelled (if a response has not yet been received from the target) or the Reply is discarded (if the Request had already been delivered and a Reply returned from the target). This policy is applied to both synchronous and asynchronous invocations.

When instances of **RelativeRoundtripTimeoutPolicy** are created, a value of type **TimeBase::TimeT** containing a relative time is passed to **CORBA::ORB::create_policy**. This policy is only applicable as a client-side override. When an instance of **RelativeRoundtripTimeoutPolicy** is propagated within a **PolicyValue** in an **INVOCATION_POLICIES** Service Context, the **ptype** has value **REPLY_END_TIME_POLICY_TYPE** and the **pvalue** is a CDR encapsulation containing the **relative_expiry** converted into a **TimeBase::UtcT** end time (as in the case of **ReplyEndTimePolicy**).

Since a **RelativeRoundtripTimeoutPolicy** is converted to a **ReplyEndTimePolicy** before transmitting the request to the target ORB, see interface ReplyEndTimePolicy on page 423 for the required behavior of an ORB or router when the timeout expires.

17.2.5 Routing

17.2.5.1 typedef short RoutingType

Describes the type of Routing to be used for invocations on an Object reference. Values of type **RoutingType** are used in conjunction with a **RoutingPolicy** as described in interface RoutingPolicy on page 424. All non-negative values are reserved for use in OMG specifications. Any negative value of **RoutingType** is considered a vendor extension.

- **ROUTE_NONE** - Synchronous or Deferred Synchronous delivery is used. No Routers will be used to aid in the delivery of the request.
- **ROUTE_FORWARD** - Asynchronous delivery is used. The request is made through the use of a Router and not delivered directly to the target by the client ORB.
- **ROUTE_STORE_AND_FORWARD** - Asynchronous TII is used. The request is made through the use of a Router that persistently stores the request before attempting delivery.

17.2.5.2 struct RoutingTypeRange

This structure describes a range of routing types. A **RoutingTypeRange** with minimum **RoutingType** greater than maximum **RoutingType** is invalid.

17.2.5.3 interface RoutingPolicy

This interface is a local object derived from **CORBA::Policy**. It is used to indicate whether or not the ORB must ensure delivery of a request through the use of queueing. If the effective Policy of this type has a **RoutingTypeRange** with min value of **ROUTE_FORWARD** or **ROUTE_STORE_AND_FORWARD**, the interoperable Routing protocol described in

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

III - Introduction on page 460 is used. This policy does not apply to synchronous invocations. If, for example, the min is **ROUTE_NONE** and the max is **ROUTE_FORWARD**, the Routing protocol will normally be used but a direct connection may be used if available. When instances of **RoutingPolicy** are created, a value of type **RoutingTypeRange** is passed to **CORBA::ORB::create_policy**. An instance of **RoutingPolicy** may be specified when creating a POA (and therefore may be represented in Object references). In addition, a POA's **RoutingPolicy** is visible to clients through the Object references it creates, and reconciled with the client's override. If set on both the client and server, reconciliation is performed by intersecting the server-specified **RoutingPolicy** range with the range of the client's effective override. When an instance of **RoutingPolicy** is propagated within a **PolicyValue** in a **TAG_POLICIES** Profile Component or **INVOCATION_POLICIES** Service Context, the **ptype** has value **ROUTING_POLICY_TYPE** and the **pvalue** is a CDR encapsulation containing a **Messaging::RoutingTypeRange**.

17.2.5.4 interface MaxHopsPolicy

This interface is a local object derived from **CORBA::Policy**. It is used to indicate the maximum number of routing hops that can occur when routing a request from the client to the target. When instances of **MaxHopsPolicy** are created, a value of type unsigned short is passed to **CORBA::ORB::create_policy**. This policy is only applicable as a client-side override. When an instance of **MaxHopsPolicy** is propagated within a **PolicyValue** in an **INVOCATION_POLICIES** Service Context, the **ptype** has value **MAX_HOPS_POLICY_TYPE** and the **pvalue** is a CDR encapsulation containing an **unsigned short**.

17.2.6 Queue Ordering

17.2.6.1 typedef short Ordering

Describes the ordering policy for the consideration of routers that prioritize delivery of requests. Values of type Ordering are used in conjunction with a **QueueOrderPolicy** as described in interface QueueOrderPolicy on page 425. This policy is only used if the effective **RoutingType** is at least **ROUTE_FORWARD** (which implies the use of a **Router**). Support for multiple ordering policies is indicated by "or"-ing together individual values in a combined Ordering.

- **ORDER_ANY** - the client doesn't care in what order its requests are processed.
- **ORDER_TEMPORAL** - the client wants to be sure that its requests are processed in the order in which they were issued. **ORDER_TEMPORAL** is the default.
- **ORDER_PRIORITY** - the client wants its requests processed based on the priority assigned in the QoS structure described below.
- **ORDER_DEADLINE** - the client wants its requests ordered so that those whose time_to_live is about to expire are moved to the front of the queue.

17.2.6.2 interface QueueOrderPolicy

This interface is a local object derived from **CORBA::Policy**. It is used to indicate the basis upon which a Router orders delivery of requests. When instances of **QueueOrderPolicy** are created, a value of type **Messaging::Ordering** is passed to **CORBA::ORB::create_policy**. This specified **Ordering** value can be the result of "or"-ing together individual orderings. An instance of **QueueOrderPolicy** may be specified when creating a POA (and therefore may be represented in Object references). In addition, an Object reference's **QueueOrderPolicy** may be overridden by the client. If set on both the client and server, reconciliation is performed by intersecting the server-specified list of supported Ordering values with the list of values in the client's effective override. When an instance of **QueueOrderPolicy** is propagated within a **PolicyValue** in a **TAG_POLICIES** Profile Component or **INVOCATION_POLICIES** Service Context, the **ptype** has value **QUEUE_ORDER_POLICY_TYPE** and the **pvalue** is a CDR encapsulation containing a **Messaging::Ordering**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.3 Propagation of Messaging QoS

This sub clause defines the profile Component through which QoS requirements are expressed in an object reference, and the Service Context through which QoS requirements are expressed as part of a GIOP request.

```
module Messaging {
    typedef CORBA::OctetSeq PolicyData;
    struct PolicyValue {
        CORBA::PolicyType          ptype;
        PolicyData                 pvalue;
    };
    typedef sequence<PolicyValue> PolicyValueSeq;

    const IOP::ComponentId TAG_POLICIES =          2;
    const IOP::ServiceId INVOCATION_POLICIES =     7;
};
```

17.3.1 Structures

PolicyValue

This structure contains the value corresponding to a Policy of the **PolicyType** indicated by its **ptype**. This representation allows the compact transmission of QoS policies within IORs and Service Contexts. The format of **pvalue** for each type is given in the specification of that Policy.

17.3.2 Messaging QoS Profile Component

A new **IOP::TaggedComponent** is defined for transmission of QoS policies within interoperable Object References. The body of this Component is a CDR encapsulation containing a **Messaging::PolicyValueSeq**. When creating Object references, Portable Object Adapters may encode the relevant policies with which it was created in this **TaggedComponent**. POA Policies that are exported in this way are clearly noted as *client-exposed* in their definitions. These policies are reconciled with the effective client-side override when clients invokes operations on that reference. For example, if a POA is created with a **RequestPriorityPolicy** with minimum value 0 and maximum value 10, all Object references created by that POA will have that default **RequestPriorityPolicy** encoded in their IOR. Furthermore, if a client sets an overriding **RequestPriorityPolicy** with both minimum and maximum of 5 (the client requires its requests to have a priority of value 5), the ORB will reconcile the effective Policy for any invocations on this Object reference to have a priority of 5 (since this value is within the range of priorities allowed by the target). On the other hand, if the client set an override with minimum value of 11, any invocation attempts would raise the system exception INV_POLICY.

17.3.3 Messaging QoS Service Context

A new **IOP::ServiceContext** is defined for transmission of QoS policies within GIOP requests and replies. The body of this Context is a CDR encapsulation containing a **Messaging::PolicyValueSeq**.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Section II - Messaging Programming Model

17.4 Section II - Introduction

Asynchronous Method Invocations allow clients to make non-blocking requests on a target. The AMI is treated as a client-side language mapping issue only. In most cases, server-side implementations are not required to change as from the server-side programmer's point of view all invocations can be treated identically regardless of their synchronicity characteristics. In certain situations, such as with transactional servers, the asynchrony of a client does matter and requires server-side changes if expected to handle transactional asynchronous requests. This specific issue is addressed in Annex B, Transaction Service on page 491.

Clients may, at any time, make either asynchronous or synchronous requests on the target. Two models of asynchronous requests are supported: callback and polling. In the *callback* model, the client passes a reference to a reply handler (a client-side CORBA object implementation that handles the reply for a client request), in addition to the normal parameters needed by the request. The reply handler interface defines operations to receive the results of that request (including **inout** and **out** values and possible exceptions). The **ReplyHandler** is a normal CORBA object that is implemented by the programmer as with any object implementation. In the polling model, the client makes the request passing in all the parameters needed for the invocation, and is returned a Poller object that can be queried to obtain the results of the invocation. This Poller is an instance of a valuetype.

AMI may be used in single- and multi-threaded applications. AMI calls may have any legal return type, parameters, and contexts. AMI operations do not raise user exceptions. Rather, user exceptions are passed to the implemented type-specific **ReplyHandler** or returned from the type-specific Poller. If an AMI operation raises a system exception with a completion status of **COMPLETED_NO**, the request has not been made. This clearly distinguishes exceptions raised by the server (which are returned via the **ReplyHandler** or Poller) from local exceptions that caused the AMI to fail.

This sub clause focuses entirely on the static (typed) asynchronous invocations that are based on the interface that is the target of the operation. This sub clause describes the mapping for the generated asynchronous method signatures. It also describes the generated reply handlers that are passed to those async methods when the callback model is used, and the generated poller values that are returned from those async methods when the polling model is used. The AMI mapping contains an IDL to "implied-IDL" mapping, which defines the new operations and interfaces required to perform asynchronous invocations and obtain the replies to these requests. The new interfaces and values defined in this implied-IDL are considered to be real IDL since they can correspond to entries in the Interface Repository and have behavior consistent with all other definitions in IDL. In several cases, this implied-IDL adds new operations to existing interfaces. These new asynchronous stub interfaces are not considered to be real IDL in that they do not correspond to entries in the Interface Repository. The distinction between these types of implied-IDL is made clear in the rest of this sub clause. In general, the implied-IDL is used to avoid explicitly mapping the AMI API to each of the currently supported languages.

When a messaging-enabled IDL code generator is run on an interface, the following is performed in addition to the processing specified in *CORBA*:

- A Servant mapping is generated for a type-specific **ReplyHandler** from which the client application derives its **ReplyHandler** implementation. No type-specific **ReplyHandler** stubs need be generated, but their absence is not a requirement. The Servant base is generated as if from an IDL interface with a definition as specified in Type-Specific ReplyHandler Mapping on page 434.
- A type-specific **Poller valuetype** is generated. The implementation of this **Poller** is provided by the messaging-aware ORB. The language-specific generated code corresponds to a **valuetype** as if it were defined in IDL as specified in Type-Specific Poller Mapping on page 438.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- Asynchronous request operations are generated with signatures exactly as if the operations were declared on the original interface. The implied-IDL signature of these operations is specified in Async Operation Mapping on page 429. The implied-IDL is used entirely so that each individual supported language mapping need not be given for the asynchronous request operations.

NOTE: These implied-IDL operations are not intended to be seen by the Object implementation and are not implemented by the Servant. They are purely a client-side construct for describing the operation signatures for generated code.

- Furthermore, these operations are not part of the interfaces **CORBA::InterfaceDef** and do not correspond to synchronous operations. The generated code for these operations interacts with a messaging-aware ORB in ways outside of the scope of this sub clause. The mechanism of this interaction is specified for interoperability purposes in Message Routing on page 457. An application programmer need not be aware of this mechanism.

17.5 Running Example

A running example is used throughout this sub clause to clarify the generation of the new typed asynchronous invocation stubs, the new reply handling interfaces for receiving callback responses, and the new poller values for querying the status of an outstanding request. The example features a simple stock portfolio manager interface. Most importantly, the interface includes operations that cover all cases of operation signature:

- attributes
- in arguments
- inout arguments
- out arguments
- return values
- user exceptions

Operations declared oneway are not mapped to asynchronous invocation stubs because they are already asynchronous in nature.

```
// Original IDL
exception InvalidStock { string sym; };

interface StockManager {
    attribute string stock_exchange_name;

    boolean     add_stock(in string symbol, in double quote);
    void        edit_stock(in string symbol, in double new_quote)
        raises(InvalidStock);
    void        remove_stock(in string symbol, out double quote)
        raises(InvalidStock);

    boolean         find_closest_symbol(inout string symbol);
    double get_quote(in string symbol) raises(InvalidStock);
};
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.6 Async Operation Mapping

For each operation in an interface, corresponding callback and polling asynchronous method signatures are generated. Note that no callback and polling asynchronous method signatures are generated for any operations or attributes of abstract interfaces.

Even though vanilla oneway operations have no associated reply, under certain circumstance, like for **SyncScope** value of **SYNC_TARGET** or **SYNC_SERVER**, it may be useful and necessary to receive a reply (either normal or exceptional). The **sendc_** and **sendp_** operations therefore need to be created for oneway operations too.

Note that for other **SyncScopes** (**SYNC_NONE** and **SYNC_TRANSPORT**), invocations of **sendc_** oneway operations should result in an immediate callback, and invocations of **sendp_** oneway operations should result in a poll becoming immediately ready.

Due to the way in which identifier names are generated in the implied IDL, in order to avoid name clashes, any IDL that is meant to be used with Asynchronous Messaging must not contain any identifiers that have the string “AMI_” as a prefix.

These signatures are described in implied-IDL, which is used to generate language-specific operation signatures. The implementation of these methods must generate a method invocation as described in Message Routing on page 457. Note that these generated operations are not included in the interface’s definition (**CORBA::InterfaceDef**). These operations do not raise user exceptions. Just as with the currently specified **CORBA::Request::send operation**, they can (but are not required to) raise system exceptions. For explanatory purposes, the sub clauses below show the Callback and Polling implied-IDL in separate pieces. Logically, the IDL compiler deals with async as if the IDL included all three pieces: the original IDL and the implied IDL for both async models.

17.6.1 Callback Model Signatures (sendc)

When the callback model is used, the client supplies a reply handler when making the asynchronous invocation. The interface’s operations and attributes are mapped to implied-IDL operations with names prefixed by “**sendc_**”. If this implied-IDL operation name conflicts with existing operations on the interface or any of the interface’s base interfaces, “**ami_**” strings are inserted between “**sendc_**” and the original operation name until the implied-IDL operation name is unique.

17.6.1.1 Implied-IDL for Operations

The signature of the implied-IDL for a given IDL operation is:

- void return type, followed by;
- **sendc_<opName>** where **opName** is the name of the operation.

The async callback version takes the following arguments in order:

- An object reference to a type-specific **ReplyHandler** as described in Type-Specific ReplyHandler Mapping on page 434, with the parameter name **ami_handler**. If a nil **ReplyHandler** reference is specified when this operation is invoked, no response will be returned for this invocation. A system exception may be raised by the ORB during evaluation of the request, but once **sendc** returns, no further results of the operation will be made available. This is equivalent to setting the **CORBA::INV_NO_RESPONSE** flag when making a DII deferred request.
- Each of the **in** and **inout** arguments in the order that they appeared in the operation's declaration in IDL, all with a parameter attribute of in and with the type specifier and parameter name of the original argument.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- **out** arguments are ignored (i.e., are not part of the async signature).

The implied-IDL operation signature has a context expression identical to the one from the original operation (if any is present).

17.6.1.2 Implied-IDL for Attributes

The signature of the implied-IDL for the callback model getter and setter operations corresponding to an interface's attribute is as follows.

- Setter operations are only generated for attributes that are not defined readonly
- void return type, followed by the operation name, which to distinguish between the getter and setter operations for the attribute is given by either:
  - **sendc_get_<attributeName>** for reading the attribute value, where attributeName is the name of the attribute, or
  - **sendc_set_<attributeName>** for setting the attribute value, where attributeName is the name of the attribute that is not defined readonly.

The callback implied-IDL operations take the following arguments in order:

- An object reference of a type-specific **ReplyHandler** as described in Section 17.8, Type-Specific ReplyHandler Mapping, on page 434, with the parameter name **ami_handler**.
- The additional arguments for asynchronous implied-IDL operations for **attributes** are as follows:
  - For the attribute's generated **get** operation, there are no additional arguments.
  - For the attribute's generated **set** operation, there is one additional argument, in **<attrType>** attr_**<attributeName>**, where **attrType** is the type of the attribute, and **attributeName** is the name of that attribute. The **set** operation is only generated for attributes that are not defined **readonly**.

17.6.1.3 Example

The following implied-IDL is generated from the interface definitions used in the running example:

```
// AMI implied-IDL including callback operations
// for original example IDL defined in  Section 17.5

exception InvalidStock { string sym; };

interface AMI_StockManagerHandler;

interface StockManager {

    // Original operation Declarations
    attribute string stock_exchange_name;
    boolean     add_stock(in string symbol, in double quote);
    void        edit_stock(in string symbol, in double new_quote)
        raises(InvalidStock);
    void        remove_stock(in string symbol, out double quote)
        raises(InvalidStock);
    boolean     find_closest_symbol(inout string symbol);
    double get_quote(in string symbol) raises(InvalidStock);
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    // Async Callback operation Declarations
    void sendc_get_stock_exchange_name(
            in AMI_StockManagerHandler ami_handler);
    void sendc_set_stock_exchange_name(
            in AMI_StockManagerHandler ami_handler,
            in string attr_stock_exchange_name);

    void    sendc_add_stock(
            in AMI_StockManagerHandler ami_handler, in string symbol,
            in double quote);
    void    sendc_edit_stock(
            in AMI_StockManagerHandler ami_handler,
            in string symbol, in double new_quote);
    void    sendc_remove_stock(
            in AMI_StockManagerHandler ami_handler,
            in string symbol);
    void    sendc_find_closest_symbol(
            in AMI_StockManagerHandler ami_handler,
            in string symbol);
    void    sendc_get_quote(
            in AMI_StockManagerHandler ami_handler,
            in string symbol);
};
```

17.6.2 Polling Model Signatures (sendp)

When the polling model is used, the client is returned a queriable poller when making the asynchronous invocation. The interface's operations and attributes are mapped to implied-IDL operations with names prefixed by **sendp**_. If this implied-IDL operation name conflicts with existing operations on the interface or any of the interface's base interfaces, **ami**_ strings are inserted between **sendp**_ and the original operation name until the implied-IDL operation name is unique.

17.6.2.1 Implied-IDL for Operations

The signature of the implied-IDL for a given IDL operation is:

- A type-specific Poller return type as described in Type-Specific Poller Mapping on page 438, followed by **sendp_<opName>** where **opName** is the name of the operation.

The async polling version takes the following parameters in order:

- Each of the **in** and **inout** arguments in the order that they appeared in the operation's declaration in IDL, all with a parameter attribute of **in** and with the type specifier and parameter name of the original argument.
- **out** arguments are ignored (i.e., are not part of the async signature).

The implied-IDL operation signature has a context expression identical to the one from the original operation (if any is present).

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.6.2.2 Implied-IDL for Attributes

The signature of the implied-IDL for the polling model getter and setter operations corresponding to an interface's attribute is as follows:

- Setter operations are only generated for attributes that are not defined readonly.
- A type-specific Poller return type as described in *Type-Specific Poller Mapping on page 438*, followed by the operation name, which to distinguish between the getter and setter operations for the attribute is given by either:
  - **sendp_get_<attributeName>** for reading the attribute value, where attributeName is the name of the attribute, or
  - **sendp_set_<attributeName>** for setting the attribute value, where attributeName is the name of the attribute that is not defined readonly.
- Asynchronous implied-IDL operations for attributes have argument lists as follows:
  - For the attribute's generated **get** operation, there are no arguments.
  - For the attribute's generated **set** operation, there is one argument, in **<attrType> attr_<attributeName>**, where **attrType** is the type of the attribute, and **attributeName** is the name of that attribute. The **set** operation is only generated for attributes that are not defined readonly.

17.6.2.3 Example

The following implied-IDL is generated from the interface definitions used in the running example:

```
// AMI implied-IDL including polling operations
// for original example IDL defined in  Section 17.5
exception InvalidStock { string sym; };

valuetype AMI_StockManagerPoller;

interface StockManager {
    // Original operation Declarations
    attribute string stock_exchange_name;
    boolean     add_stock(in string symbol, in double quote);
    void        edit_stock(in string symbol, in double new_quote)
        raises(InvalidStock);
    void        remove_stock(in string symbol, out double quote)
        raises(InvalidStock);
    boolean     find_closest_symbol(inout string symbol);
    double get_quote(in string symbol) raises(InvalidStock);

    // Async Polling operation Declarations
    AMI_StockManagerPoller sendp_get_stock_exchange_name();
    AMI_StockManagerPoller sendp_set_stock_exchange_name(
            in string attr_stock_exchange_name);
    AMI_StockManagerPoller sendp_add_stock(
            in string symbol,
            in double quote);
    AMI_StockManagerPoller sendp_edit_stock(
            in string symbol, in double new_quote);
    AMI_StockManagerPoller sendp_remove_stock(
```


© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            in string symbol);
    AMI_StockManagerPoller sendp_find_closest_symbol(
            in string symbol);
    AMI_StockManagerPoller sendp_get_quote(
            in string symbol);
};
```

17.7 Exception Delivery in the Callback Model

The **ReplyHandler** interface is expressed in IDL and thus cannot have operations that take exceptions as arguments. Furthermore, the most natural way for a **ReplyHandler** to deal with exceptions is by invoking some operation that raises exceptions, not through inspecting operation parameters. Therefore, exception replies are propagated to the **ReplyHandler** in the form of a type-specific **Messaging::ExceptionHolder valuetype** instance that contains the marshaled exception as its state and has **raise_exception** and **raise_exception_with_list** operations for raising the encapsulated exception.

17.7.1 Messaging::ExceptionHolder valuetype

The **Messaging::ExceptionHolder** valuetype encapsulates the exception data and enough information to turn that data back into a raised exception.

```
// IDL
module Messaging {

    // ... all the other stuff

    typedef CORBA::OctetSeq MarshaledException;
    native UserExceptionBase;
    valuetype ExceptionHolder {
        void raise_exception() raises (UserExceptionBase);
        void raise_exception_with_list(
            in CORBA::ExceptionList      exc_list
        ) raises (UserExceptionBase);
        private boolean is_system_exception;
        private boolean byte_order;
        private MarshaledException marshaled_exception;
    };
};
```

- **raise_exception()** - This method is used by applications to raise exception from the encapsulated **marshaled_exception** member.
- **raise_exception_with_list()** - If **is_system_exception** is true, this function is same as **raise_exception()**. Otherwise, this method raises an exception from the **marshaled_exception** using an application provided user exception list. It is useful and should only be used when the given exception holder is not from a skeleton reply handler's xxx_excep() method. For instance, it is from a DSI reply handler servant or from another ORB runtime. In these cases, the exception holder may not have an internal user exception list available.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- **UserExceptionBase** - Language mapping of this native type should allow any user exception to be raised from this method. For instance, it is mapped to CORBA::UserException in C++ and to org.omg.CORBA.UserException in java. As usual, system exceptions do not need to be in the raises clause for raising them from this method.

17.8 Type-Specific ReplyHandler Mapping

For each interface, a type-specific reply handler is generated by the IDL compiler. The client application implements and registers a reply handler with each asynchronous request and receives a callback when the reply is returned for that request. The interface name of the type-specific handler is **AMI_<ifaceName>Handler**, where **ifaceName** is the original unqualified interface name. If the interface **ifaceName** derives from some other IDL interface **baseName**, then the handler for **ifaceName** is derived from **AMI_<baseName>**, but if it does not, then it is derived from the generic **Messaging::ReplyHandler**. If the interface name **AMI_<ifaceName>Handler** conflicts with an existing identifier, uniqueness is obtained by inserting additional "AMI_" prefixes before the **ifaceName** until the generated identifier is unique.

When invoking an async operation, the client first generates an object reference for its **ReplyHandler** and then associates it with the request by passing the reference as an argument to the operation. The reply will be targeted to that **ReplyHandler**. So that a single **ReplyHandler** servant instance can be supplied to multiple requests, the client can assign unique **ObjectId**s for each request if the application code needs to distinguish between each of these requests at a later time. Most commonly, the application needs to access information from the calling scope while in the scope of the callback. That information can be associated with the **ReplyHandler**'s **ObjectId** by the client application at the time of invocation. Obtaining the **ReplyHandler**'s **ObjectId** within the callback implementation allows that implementation to obtain any information previously associated with the original request. Since the assignment and accessing of these **ObjectId**s is fully supported within the Portable Object Adapter defined in *CORBA*, there is no need to specify the notion of unique request ids in this document.

The **ReplyHandler** object reference will be serviced by a servant running under a POA with a particular set of POA policies. These policies are not affected by the fact that it is a **ReplyHandler**, so these Policy values have the same considerations as with any server. The POA **LifeSpanPolicy** will probably be affected depending on whether or not TII is used:

- If TII is not used, the **LifeSpanPolicy** can be either **PERSISTENT** or **TRANSIENT**, depending on the implementation. **LifeSpanPolicy** would likely be **PERSISTENT** if the same **ReplyHandler** implementation is used for replies from multiple clients. It could be **TRANSIENT** if the programmer creates the **ReplyHandler** object reference in the same process as that of the async invocation and wants the **ReplyHandler** object reference to become invalid when the creating POA terminates. In this case, replies are discarded by the ORB once the client terminates.
- If TII is used, **LifeSpanPolicy** of **PERSISTENT** is almost required since TII means that the **ReplyHandler** can validly be located in a process that can be different than the process of the original client. It is possible for **LifeSpanPolicy** to be **TRANSIENT**, but this would be a rare usage in which the original client obtains the **ReplyHandler** reference from a process other than itself. This usage would allow a **ReplyHandler** to be in effect only for the life of that other process, supporting a rather limited form of TII.

17.8.1 ReplyHandler Operations for NO_EXCEPTION Replies

For each operation declared in the interface, an operation with the following signature is defined on the generated reply handler:

- return type void**,** followed by
- the name of the operation, followed by

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- arguments in order (all “in” parameters).
  - If the original operation has a return value, the type returned by the operation declared in IDL with parameter named **ami_return_val**.
  - Each inout/out **type** name and **argument** name as they were declared in IDL.

These operations do not raise any exceptions because they are never invoked by a client and have no client to respond to such an exception. Only a system exception could be raised by such operations, and only with the effect of causing a transaction to roll back. See Annex B, Changes to Current OTS Behavior on page 491 for a discussion of the Unshared Transaction model in which a **ReplyHandler** may be invoked as part of a transaction.

For an attribute with the name “attributeName,” the following operations are generated on the reply handler: return type void**,** followed by **get_<attributeName>** for the getter (or **set_<attributeName>** for the setter operation if the attribute is not defined to be readonly). For the “get” operation, there is one argument (the setter callback operation takes no arguments): **in <attrType> ami_return_val** where the attribute of name **ami_return_val** is of type **attrType**.

There are two cases where the above mapping results in an operation with no parameters. The first is for an operation with no return value and either with no parameters or with only **in** parameters. The second is the mapping of a setter on an attribute. In these cases, it is worth noting that the only meaning that can be associated with the operation is that the AMI operation completed successfully. This is significant information, essentially an acknowledgment of completion.

17.8.2 ReplyHandler Operations for Exceptional Replies

If the AMI didn’t succeed at the target, the exception is delivered via the generated **_excep ReplyHandler** operation corresponding to the operation originally invoked. This sub clause describes the implied-IDL rules for generating these operations on the **ReplyHandler**.

For each operation, **operName**, on the original interface named **ifaceName**, an operat ion with the following signature is generated on the type-specific **ReplyHandler**:

```
void <operName>_excep(
    in Messaging::ExceptionHolder excep_holder);
```

For each attribute, **attrName**, on the original interface named **ifaceName**, an operation with the following signature is generated on the type-specific **ReplyHandler**:

```
void get_<attrName>_excep(
    in Messaging::ExceptionHolder excep_holder);
```

For each non-**readonly** attribute, **attrName**, on the original interface named **ifaceName**, an operation with the following signature is generated on the type-specific **ReplyHandler**:

```
void set_<attrName>_excep(
    in Messaging::ExceptionHolder excep_holder);
```

If the name generated by the method described above clashes with a name that already exists in the interface, “_ami” strings are inserted immediately preceding the “_excep” repeatedly, until generated IDL operation name is unique in the interface.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.8.3 Example

The example IDL causes the generation of the following additional IDL when asynchronous operations are to be used. This IDL is "real" in that the interfaces described here are CORBA objects. However, the generation of stubs for these interfaces is not required, as no client ever invokes these operations remotely in this model. The operations are invoked directly by the messaging-aware ORB when a reply becomes available.

```
// AMI implied-IDL of ReplyHandler
// for original example IDL defined in  Section 17.5
interface AMI_StockManagerHandler : Messaging::ReplyHandler {

    void get_stock_exchange_name(
        in string ami_return_val);
    void get_stock_exchange_name_excep(
        in Messaging::ExceptionHolder excep_holder);

    void set_stock_exchange_name();
    void set_stock_exchange_name_excep(
        in Messaging::ExceptionHolder excep_holder);

    void add_stock(
        in boolean ami_return_val);
    void add_stock_excep(
        in Messaging::ExceptionHolder excep_holder);
    void edit_stock();
    void edit_stock_excep(
        in Messaging::ExceptionHolder excep_holder);

    void remove_stock(
        in double quote);
    void remove_stock_excep(
        in Messaging::ExceptionHolder excep_holder);

    void find_closest_symbol(
        in boolean ami_return_val,
        in string symbol);
    void find_closest_symbol_excep(
        in Messaging::ExceptionHolder excep_holder);

    void get_quote(
        in double ami_return_val);
    void get_quote_excep(
        in Messaging::ExceptionHolder excep_holder);
};
```

17.9 Generic Poller Value

The generic base **Poller valuetype** can be queried to obtain the status of a potentially outstanding request. So that it can be registered in a **CORBA::PollableSet**, it derives from the abstract valuetype **CORBA::Pollable**. The inherited **Pollable is_ready** returns the value TRUE if and only if a reply is currently available for the outstanding request. If it

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

returns the value FALSE, the reply has not yet been returned from the target. This operation raises the system exception OBJECT_NOT_EXIST with standard minor code 5 if the reply has already been obtained by some client at the time of the query.

The Poller has the following definition:

```
module Messaging {
    abstract valuetype Poller : CORBA::Pollable {
    typeid ::Messaging::Poller "IDL:omg.org/Messaging/Poller:3.1";
        readonly attribute Object          operation_target;
        readonly attribute string          operation_name;

        attribute ReplyHandler             associated_handler;
        readonly attribute boolean         is_from_poller;
    };
};
```

17.9.1 operation_target

The target of the asynchronous invocation is accessible from any Poller.

17.9.2 operation_name

The name of the operation that was invoked asynchronously is accessible from any Poller. The returned string is identical to the operation name from the target interface's **InterfaceDef**.

17.9.3 associated_handler

If the **associated_handler** is set to nil, the polling model is used to obtain the reply to the request. If it is non-nil, the associated **ReplyHandler** is invoked when a reply becomes available.

Switching between the callback and polling models is supported by this specification. The request must be made using the polling model, and thus a Poller is obtained. Through the attribute **associated_handler**, a **ReplyHandler** may be registered. When the reply is available, the associated **ReplyHandler** will be invoked just as if the callback model had been used to make the original request. By setting the attribute to nil, the **ReplyHandler** can be disassociated at any time to allow the client application to resume use of the Polling model. The Poller implementation is responsible for ensuring that in multi-threaded applications, access to the **associated_handler** is multi-thread safe.

17.9.4 is_from_poller

As described below, the type-specific pollers are queried to obtain the reply from an asynchronously invoked operation. If the reply is a system exception, it may be important for the client application to distinguish between an exception raised by the poll itself and an exception that is actually the reply for the asynchronous invocation. The **is_from_poller** attribute returns the value TRUE if and only if the poller itself has raised a system exception during the invocation of one of the type specific poller operations. If the exception raised from one of the type specific poller operations is the reply for the asynchronous operation, the value FALSE is returned. If the Poller has not yet returned a response to the client, the BAD_INV_ORDER system exception with standard minor code 22 is raised.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.10 Type-Specific Poller Mapping

The polling model requires usage of generated type-specific **Poller valuetype**s. A **valuetype** is used because all operations are locally implemented. The basic generated Poller encapsulates the operations for obtaining replies to an outstanding asynchronous request. A derived **PersistentPoller valuetype** also adds private state that allows the response to be obtained from a client other than the client that made the request. This private state is used by the **PersistentPoller** implementation in conjunction with the messaging-aware ORB.

17.10.1 Basic Type-Specific Poller

For each interface, the IDL compiler generates a type-specific **Poller** value. A **Poller** is created by the ORB for each asynchronous invocation that uses the polling model operations. The name of the basic type-specific **Poller** is **AMI_<ifaceName>Poller**, where **ifaceName** is the unqualified name of the interface for which the **Poller** is being generated. If the interface **ifaceName** derives from one or more IDL interfaces, then the **Poller** is derived from the corresponding **Poller** for each base interface, but if it does not, then it is derived from **Messaging::Polle**r. **Poller** valuetypes are declared abstract. If this name conflicts with definitions in the original IDL, additional **AMI**_ prefixes are prepended before **<ifaceName>** until a unique valuetype name is generated (such as "**AMI_AMI_FooPoller**" for interface **Foo**).

17.10.1.1 Poller operations for Interface operations

For each operation declared in the interface, a polling operation with the following signature is declared:

1. Return type void followed by
2. The name of the operation, followed by
3. A first parameter that is in unsigned long **ami_timeout** indicating for how many milliseconds this call should wait until the response becomes available. If this timeout expires before a reply is available, the operation raises the system exception CORBA:TIMEOUT with standard minor code 1. Any delegated invocations used by the implementation of this polling operation are subject to the single timeout parameter, which supersedes any ORB or thread-level timeout quality of service. Two specific values are of interest:
   - 0 - the call is a non-blocking poll, which raises the exception CORBA::NO_RESPONSE with the standard minor code 1 if the reply is not immediately available.
   - 232-1 - the maximum value for unsigned long indicates no timeout should be used. The poll will not return until the reply is available.

The remaining arguments, if any, are in order (all “out” parameters):

1. If the original operation has a return value, the type returned by the operation declared in IDL with parameter named **ami_return_val**.
2. Each inout/out type name and argument name as they were declared in IDL raises (<**exceptionList**>, **CORBA::WrongTransaction** where **exceptionList** contains the original operation raises exceptions, each exception from the original raises clause.
3. In addition, if the deferred synchronous model is being used:
   - the poll raises the CORBA::WrongTransaction user exception (if the request has an associated transaction context), and

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- the polling thread either has a null transaction context or a non-null transaction context that differs from that of the request.

When a polling call is made, the operation returns in one of the following ways:

1. With the out arguments set - the reply has been returned and future queries will raise the standard exception OBJECT_NOT_EXIST with standard minor code 5.
2. By raising the reply's exception - the reply has been returned and future queries will raise the standard exception OBJECT_NOT_EXIST with standard minor code 5. The base Poller's **is_from_poller** has a value of FALSE.
3. By raising a system exception or **CORBA::WrongTransaction** due to a failure in the polling operation. The base Poller's **is_from_poller** has a value of TRUE. Two specific exceptions are worth noting:
   - TIMEOUT - If a non-zero timeout value is specified, this system exception is raised with standard minor code 1 to indicate that the specified timeout has expired and the reply has not yet been returned.
   - NO_RESPONSE - If a timeout with value 0 is specified, this system exception is raised with standard minor code 1 to indicate that the reply is not available.

17.10.1.2 Poller operations for Interface attributes

For each attribute declared in the interface, a polling operation with the following signature is declared. Setter polling operations are only generated for attributes that are not declared readonly: return type void followed by the name of the generated operation, which to distinguish between the getter and setter operations for an attribute is given by (respectively):

- **get_<attributeName>**, where **attributeName** is the name of the interface's attribute, or
- **set_<attributeName>**, where **attributeName** is the name of the interface's attribute that was not declared readonly.

A first parameter that is in unsigned long **ami_timeout** indicating how many milliseconds this call should wait until the response becomes available. If this timeout expires before a reply is available, the operation raises the system exception CORBA::TIMEOUT with the standard minor code 1. Any delegated invocations used by the implementation of this polling operation are subject to the single timeout parameter, which supersedes any ORB or thread-level timeout quality of service. Two specific values are of interest:

- 0 - the call is a non-blocking poll, which raises the exception CORBA::NO_RESPONSE with the standard minor code 1 if the reply is not immediately available.
- 232-1 - the maximum value for **unsigned long** indicates no timeout should be used. The poll will not return until the reply is available.

For the getter operation only

An additional argument **out <attrType> ami_return_val** where **attrType** is the type of the attribute.

The **set** operation takes no additional arguments.

Raises (CORBA::WrongTransaction) - If the deferred synchronous model is being used, the poll raises the CORBA::WrongTransaction user exception if the request has an associated transaction context, and the polling thread either has a null transaction context or a non-null transaction context that differs from that of the request.

When a polling call is made, the operation returns in one of the following ways:

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- With the out arguments set - the reply has been returned and future queries will raise the standard exception OBJECT_NOT_EXIST with standard minor code 5.
- By raising the reply's exception - the reply has been returned and future queries will raise the standard exception OBJECT_NOT_EXIST with standard minor code 5. The base Poller's **is_from_poller** has a value of FALSE.
- By raising a system exception or **CORBA::WrongTransaction** due to a failure in the polling operation. The base Poller's **is_from_poller** has a value of TRUE. Two specific exceptions are worth noting:
  - TIMEOUT - If a non-zero timeout value is specified, this system exception is raised with standard minor code 1 to indicate that the specified timeout has expired and the reply has not yet been returned.
  - NO_RESPONSE - If a timeout with value 0 is specified, this system exception is raised with standard minor code 1 to indicate that the reply is not available.

17.10.2 Persistent Type-Specific Poller

When Time-Independent Invocations are made, the reply may be obtained by a different client than the one that made the original request. An instance of persistent poller is returned from such invocations. The **PersistentPoller** contains the state necessary to allow polling to be performed in a client distinct from the one that made the request. This state is used privately by the messaging-aware ORB and is not directly accessible to the application.

The generated **PersistentPoller valuetype** is derived from the basic one. It adds no methods, only one piece of private state. For an interface with the unqualified name **ifaceName** the following **PersistentPoller** is generated:

```
valuetype AMI_<ifaceName>PersistentPoller : AMI_<ifaceName>Poller
    private MessageRouting::PersistentRequest outstanding_request;
    private Object target;
    private string op_name;
};
```

Just as with any CORBA **valuetype** this **PersistentPoller** can be passed as an argument to IDL operations and a copy of the **Poller** will be instantiated local to the callee.

17.10.3 Example

The example IDL causes the generation of the following additional IDL when asynchronous polling operations are to be used. This IDL is "real" in that the **valuetype**s described here are normal CORBA **valuetype**s.

```
// AMI implied-IDL of type-specific Poller
// for original example IDL defined in  Section 17.5
abstract valuetype AMI_StockManagerPoller : Messaging::Poller {
    void get_stock_exchange_name(
            in unsigned long ami_timeout,
            out string ami_return_val)
        raises (CORBA::WrongTransaction);
    void set_stock_exchange_name(
            in unsigned long ami_timeout)
        raises (CORBA::WrongTransaction);
    void add_stock(
            in unsigned long ami_timeout,
            out boolean ami_return_val)
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        raises (CORBA::WrongTransaction);
    void edit_stock(
            in unsigned long ami_timeout)
        raises (InvalidStock, CORBA::WrongTransaction);
    void remove_stock(
            in unsigned long ami_timeout,
            out double quote)
        raises (InvalidStock, CORBA::WrongTransaction);
    void find_closest_symbol(
            in unsigned long ami_timeout,
            out boolean ami_return_val,
            out string symbol)
        raises (CORBA::WrongTransaction);
    void get_quote(
            in unsigned long ami_timeout,
            out double ami_return_val)
        raises (InvalidStock, CORBA::WrongTransaction);
};

valuetype AMI_StockManagerPersistentPoller : AMI_StockManagerPoller{
    private MessageRouting::PersistentRequest request;
    private Object target;
    private string op_name;
};
```

17.11 Example Programmer Usage

17.11.1 Example Programmer Usage (Examples Mapped to C++)

The following is an illustrative example of how the ideas from “II - Introduction” on page 429 and other sub clauses come together from the programmer’s point of view. It contains no new definitions; Example Programmer Usage on page 441 is solely meant to demonstrate an application use of Messaging. Since the example is implemented in C++, the expected C++ mapping of II - Introduction on page 429 implied-IDL is shown in Example Programmer Usage on page 441.

17.11.2 Client-Side C++ Example for the Asynchronous Method Signatures

This sub clause shows sample C++ that is generated from the implied-IDL of the previous sub clauses of II - Introduction on page 429. The C++ mapping specifies a generated interface class (stub) on which method invocations are translated into operation requests. It is this class on which the function signatures are generated from their operation declarations in IDL. It is in this class that the async functions signatures are also declared (and implemented). Using the IDL from the example in the previous sub clause the stub class **`StockManager`** is generated following the C++ mapping. The following notes apply to this sample generated C++ code:

- Only the generated synchronous and asynchronous method signatures are shown. Vendor-specific constructors, methods, and members are omitted.
- Although optional according to the IDL to C++ language mapping, method signatures are generated as virtual.
- Since optional according to the IDL to C++ language mapping, exception specifications are not included in generated methods.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
// Generated file:  stockmgr_c.hh (Filename is non-normative)

// C++ - StockManager  declaration
class StockManager  :  public virtual CORBA::Object
{
public:
// … all the other stuff.
// StockManager SYNCHRONOUS CALLS
virtual void stock_exchange_name(const char * attr);
virtual char * stock_exchange_name();
virtual CORBA::Boolean  add_stock(const char* symbol,CORBA::Double q);
virtual void edit_stock(const char* symbol, CORBA::Double q);
virtual void remove_stock(const char* symbol, CORBA::Double_out q);
virtual CORBA::Boolean find_closest_symbol(CORBA::String_out symbol);
virtual CORBA::Double get_quote(const char * symbol);

// ASYNCHRONOUS CALLBACK-MODEL CALLS
virtual void sendc_get_stock_exchange_name(
   AMI_StockManagerHandler_ptr ami_handler);
virtual void sendc_set_stock_exchange_name(
   AMI_StockManagerHandler_ptr ami_handler,
   const char* attr_stock_exchange_name);
virtual void sendc_addStock(
   AMI_StockManagerHandler_ptr ami_handler,
   const char* symbol, CORBA::Double q);
virtual void sendc_editStock(
   AMI_StockManagerHandler_ptr ami_handler,
   const char* symbol, CORBA::Double q);
virtual void sendc_removeStock(
   AMI_StockManagerHandler_ptr ami_handler,
   const char* symbol);
virtual void sendc_find_closest_symbol(
   AMI_StockManagerHandler_ptr ami_handler,
   const char * symbol);
virtual void sendc_get_quote(
   AMI_StockManagerHandler_ptr ami_handler,
   const char * symbol);

// ASYNCHRONOUS POLLING-MODEL CALLS
virtual AMI_StockManagerPoller* sendp_get_stock_exchange_name( );
virtual AMI_StockManagerPoller* sendp_set_stock_exchange_name(
   const char* attr_stock_exchange_name);
virtual AMI_StockManagerPoller* sendp_addStock(
   const char* symbol, CORBA::Double q);
virtual AMI_StockManagerPoller* sendp_editStock(
   const char* symbol, CORBA::Double q);
virtual AMI_StockManagerPoller* sendp_removeStock(
   const char* symbol);
virtual AMI_StockManagerPoller* sendp_find_closest_symbol(
   const char * symbol);
virtual AMI_StockManagerPoller* sendp_get_quote(
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
   const char * symbol);
};
```

17.11.3 Client-Side C++ Example of the Callback Model

17.11.3.1 C++ Example of Generated ReplyHandler

The **ReplyHandler** Servant class generated for the **StockManager** interface is:

```
// Generated  file:  stockmgr_s.hh (Filename is non-normative)
// C++ - AMI_StockManagerHandler  declaration
class POA_AMI_StockManagerHandler
   : public POA_Messaging::ReplyHandler
{
public:
// Programmer must implement the following pure virtuals:

// Mappings for attribute handling functions
virtual void get_stock_exchange_name(
   const char * ami_return_val) = 0;
virtual void get_stock_exchange_name_excep(
   Messaging::ExceptionHolder_ptr excep_holder) = 0;

virtual void set_stock_exchange_name() = 0;
virtual void set_stock_exchange_name_excep(
   Messaging::ExceptionHolder_ptr excep_holder) = 0;

// Mappings for the operation handling functions
virtual void add_stock(CORBA::Boolean ami_return_val) = 0;
virtual void add_stock_excep(
   Messaging::ExceptionHolder_ptr excep_holder) = 0;

virtual void edit_stock() = 0;virtual void edit_stock_excep(
   Messaging::ExceptionHolder_ptr excep_holder) = 0;

virtual void remove_stock(
   CORBA::Double quote) = 0;
virtual void remove_stock_excep(
   Messaging::ExceptionHolder_ptr excep_holder) = 0;

virtual void find_closest_symbol(
   CORBA::Boolean ami_return_val,
   const char * symbol) = 0;
virtual void find_closest_symbol_excep(
   Messaging::ExceptionHolder_ptr excep_holder) = 0;

virtual void get_quote(
   CORBA::Double d) = 0;
virtual void get_quote_excep(
   Messaging::ExceptionHolder_ptr excep_holder) = 0;
};
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The programmer must now derive from the generated handler and implement the pure virtual methods. The following points should be considered when implementing these handler-derived reply handlers:

- System and User exceptions are “raised” through invocations of the generated “_excep” operations. If a regular type-specific operation is invoked, the reply was not an exception.
- Any exception raised from a **ReplyHandler** method can only be visible to the messaging-aware ORB that is invoking that **ReplyHandler**. In most cases, this means that exceptions should never be raised. In the case of an Unshared Transaction, the **ReplyHandler** method may invoke **CosTransactions::Current::rollback_only** or **CosTransactions::coordinator::rollback_only** and then raise the CORBA::TRANSACTION_ROLLEDBACK system exception to roll back this attempted delivery of the reply.
- All heap-allocated storage associated with any of the arguments to the **ReplyHandler** methods may be owned by the ORB. If so, any data passed into the handler must be copied if the data is to be kept. This corresponds to the usual memory management rules for **in** arguments.

17.11.3.2 C++ Example of User -Implemented ReplyHandler

The following code is an example implementation of a user derived and implemented reply handler based on the generated reply handler from C++ Example of Generated ReplyHandler on page 443. The inherited methods, which were previously declared as pure virtual are declared here as virtual and are implemented as part of this class:

```
// File:  AsyncStockHandler.h
// C++ - Declaration in my own header
#include "stockmgr_s.hh"// Include filename non-normative

class AsyncStockHandler : public POA_AMI_StockManagerHandler
{
public:
AsyncStockHandler()  { }
virtual ~AsyncStockHandler() {}

// Mappings for attribute handling functions
virtual void get_stock_exchange_name(
   const char * ami_return_val);
virtual void get_stock_exchange_name_excep(
   Messaging::ExceptionHolder_ptr excep_holder);

virtual void set_stock_exchange_name();
virtual void set_stock_exchange_name_excep(
   Messaging::ExceptionHolder_ptr excep_holder);

// Mappings for the operation handling functions
virtual void add_stock(CORBA::Boolean ami_return_val);
virtual void add_stock_excep(
   Messaging::ExceptionHolder_ptr excep_holder);

virtual void edit_stock();
virtual void edit_stock_excep(
   Messaging::ExceptionHolder_ptr excep_holder);

virtual void remove_stock(
```


© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
   CORBA::Double quote);
virtual void remove_stock_excep(
   Messaging::ExceptionHolder_ptr excep_holder);

virtual void find_closest_symbol(
   CORBA::Boolean ami_return_val,
   const char * symbol);
virtual void find_closest_symbol_excep(
   Messaging::ExceptionHolder_ptr excep_holder);

virtual void get_quote(
   CORBA::Double d);
virtual void get_quote_excep(
   Messaging::ExceptionHolder_ptr excep_holder);
};
```

Each of these callback operations have implementations as in the following. Please note that for the sake of brevity, each pointer is not checked before it is used. This is intentional.

```
// AsyncStockHandler.cpp
#include <AsyncStockHandler.h>

void
AsyncStockHandler::get_stock_exchange_name(
const char * ami_return_val)
{
cout << "Exchange Name = " << ami_return_val << endl;
}
void
AsyncStockHandler::get_stock_exchange_name_excep(
   Messaging::ExceptionHolder_ptr excep_holder);
{
try {
   excep_holder->raise_exception();
}
catch (const CORBA::SystemException& e) {
   cout << "Get stock_exchange_name exception [" << e << "]" << endl;
}
}

void
AsyncStockHandler::set_stock_exchange_name()
{
// No data returned since this was the "set" of the attribute.
cout << "Set stock_exchange_name succeeded!" << endl;
}
void
AsyncStockHandler::set_stock_exchange_name_excep(
   Messaging::ExceptionHolder_ptr excep_holder)
{
try {
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
   excep_holder->raise_exception();
}
catch (const CORBA::SystemException& e) {
   cout << "Set stock_exchange_name exception [" << e << "]" << endl;
}
}

void
AsyncStockHandler::add_stock()
{
// No data returned but no exception either which is good news.
cout << "Stock was added!" << endl;
}
void
AsyncStockHandler::add_stock_excep(
   Messaging::ExceptionHolder_ptr excep_holder)
{
try {
   excep_holder->raise_exception();
}
catch (const CORBA::SystemException& e) {
   cout << "add_stock exception [" << e << "]" << endl;
}
}

void
AsyncStockHandler::edit_stock()
{
// No return data but no exception either which is good.
cout << "Stock was edited!" << endl;
}
void
AsyncStockHandler::edit_stock_excep(
   Messaging::ExceptionHolder_ptr excep_holder)
{
try {
   excep_holder->raise_exception();
}
catch (const CORBA::SystemException& e) {
   cout << "edit_stock System Exception exception [" << e << "]" <<
   endl;
}
catch (const InvalidStock& e) {
   cout << "edit_stock invalid symbol [" << e.sym << "]" << endl;
}
}

void
AsyncStockHandler::remove_stock(
CORBA::Double quote)
{
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
cout << "Stock Removed and quote = " << quote << endl;
}
void
AsyncStockHandler::remove_stock_excep(
   Messaging::ExceptionHolder_ptr excep_holder)
{
try {
   excep_holder->raise_exception();
}
catch (const CORBA::SystemException& e) {
   cout << "remove_stock System Exception exception [" << e << "]" <<
    endl;
}
catch (const InvalidStock& e) {
   cout << "remove_stock invalid symbol [" << e.sym << "]" << endl;
}
}

void
AsyncStockHandler::find_closest_symbol(
CORBA::Boolean ami_return_val,
const char* symbol)
{
if (ami_return_val)
   cout << "Closest stock  = " << symbol << endl;
else
   cout << "No closest stock could be found!" << endl;
}
void
AsyncStockHandler::find_closest_symbol_excep(
   Messaging::ExceptionHolder_ptr excep_holder)
{
try {
   excep_holder->raise_exception();
}
catch (const CORBA::SystemException& e) {
   cout << "find_closest_symbol exception [" << e << "]" << endl;
}
}

void
AsyncStockHandler::get_quote(CORBA::Double quote)
{
cout << "Quote  = " << quote << endl;
}
void
AsyncStockHandler::get_quote_excep(
   Messaging::ExceptionHolder_ptr excep_holder)
{
try {
   excep_holder->raise_exception();
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
}
catch (const CORBA::SystemException& e) {
   cout << "get_quote System Exception exception [" << e << "]" <<
    endl;
}
catch (const InvalidStock& e) {
   cout << "get_quote invalid symbol [" << e.sym << "]" << endl;
}
}
```

17.11.3.3 C++ Example of Callback Client Program

The following code shows how to set QoS at the ORB and object reference scopes (the two most common levels) and make asynchronous invocations using the user-implemented reply handler from the previous sub clause. Again, for the sake of brevity, checking for valid pointers and placing all of the CORBA calls in try blocks has been omitted.

```
// callback_client_main.cpp
#include <AsyncStockHandler.h>
int main(int argc, char ** argv)
{
// Initialize the ORB
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Initializing objRef for StockManager -- assumes IOR is passed
// on command-line
CORBA::Object_var obj = orb->string_to_object(argv[1]);
StockManager_var stockMgr = StockManager::_narrow(obj);

// Obtain the ORB's PolicyManager
CORBA::Object_var orbQosObj =
   orb->resolve_initial_references("ORBPolicyManager");
CORBA::PolicyManager_var orbQos =
CORBA::PolicyManager::_narrow(orbQosObj);

// Create and apply an ORB-wide Routed Delivery QoS
CORBA::Any routing_val;
Messaging::RoutingTypeRange routing;
routing.min = Messaging::FORWARD;
routing.max = Messaging::STORE_AND_FORWARD;
routing_val <<= routing;
CORBA::PolicyList orb_pols(1);
orb_pols.length(1);
orb_pols[(CORBA::ULong) 0] =
orb->create_policy(Messaging::ROUTING_POLICY_TYPE, routing_val);
orbQos->set_policy_overrides(orb_pols, CORBA::ADD_OVERRIDE);

// Create and apply an object-reference-specific Priority QoS
CORBA::Any priority_val;
Messaging::PriorityRange priority;
priority.min = 5;
priority.max = 15;
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
priority_val <<= priority;
CORBA::PolicyList obj_pols(1);
obj_pols.length(1);
obj_pols[(CORBA::ULong) 0] =
orb->create_policy(Messaging::REQUEST_PRIORITY_POLICY_TYPE,
priority_val);
stockMgr = stockMgr->set_policy_overrides(obj_pols);

// At this point QoS has been set and a protocol selected.

// Create an async handler for each async function.
// Note that the same handler instance could be used across the board
// if we wanted to only create a new Object Reference for each
// invocation and then correlate the timing data with each ObjectId
// ourselves.
//
// The following code assumes implicit activation of Servants with the
// RootPOA
AsyncStockHandler* handlerImpls[6];
for (int i = 0; i < 6; i++)
   handlerImpls[i] = new AsyncStockHandler();

AMI_StockManagerHandler_var handlerRefs[6];
for (int i=0; i < 6; i++)
   handlerRefs[i] = handlerImpls[i]._this();

// Async Attributes
stockMgr->sendc_set_stock_exchange_name(handlerRefs[0], "NSDQ");
stockMgr->sendc_get_stock_exchange_name(handlerRefs[1]);
// Async Operations
stockMgr->sendc_add_stock(handlerRefs[2], "ACME", 100.5);
stockMgr->sendc_edit_stock(handlerRefs[3], "ACME", 150.4);

// Notice no out param is passed.
stockMgr->sendc_remove_stock(handlerRefs[4], "ABC");

stockMgr->sendc_find_closest_symbol(handlerRefs[5], "ACMA");

// callbacks get invoked during other distributed requests and during
// eventloop processing.
// Assume that done is set by handler implementation when all replies
// have been received or request have timed out.while(!done)
   orb->perform_work();
return 0;
}
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.11.4 Client-Side C++ Example of the Polling Model

17.11.4.1 C++ Example of Generated Poller

The typed **Poller valuetype** class implementation is provided by the messaging-aware ORB. The generated C++ class has the following declaration:

```
// Generated file:  stockmgr_c.hh (Filename is non-normative)
class AMI_StockManagerPoller : public Messaging::Poller
{
public:
   virtual void get_stock_exchange_name(
                     CORBA::ULong ami_timeout,
               CORBA::String_out ami_return_val);


            virtual void set_stock_exchange_name(
                     CORBA::ULong ami_timeout),

   virtual void add_stock(
                     CORBA::ULong ami_timeout,
               CORBA::Boolean_out ami_return_val);

   virtual void edit_stock(
                     CORBA::ULong ami_timeout),

   virtual void remove_stock(
                     CORBA::ULong ami_timeout,
               CORBA::Double_out quote);

   virtual void find_closest_symbol(
                     CORBA::ULong ami_timeout,
               CORBA::Boolean_out ami_return_val,
               CORBA::String_out symbol);

   virtual void get_quote(
                     CORBA::ULong ami_timeout,
               CORBA::Double_out ami_return_val);
};
```

17.11.4.2 C++ Example of Polling Client Program

The following example client program demonstrates the use of the Polling model. The bulk of the program is exactly the same as the program demonstrated in C++ Example of Callback Client Program on page 448. Each invocation uses the polling “sendp_” in this program and the returned Pollers are then sequentially called to obtain the results. The following notes apply to this sample program:

- All polling calls are fully blocking (no timeouts are used).
- Since transactions are not used in this example, the polling program does not catch **`CORBA::WrongTransaction`** exceptions.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
// polling_client_main.cpp
#include <stockmgr_c.hh> // include filename is non-normative

int main(int argc, char ** argv)
{
   // Initialize the ORB
   CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

            // Initializing objRef for StockManager -- assumes IOR is passed
            // on command-line
               CORBA::Object_var obj = orb->string_to_object(argv[1]);
               StockManager_var stockMgr = StockManager::_narrow(obj);

            // Obtain the ORB's PolicyManager
               CORBA::Object_var orbQosObj =
                  orb->resolve_initial_references("ORBPolicyManager");
               CORBA::PolicyManager_var orbQos =
               CORBA::PolicyManager::_narrow(orbQosObj);

            // Create and apply an ORB-wide Routed Delivery QoS
               CORBA::Any routing_val;
               Messaging::RoutingTypeRange routing;
               routing.min = Messaging::FORWARD;
               routing.max = Messaging::STORE_AND_FORWARD;
               routing_val <<= routing;
               CORBA::PolicyList orb_pols(1);
               orb_pols.length(1);
               orb_pols[(CORBA::ULong) 0] =
                  orb->create_policy(Messaging::ROUTING_POLICY_TYPE, routing_val);
               orbQos->set_policy_overrides(orb_pols, CORBA::ADD_OVERRIDE);

            // Create and apply an object-reference-specific Priority QoS
               CORBA::Any priority_val;
               Messaging::PriorityRange priority;
               priority.min = 5;
               priority.max = 15;
               priority_val <<= priority;
               CORBA::PolicyList obj_pols(1);
               obj_pols.length(1);
               obj_pols[(CORBA::ULong) 0] =
                  orb->create_policy(Messaging::REQUEST_PRIORITY_POLICY_TYPE,
                        priority_val);
               stockMgr = stockMgr->set_policy_overrides(obj_pols);

            // At this point QoS has been set and a protocol selected.
               // Make each invocation and store the returned Pollers
               AMI_StockManagerPoller_var pollers[6];

            // Async Attributes
               pollers[0] = stockMgr->sendp_set_stock_exchange_name("NSDQ");
               pollers[1] = stockMgr->sendp_get_stock_exchange_name();
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
// Async Operations
   pollers[2] = stockMgr->sendp_add_stock("ACME", 100.5);
   pollers[3] = stockMgr->sendp_edit_stock("ACME", 150.4);

// Notice no out param is passed.
   pollers[4] = stockMgr->sendp_remove_stock("ABC");
   pollers[5] = stockMgr->sendp_find_closest_symbol("ACMA");

// Now obtain each result
   CORBA::ULong max_timeout = (CORBA::ULong) -1;
   pollers[0]->set_stock_exchange_name(max_timeout);
   cout << "Setting stock exchange name succeeded" << endl;

CORBA::String_var exchange_name;
   pollers[1]->get_stock_exchange_name(max_timeout,
    exchange_name.out());
   cout << "Obtained stock exchange name [" << exchange_name << "]" <<
    endl;

CORBA::Boolean stock_added;
   pollers[2]->add_stock(max_timeout, stock_added);
   if (stock_added)
      cout << "Stock added successfully" << endl;
   else
      cout << "Stock not added" << endl;

try {
      pollers[3]->edit_stock(max_timeout);
      cout << "Edited stock successfully" << endl;
   }
   catch (const CORBA::Exception& e) {
      cout << "Edit stock failure [" << e << "]" << endl;
   }

try {
      CORBA::Double quote;
      pollers[4]->remove_stock(max_timeout, quote);
      cout << "Removed stock successfully with quote [" << quote << "]"
          << endl;
   }
   catch (const CORBA::Exception& e) {
      cout << "Remove stock failure [" << e << "]" << endl;
   }

CORBA::Boolean closest_found;
   CORBA::String_var symbol;
   pollers[5]->find_closest_symbol(max_timeout, closest_found,
      symbol.out());
   if (closest_found)
      cout << "Found closest symbol [" << symbol << "]" << endl;
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            cout << "Exiting Polling Client" << endl;
            return 0;
         }
```

17.11.4.3 C++ Example of Using PollableSet in a Client Program

The following example client program demonstrates the use of the **PollableSet** and wait for multiple requests to finish. The program would be exactly the same as that of the previous sub clause, as far as the comment "// Now obtain each result."

In this example, after the **PollableSet::get_ready_pollable** indicates that a particular **Poller** has finished, the code makes the call on the type-specific poller in a non-blocking manner and doesn't bother checking for completion in the return value. Checking isn't necessary when only a single client is using the **Poller**, but it is the safe practice if multiple clients are waiting.

```
// Obtain results in any order. First set up the PollableSet.

CORBA::PollableSet_var poll_set = pollers[0]->create_pollable_set();

for (int i=0; i<6, i++) {
   poll_set->add_pollable(pollers[i]);
}

// repeat until all completions have been received
CORBA::ULong max_timeout = (CORBA::ULong) -1;
while (poll_set->number_left() > 0) {
   // wait for a completion
   CORBA::Pollable_var pollable = poll_set->get_ready_pollable(max_timeout);
   // the returned Pollable is ready to return its reply
   for (int j=0; j < 6; j++) {
         if (pollers[j] == pollable.in())
             break;
   }

            switch(j) {
   case 0:
         pollers[0]->set_stock_exchange_name(0UL);
         cout << "Setting stock exchange name succeeded"
            << endl;
         break;
   case 1:
         CORBA::String_var exchange_name;
         pollers[1]->get_stock_exchange_name(0UL, exchange_name.out());
         cout << "Obtained stock exchange name ["
             << exchange_name << "]" << endl;
         break;
   case 2:
         CORBA::Boolean stock_added;
         pollers[2]->add_stock(0UL, stock_added);
         if (stock_added)
            cout << "Stock added successfully" << endl;
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
         else
            cout << "Stock not added" << endl;
         break;
   case 3:
         try {
             pollers[3]->edit_stock(0UL);
            cout << "Edited stock successfully" << endl;
         }
         catch (const CORBA::Exception& e) {
             cout << "Edit stock failure [" << e << "]"
               << endl;
         }
         break;
   case 4:
         try {
            CORBA::Double quote;
            pollers[4]->remove_stock(0UL, quote);
            cout << "Removed stock successfully with quote ["
               << quote << "]" << endl;
         }
         catch (const CORBA::Exception& e) {
            cout << "Remove stock failure [" << e << "]"
               << endl;
         }
         break;
   case 5:
         CORBA::Boolean closest_found;
         CORBA::String_var symbol;
         pollers[5]->find_closest_symbol(0UL, closest_found, symbol.out());
         if (closest_found)
             cout << "Found closest symbol [" << symbol
                << "]" << endl;
         break;
   }
}
```

17.11.5 Server Side

The following example of the **server-side main()** assumes a C++ implementation of the **StockManager** interface called **StockManager_impl**.

```
#include <StockManagerImpl.h>  // Implementation header

int main(int argc, char ** argv)
{
// Initialize the ORB
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
// Obtain the POA
PortableServer::POA_var poa =
   orb->resolve_initial_references("RootPOA");
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
// Create a POA that supports Unshared transactions and processes
// queued requests in priority order
CORBA::Any policy_val;
CORBA::PolicyList pols(2);
pols.length(2);

policy_val <<= (Messaging::PRIORITY | Messaging::DEADLINE);
pols[(CORBA::ULong) 0] =
   orb->create_policy(Messaging::QUEUE_ORDER_POLICY_TYPE,
               policy_val);

policy_val <<= CosTransactions::Allows_either;
pols[(CORBA::ULong) 1] =
   orb->create_policy(CosTransactions::TRANSACTION_POLICY_TYPE,
              policy_val);
poa = poa->create_POA(
            "MessagingPOA",
            PortableServer::POAManager::_nil(),
            pols);

// Instantiate the servant.
StockManager_impl* stockMgr = new StockManager_impl("NYSE");
// register the servant for use.
PortableServer::ObjectId_var servantId =
   poa->activate_object(stockMgr);
orb->run();
return 0;
}
```

Section III - Message Routing Interoperability

17.12 Section III - Introduction

Asynchronous method invocation and time-independent delivery of requests and responses cannot be handled in a first-class manner within the synchronous dialog of the GIOP 1.1. The basic requirement for Messaging is that individual request and reply messages (and their components) can be discussed by routing agents. These agents, or *Routers*, explicitly pass messages between them and interact with clients and targets of asynchronous operations. This sub clause describes the interactions between a client and the first Router to handle its request, between successive Routers as the request is passed along the path to the target, and between the target and the Router that actually makes the request on behalf of the original client. This Router closest to the Target then turns the reply into a Request on a **ReplyHandler**, allowing the Reply to be routed using the same mechanism as the original request. The reply is finally delivered to an application's **ReplyHandler** or through an application's use of the Polling APIs.

NOTE: This Introduction specifies Routing interoperability for CORBA Messaging products. The information presented in this sub clause is not required for building applications that make Asynchronous operation invocations.

Throughout this Introduction a configuration is assumed in which the Client is separated from the Target by the Internet. Using this "most complex" scenario, all the details of the Routing procedure are exposed. To help understand this design, consider Figure 17.1.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

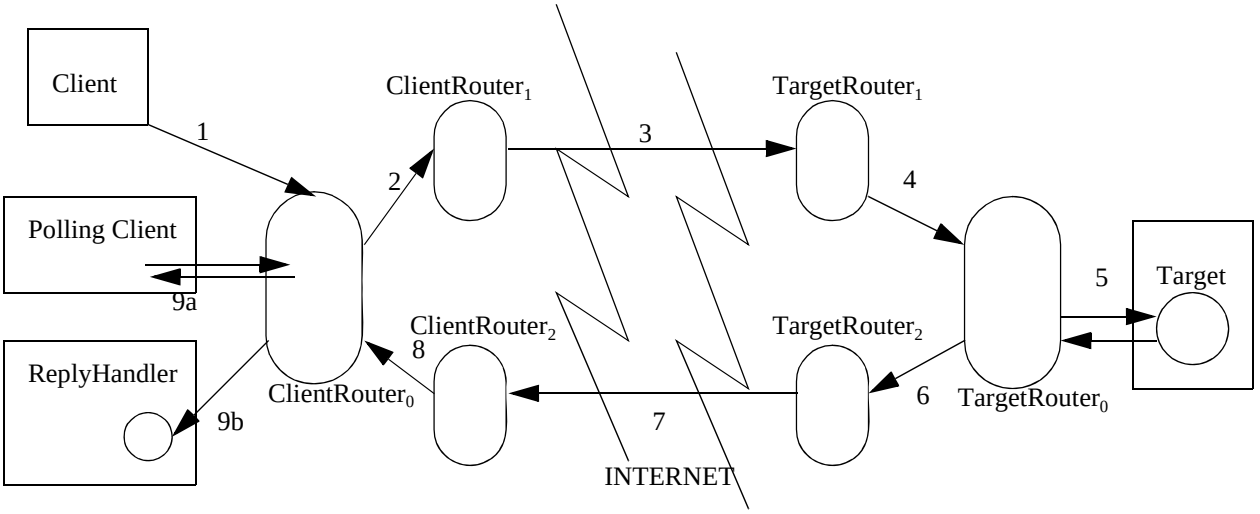


Figure 17.1 - Routing Interoperability Overview

17.13 Routing Object References

This specification is designed to support scenarios in which a target may be disconnected for a long period of time. It would be inefficient for a client's router to need to monitor the availability of all targets for which it holds outstanding requests. To make this scenario scalable, it is possible for the target to specify a more highly available temporary destination for its asynchronous requests. This destination is a Router, and the natural place for the target to specify this Router's location is within a component of the Target's IOR. For extensibility, this specification defines a **TaggedComponent** that contains a sequence of Router IORs.

```
module MessageRouting {
    const IOP::ComponentId TAG_MESSAGE_ROUTERS = 3;

    interface Router;
    typedef sequence<Router> RouterList;
};
```

A **TaggedComponent** containing Target routing hints is built by setting the tag member to **MessageRouting::TAG_MESSAGE_ROUTERS** and the **component_data** to a CDR encapsulation of a **MessageRouting::RouterList**. This component can appear in **TAG_INTERNET_IOP** and **TAG_MULTIPLE_COMPONENTS** profiles.

Routers are listed in this sequence in order from most highly available to least highly available. It is expected that the least highly available Router will be "closest" to the Target, whereas the most highly available Target Router will be "closest" to the Internet. For example, the target in the reference example of *III - Introduction on page 460* would have an IOR containing a **TAG_MESSAGE_ROUTERS** Component containing a sequence of two Router IORs. The first element in this sequence would be the reference of **TargetRouter1** and the second element would be the reference of **TargetRouter**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.14 Message Routing

The messaging Routers serve two main purposes:

- forward a message to another Router, and
- synchronously deliver a message to its intended target.

This sub clause explains the interfaces and mechanisms that support these two functions of Routers. The interfaces described here are not exposed to the application programmer in any way. They are intended entirely for use by Messaging vendors to support interoperability between messaging implementations.

The following IDL is used to route asynchronous requests and their corresponding replies:

```
// IDL
module Messaging {

    interface ReplyHandler { };
};

    module MessageRouting {

    typedef CORBA::OctetSeq BodyData;
    struct MessageBody {
        BodyData        body;
        boolean         byte_order;
    };

    struct RequestMessage {
        GIOP::Version giop_version;
        IOP::ServiceContextList service_contexts;
        octet response_flags;
        GIOP::RequestReserved reserved;
        IOP::ObjectKey object_key;
        string operation;
        MessageBody body;
    };

    enum ReplyDisposition { TYPED, UNTYPED };
    struct ReplyDestination {
        ReplyDisposition            handler_type;
        Messaging::ReplyHandler     handler;
    };

    interface Router;
    typedef sequence<Router> RouterList;
    struct RequestInfo {
        RouterList                  visited;
        RouterList                  to_visit;
        Object                      target;
        unsigned short              profile_index;
        ReplyDestination            reply_destination;
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        Messaging::PolicyValueSeq    selected_qos;
        RequestMessage               payload;
    };
    typedef sequence<RequestInfo> RequestInfoSeq;

    interface Router {
        void send_request(in RequestInfo req);
        void send_multiple_requests(in RequestInfoSeq reqSeq);
    };

    //
    // Polling-related interfaces
    //

    interface UntypedReplyHandler : Messaging::ReplyHandler {
        void reply(
                in string operation_name,
                in GIOP::ReplyStatusType reply_type,
                in MessageBody reply_body);
    };

    exception ReplyNotAvailable { };

    interface PersistentRequest {
        readonly attribute boolean reply_available;

        GIOP::ReplyStatusType get_reply(
                in boolean blocking,
                in unsigned long timeout,
                out MessageBody reply_body)
            raises (ReplyNotAvailable);

        attribute Messaging::ReplyHandler associated_handler;

        GIOP::ReplyStatusType get_reply_with_context(
                 in boolean blocking,
                 in unsigned long timeout,
                out MessageBody reply_body,
                 out IOP::ServiceContextList  service_contexts)
        raises (ReplyNotAvailable);
    };

    interface PersistentRequestRouter {
        PersistentRequest create_persistent_request(
                in unsigned short profile_index,
                in RouterList to_visit,
                in Object target,
                in CORBA::PolicyList current_qos,
                in RequestMessage payload);
    };
};
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.14.1 Structures

17.14.1.1 MessageBody

This structure is used to wrap the marshaled GIOP message data (either request arguments or reply data) to support repackaging as the request components around that data (such as service contexts or object key) change due to Routing. Since GIOP 1.2 Request and Reply Bodies are always aligned to an 8-octet boundary, it is necessary to keep track of the

- data and the length of that data as a sequence of octet, and
- the byte order with which that data was originally marshaled.

17.14.1.2 RequestMessage

This structure explicitly contains all the components of a GIOP request. When the target is actually invoked, its members are used to compose an actual GIOP request.

The **RequestMessage** has the following members:

- **iop_version** - the version of the GIOP that was used when the message was marshaled.
- **service_contexts** - the sequence of service contexts selected for this request. Routers must propagate all Service Contexts with unknown tags.
- **response_flags** - As explained further in the General Inter-ORB Protocol clause, the meaning of the two least significant bits is defined as:
  - the least significant bit (bit-0) indicates whether or not a response may be returned. If this bit is “1”, then the server-side ORB shall always send a **ReplyMessage**. If the bit-0 is “0”, no **ReplyMessage** will be sent. This replicates the function of the **response_expected** boolean in *CORBA*.
  - Bit-1 is considered if and only if bit-0 is “1.” If bit-1 is “0” the server sends a **ReplyMessage** before invoking the target. If bit-1 is “1,” the **ReplyMessage** is sent after the target has completed the invocation reserved.
- **object_key** - the opaque object key of the target. This may change if a GIOP object forwarding occurs for this request.
- **operation** - the operation name of the request being made.
- **body** - the CDR stream message payload and marshaling byte order for repackaging within a new GIOP request once the routed message can be synchronously invoked on the target.

17.14.1.3 ReplyDestination

This structure contains enough information for the response to be returned once the actual invocation has been made on the target.

- **handler_type** - Either **UNTYPED** or **TYPED** indicating which type of **ReplyHandler** is to receive the response. This flag is necessary to ensure that no **is_a** must be performed when the Target Router is ready to return the reply as described in Target Router on page 464.
- **handler** - an Object reference to the **ReplyHandler** that is the destination of the response.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.14.1.4 RequestInfo

This structure contains the information required for an intermediate Router to get a request closer to its target and for a target Router to invoke that request on its target.

- **visited** - the sequence of Routers through which the message has been sent already. Each router may add its reference to this sequence before forwarding the request to another Router. This sequence can be used by a Router to detect cycles in a network of Routers, but this is not a requirement step in the Routing protocol.
- **to_visit** - the suggested sequence of Routers to which the message should be sent if the target is not available. This sequence may be modified as the request is sent from Router to Router.
- **profile_index** - the index of the profile in the target IOR that is being used for this request. This is necessary so the target router can choose the correct object key when composing the final GIOP request.
- **target** - the full IOR of this message's target.
- **reply_destination** - a reference to the **ReplyHandler** for this request along with the disposition of that **ReplyHandler**. If the **handler_type** is **UNTYPED**, the destination is an untyped **ReplyHandler** (meaning that it was created when **create_persistent_request** was called and is implemented by the **ClientRouter**). If the **handler_type** is **TYPED**, the reply destination is a type-specific **ReplyHandler** implemented by an application using the callback model. If the reply destination is **nil**, no reply will be sent and the **handler_type** can be ignored.
- **selected_qos** - the list of QoS that was selected for the Routing of this message.
- **message** - the payload (arguments, return value, raised exception) for this message, including the byte order with which the message was originally marshaled.

17.14.2 Interfaces

17.14.2.1 ReplyHandler

The **ReplyHandler** interface is a base interface for all specific **ReplyHandler**s (either type-specific or Generic ones). It is used as the generic **reply_destination** argument when a request is sent to a Router:

17.14.2.2 Router

The **Router** interface is used to pass messages when a request cannot be synchronously invoked on its final target.

17.14.2.3 send_request

The Router is passed all the information necessary to either route the request toward the target by calling **send_request** on another Router, or to invoke the request on its final target.

17.14.2.4 send_multiple_requests

The Router is passed a sequence of **RequestInfo** structures, where each **RequestInfo** is a completely self-contained set of information allowing the Router to either route the request toward the target by calling **send_request** on another Router, or to invoke the request on its final target.

17.14.2.5 UntypedReplyHandler

This interface is the target of replies when the polling model is used.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.14.2.6 reply

The reply operation is invoked when the reply to a **PersistentRequest** becomes available. The operation is invoked with the following arguments:

- **operation_name** - The string name of the original request operation. This is necessary if the untyped reply must be turned into a callback on a typed **ReplyHandler** (as is the case if the polling client has switched models after making the request and associated a **ReplyHandler** with its Poller).
- **reply_type** - The status of the Reply (either **NO__EXCEPTION**, **USER_EXCEPTION**, or **SYSTEM_EXCEPTION**). **LOCATION_FORWARD** replies are not invoked on the **ReplyHandler**.
- **reply_body** - The marshaled data of the reply along with the byte order with which it was marshaled.

17.14.2.7 PersistentRequest

Instances of this interface are created by the Client Router for polling model invocations, and is queried to obtain the status of a request, including the reply's data if available.

17.14.2.8 readonly attribute reply_available

Returns the value **TRUE** if and only if the reply is currently available and has not yet been returned to some caller of **get_reply**. Returns the value **FALSE** if and only if the reply has not yet been returned to the ClientRouter. This attribute cannot be checked if the response has already been delivered to some caller of **get_reply**, as the **PersistentRequest** instance will have been deactivated at that time and the ORB will return the system exception OBJECT_NOT_EXIST on any subsequent invocations on that **PersistentRequest**.

17.14.2.9 get_reply and get_reply_with_context

The **get_reply** or **get_reply_with_context** operation is invoked to poll or block for a reply to a **PersistentRequest**. The operation returns the status of the reply (either **NO_EXCEPTION**, **USER_EXCEPTION**, or **SYSTEM_EXCEPTION**) or raises the ReplyNotAvailable exception if no reply is obtained before the specified timeout occurs. If the response is returned to the caller, the **PersistentRequest** is deactivated so that future invocations of **get_reply** or **get_reply_with_context** raise the system exception OBJECT_NOT_EXIST with standard minor code 5. The **get_reply** and **get_reply_with_context** operations takes the following arguments:

- **blocking** - if set, the operation does not return until either a reply can be returned or the **PersistentRequest** becomes invalid (due to an expired time-to-live).
- **timeout** - ignored if blocking is **TRUE**. Otherwise, the request blocks for the specified number of seconds or until a reply is available. If no reply becomes available after the specified timeout has expired, the ReplyNotAvailable exception is raised.
- **reply_body** - the data of the reply as originally marshaled by the target.

The **get_reply_with_context** operation has the following additional argument:

- service_contexts - the list of service contexts that is associated with the reply message, in the form of a **SeviceContextList**.

17.14.2.10 attribute associated_handler

The possibly **nil ReplyHandler** reference of the type-specific **ReplyHandler** registered to receive a callback reply for this request. This attribute is initially nil if the **PersistentRequest** was created for a polling client, and becomes non-nil if the client decides to switch from the polling model to the callback model.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.14.2.11 PersistentRequestRouter

This interface is used by the messaging-aware client ORB to create a request that can be queried to obtain its status and reply data (e.g., using the polling model).

17.14.2.12 create_persistent_request

When a **PersistentRequest** is created for a message, no reply destination is supplied. Instead, the **PersistentRequestRouter** establishes itself as the reply destination and returns to the caller a reference that has operations for obtaining the status and reply for the request. The operation that returns this new **PersistentRequest** takes the following arguments:

- **profile_index** - the index of the profile in the target IOR that is being used for this request. This is necessary so the target router can choose the correct object key when composing the final GIOP request.
- **to_visit** - the suggested sequence of Routers to which the message should be sent if the target is not available. This sequence may be modified as the request is sent from Router to Router.
- **target** - the full IOR of this message's target.
- **selected_qos** - the list of QoS that was selected for this message.
- **message** - the payload (arguments, return value, raised exception) for this message.

17.14.3 Routing Protocol

Processing of a time-independent invocation involves a series of roles played by various components of the distributed system. These roles include:

- the invoking client
- an initial request router
- intermediate request routers
- a target router
- the target object
- intermediate reply routers
- a final reply router
- the response-receiving client.

Not all of these distinct roles are necessarily involved in every invocation, and more than one role can be played by the same component of the distributed system. A router implementation is likely to be able to serve any of the router roles, and may even serve multiple roles for the same invocation, such as when the initial request router also serves as the target router with no intermediate request routers involved.

Routers can be collocated with client or server ORBs, or can be separate processes. Either way, routers must maintain persistent state with transactional semantics.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.14.3.1 Invoking Client

The client application makes an asynchronous invocation either by specifying a **ReplyHandler** object or by using the polling API.

Depending on QoS requirements, the client ORB may try to synchronously invoke the operation on the target object, using IIOP or some other synchronous protocol. This attempt will not be made if the client is part of an active transaction and the target has a **TransactionPolicy** of **Requires_unshared**.

If the target is unreachable via a synchronous protocol, the client ORB tries to find an initial router to use. If the target IOR has a **TAG_MESSAGE_ROUTERS** component, its list of routers may be tried, starting from the one closest to the target, which is the last in the list. If none of these are reachable, or there is no **TAG_MESSAGE_ROUTERS** component, then the client ORB's default router closest to the target may be chosen. The order in which the client ORB attempts to contact an initial router is not mandated by this specification. The client ORB may choose to send the request to any Router (such as its own closest Router in all cases) according to implementation-specific configuration. If the client application used the polling interface and a quality of service requiring the request to be persistent, the client ORB attempts to narrow the initial request router to a **PersistentRequestRouter**, and if this fails, a different router must be selected. If no router can be found meeting the required quality of service, the system exception CORBA::INV_POLICY is raised.

Once an initial request router is identified, the client ORB delivers the request to it by invoking **send_request** if a **ReplyHandler** was specified, or **create_persistent_request** if the polling API and persistent QoS was used. The client application's active transaction context, if any, is used for this invocation. Only service context information that is meaningful to the target in a time-independent invocation, such as **CodeSet**s (but not **TransactionContext**), is included in the **RequestMessage** argument to **send_request**. Future ORB service specifications must state whether their service contexts are to be considered end-to-end (and therefore included within the **RequestMessage**) or are only for a single hop (and therefore used by the ORB when invoking the initial router but not included with the **RequestMessage**).

An empty sequence is passed by the client ORB as the visited parameter. The list of routers from the target IOR's **TAG_MESSAGE_ROUTERS** component is used as the **to_visit** parameter. This list may have additional routers added to it by the client ORB depending on administration of the network of routers. If the callback model is being used, the type-specific **ReplyHandler** is passed as the **reply_destination**. If the request was originated using **create_persistent_request**, the untyped **ReplyHandler** is passed as the **reply_destination**. For the reply to be able to be delivered asynchronously, these **ReplyHandler** IORs must contain enough routing information (e.g., **TAG_MESSAGE_ROUTERS** component).

17.14.3.2 Initial Request Router

The initial request router's role depends on whether the **ReplyHandler** or polling API was used by the client.

If the client ORB passed the request message, along with a **ReplyHandler** reference, to the initial router using the **send_request** operation, the initial request router saves the request message to stable storage within the client application's transaction context, and then processes the request using the request routing algorithm described below.

If **create_persistent_request** was called, the initial request router must instantiate a **PersistentRequest** object and return its reference to the client ORB, which will return it to the client application. Until the response for the request is delivered to the client, or the request times out, such an initial request router must keep an association between the identity of this **PersistentRequest** object and the state of the request. When routing the request (as described below), this first router passes a **reply_destination**, which is an **UntypedReplyHandler** implemented by the first router itself. This **UntypedReplyHandler** may be created either before or after the **PersistentRequest** and request state is

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

committed to stable storage. After returning the **PersistentRequest** object and committing the request state to stable storage, all within the transaction context of the client application, the initial router processes the request using the routing algorithm described below. The routing process does not continue until the client's initial transaction has been committed.

17.14.3.3 Request Routing Algorithm

Any router that has received a request message and committed it to stable storage processes it in the same way. If it can invoke the operation directly on the target object, the router serves as the target router for the invocation, as described below. If not, it tries to deliver the request to another router closer to the target object. If it can't do either of these, it queues the request and tries again later, either after some period of time has elapsed, or in response to an announcement of availability from another router closer to the target as described in Router Administration on page 467.

A router typically picks another router closer to the target by selecting from the list of routers passed to it as the **to_visit** parameter to either **send_request** or **create_persistent_request**. Routers later in the list are given preference as being closer to synchronous connection with the target. The next router can also be selected from some set of known Routers based on an implementation-specific configuration. If QoS attributes of the request message require persistence of requests, a transaction is first initiated. Then **send_request** is called on the selected router. The **to_visit** parameter is formed by removing the callee from the **to_visit** list received with the original request. Any routers further from the target than the callee (earlier in the **to_visit** list) are also removed. The **target**, **reply_destination**, **selected_qos**, and **message** parameters are copied from the received request. After invoking **send_request**, the router removes the request message from its stable storage, and commits the transaction if it initiated one.

A router must ensure that exactly-once semantics are preserved. If delivering a request message results in an exception with a **CompletionStatus** of **COMPLETED_NO**, or in a transaction being aborted, it can retry. Since any invocation can raise a system exception, all exception replies with a completion status other than **COMPLETED_NO** must be reported back to the client via the reply message.

17.14.3.4 Intermediate Request Router

An intermediate router is simply a router that accepts a request message via **send_request** from one router and then, eventually, delivers it to another router, again using **send_request**. The **send_multiple_requests** operation may also be used to allow batching of requests between Routers. The intermediate routers may take a request's **QueueOrderPolicy** (if present) into account when prioritizing the delivery of requests to destination routers, but is not required to do so.

17.14.3.5 Target Router

The target router for an invocation is a router that accepts a request message, delivers it to the target object, and, if a response is expected, routes the target's reply back to the client. The target router may have to queue the request message before the invocation and/or may have to queue the response message after the invocation.

The target router may be collocated with the target, or may deliver the request to the target via a synchronous GIOP-based protocol. The target router is responsible for processing any **LOCATION_FORWARD** replies that may be generated in making the invocation on the target, so only NO_EXCEPTION, USER_EXCEPTION, or SYSTEM_EXCEPTION replies are routed back to the client. When making the synchronous GIOP request on the target, the **TargetRouter** must marshal its request with the same byte order with which the original message body was marshaled. This byte order is recorded in the **MessageBody** structure. No Router is expected to remarshal the request body with a new byte order.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

If persistence of requests is required, the target router ensures that the request message is removed from stable storage and the reply message is committed to stable storage within the scope of a single transaction. If the target object's IOR indicates that it supports time-independent transactions (through a **TransactionPolicy** of **Allows_unshared**, **Allows_either**, **Requires_unshared**, or **Requires_either**), then that same transaction context is propagated to the server application. Otherwise no transaction context is propagated to the target when the request is invoked.

When guaranteed delivery is required, there may be one, two, or three distinct transactions involved in the target router's processing of the invocation. The target router receives the request message within the context of a transaction initiated by a previous router or possibly the client ORB. If the target is accessible at that time, the operation can be invoked on the target and the reply message either stored or sent back toward the reply destination using the transaction context within which the request was received. If the target is not accessible, the request message is committed to stable storage and queued for later delivery to the target under a second transaction. When the target operation is invoked and its reply is received, the target router may deliver the reply to another router, or possibly to the client ORB. The router may deliver the reply in the same transaction as it invoked the operation, or the router may commit the reply to stable storage and later deliver it in yet another transaction. The completion of the transaction in which the **TargetRouter** actually delivers the request to the target is governed by the following cases:

- A NO_EXCEPTION reply is returned and the transaction commits. This committed reply is the one that will be returned to the client. Since the reply committed, the request is no longer waiting in some queue pending delivery.

A **NO_EXCEPTION** reply is returned but the transaction raises TRANSACTION_ROLLEDBACK with standard minor code 4 upon commit. In this case the router must ensure that the request not be considered pending delivery anymore (logically the request must be removed from some queue), and that a suitable reply be generated so that the client knows that the target's transaction rolled back. The router starts a new transaction in which it removes the request from its "to be delivered" queue and generates a reply with the system exception TRANSACTION_ROLLEDBACK with standard minor code 4. This reply is then committed as the reply for the request.

A user or system exception is returned. The Router should rollback the transaction so no work has been done in the target server. There are two subcases here:

- the target was unreachable. In this case, since the transaction has rolled back, the request is still waiting in the Router's queue of pending requests. The retry policy is used to determine when next to attempt delivery.
- the target was reachable but an exception was raised. As in the **TRANSACTION_ROLLEDBACK** case above, the Router starts a new transaction to remove the request from the queue of pending requests, and commits the exception reply that it received from the target as the reply for this operation.

If the request has a **QueueOrderPolicy** associated with it, the target router is responsible for making invocations in the proper order. Depending on the Ordering requested (e.g., **PRIORITY**, **TEMPORAL**), the appropriate request is selected for delivery. Note that end-to-end ordering guarantees cannot be made when client and target are decoupled, so this ordering is really only a guideline. If multiple threads are used in the router for request delivery, it is certainly possible for delivery of requests to be out of order. The specification of **QueueOrderPolicy** does not require a router or server ORB to limit its use of threads in delivering requests.

Regardless of how many transactions, if any, are used, the target router must route the reply back to the reply destination if and only if the **response_expected** flag was set to a non-zero value in the **RequestMessage**. The reply can take one of two forms depending on whether the **reply_destination** is a type-specific **ReplyHandler** (the client uses the Callback model) or if the **reply_destination** is an **UntypedReplyHandler** (a **PersistentRequest** was created such as when the client used the Polling model).

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NOTE: The type-specific reply handlers and the **UntypedReplyHandler** are both derived from the common base **ReplyHandler** interface, but there is no other inheritance relationship between the **UntypedReplyHandler** and the type-specific reply handlers.

Regardless of destination, the new reply must be marshaled with the same byte order used by the target when the reply was originally marshaled. The Target Router is not expected to remarshal the reply body.

17.14.3.6 Replying to a Type-specific ReplyHandler

If the client originally supplied a type-specific **ReplyHandler**, the reply must be converted into a typed request invocation on the **ReplyHandler**. The Target Router determines this by verifying that the **handler_type** disposition of the **reply_destination** argument has the value **TYPED**. The format of the generated request depends on the **reply_status**:

- **NO_EXCEPTION** - the generated reply operation has the same operation name as the request. Its **RequestBody** is exactly the same as the marshaled **ReplyBody** from the target's GIOP reply.
- **SYSTEM_EXCEPTION** or **USER_EXCEPTION** - the generated reply operation has the same name as the request operation, with the string **_excep** appended. The single argument to this request is the **Messaging::ExceptionHolder** valuetype.

A reply with status **LOCATION_FORWARD** is handled as described below.

17.14.3.7 Replying to an UntypedReplyHandler

If the client originally created a **PersistentRequest** (such as by using the Polling model), the reply must be converted into the generic request operation supported by the **UntypedReplyHandler** interface. The Target Router determines this by verifying that the **handler_type** disposition of the **reply_destination** argument has the value **UNTYPED**. The generated reply operation has the name "reply" and takes as arguments the original operation name, the **reply_status** (**NO_EXCEPTION**, **SYSTEM_EXCEPTION**, or **USER_EXCEPTION**) and a sequence of octet containing the reply data. The length is set to the size of the marshaled **ReplyBody** and the data is the marshaled body itself.

17.14.3.8 Handling of Service Contexts

When a **TargetRouter** receives a Reply, it generates a request on some **ReplyTarget** as described previously in this sub clause. If the Reply contains service contexts, the **TargetRouter** must decide whether or not these contexts are to be used in its request on the **ReplyTarget**. End-to-end service contexts, such as the **CodeSets** context, are propagated to the **ReplyTarget**. Single-hop service contexts, such as the **TransactionService** context, are consumed by the **TargetRouter**. Unknown service contexts are propagated from the reply to the generated request on the **ReplyTarget**.

17.14.3.9 Handling LOCATION_FORWARD Replies

When a **TargetRouter** receives a Reply with status **LOCATION_FORWARD**, it must either use the returned reference as the new target for the request, or must return the new reference to the **ReplyTarget**. The Messaging protocol requires that the **TargetRouter** continue processing the request by either directly invoking the new target or routing the request toward the new target as has been described thus far.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.14.3.10 Routing of Replies

As described above, the GIOP reply is turned into a request message targeted to the original **reply_destination**. Since this reply is now a request, it may be sent to its destination using the message routing protocol described in this sub clause. For example, if the **ReplyHandler**'s reference contains Routing information, the **TargetRouter** may invoke the new request using some Router's **send_request** operation. In this case, the specified routing protocol should be followed for this new request, with the **response_expected** flags all set to 0 and the **reply_destination** set to nil.

17.14.3.11 UntypedReplyHandler

When an **UntypedReplyHandler**'s reply operation is invoked, several things may happen. The specific correlation of a Router's **UntypedReplyHandler** with the **PersistentRequests** it supports is not visible to this interoperability layer, but at a high level one of the following occurs:

- A type-specific **ReplyHandler** has been associated with the corresponding **PersistentRequest**. If a callback has been registered for this reply (the **associated_handler** is non-nil), the type-specific callback operation may be invoked directly as described in Replying to a Type-specific ReplyHandler on page 466. For persistent delivery of replies, the Router starts a transaction in which the reply is delivered. Once the client returns, the Router commits and the reply is deleted. As with any transactional request, the application's **ReplyHandler** implementation may choose to invoke **CosTransactions::Current::rollback_only** or **CosTransactions::coordinator::rollback_only** and then raise the CORBA::TRANSACTION_ROLLEDBACK system exception if it wishes to rollback the Router's transaction.
- A **PersistentRequest::get_reply** is pending for this request. The reply data may be immediately returned to the waiting client. The reply is returned within the client's transaction context and when that transaction is committed the reply is deleted.
- The reply data may be saved to stable storage (for guaranteed delivery this is made durable when the sending Router commits the transaction in which the reply has been delivered) or recorded in-process (if the reply is not guaranteed). The **UntypedReplyHandler::reply** then returns. The reply is obtained by a client at a later time.

17.15 Router Administration

One basic function of a Router is to forward a request to another Router, which is "closer" to the eventual target of a client's original request. In terms of the relationship between these two routers, the first Router can be thought of as the "source Router," and the second can be called the "destination Router." In the case where the network is partitioned or the destination Router has temporarily or permanently become unavailable, the source Router will be unable to forward its message. When this occurs, the Router must determine when and how to retry the request to the destination Router.

To enable scalable networks of routers, a **RouterAdmin** interface has been specified. The interface is defined mainly for the purpose of avoiding the non-scaling scenario where a source Router has no choice but to consume network resources by continuously "pinging" its destination Router.

This problem is analogous to the one faced by the target router when attempting delivery of the request to the message's target. Therefore, the mechanism specified here generically supports registrations of destination routers as well as actual target object references.

```
module MessageRouting {

    typedef short RegistrationState;
    const RegistrationState NOT_REGISTERED   = 0;
    const RegistrationState ACTIVE           = 1;
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
const RegistrationState SUSPENDED           = 2;

exception InvalidState{
    RegistrationState registration_state;
};

valuetype RetryPolicy supports CORBA::Policy { };

const CORBA::PolicyType IMMEDIATE_SUSPEND_POLICY_TYPE = 50;
valuetype ImmediateSuspend : RetryPolicy { };

const CORBA::PolicyType UNLIMITED_PING_POLICY_TYPE = 51;
valuetype UnlimitedPing : RetryPolicy {
    public short max_backoffs;
    public float backoff_factor;
    public unsigned long base_interval_seconds;
};

const CORBA::PolicyType LIMITED_PING_POLICY_TYPE = 52;
valuetype LimitedPing : UnlimitedPing {
    public unsigned long interval_limit;
};

const CORBA::PolicyType DECAY_POLICY_TYPE = 53;
valuetype DecayPolicy supports CORBA::Policy {
    public unsigned long decay_seconds;
};

const CORBA::PolicyType RESUME_POLICY_TYPE = 54;
valuetype ResumePolicy supports CORBA::Policy {
    public unsigned long resume_seconds;
};

interface RouterAdmin {
    void register_destination(
            in Object dest,
            in boolean is_router,
            in RetryPolicy retry,
            in DecayPolicy decay);

    void suspend_destination(
            in Object dest,
            in ResumePolicy resumption)
                raises (InvalidState);

    void resume_destination(
            in Object dest)
        raises (InvalidState);

    void unregister_destination(
            in Object dest)
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            raises (InvalidState);
    };

    interface Router {
        readonly attribute RouterAdmin admin;
    };
};
```

When a request arrives at a Router (source router) that must either be delivered directly to a target, or be forwarded on via another Router (destination router), that source router attempts to send the message. If the message send fails, the source router needs to decide when to retry the send. The following use of the **RouterAdmin** is intended for router-to-router administration:

1. A source router gets a request that should be sent to a destination router. Since the source router has no registration for that destination router, it attempts to send the message.

2. Upon receipt of the message, the destination router realizes that it has never registered back with the source router and calls back to the source router's **RouterAdmin** (independent of the processing of the message - this is purely an optional administrative request to avoid poor routing behavior in the future). By calling back to the **RouterAdmin**, the destination router registers itself with its desired retry policy and decay policy for future messages. On subsequent messages, the destination router knows that it has already registered and need perform no administrative processing at this step.

3. At some time, the destination router knows it is being separated from the network. This case is termed “graceful disconnection.”
   - The destination router notifies the source router that the registration should be suspended.
   - Upon subsequent requests, the source router consults its list of registrations. Since the destination router is currently **SUSPENDED**, no send is attempted (depending on the **ResumePolicy** at the time of suspension).
   - At some later time, the destination router becomes reconnected. It resumes its registration and can now receive stored (and later) messages.

4. At some time, the destination router becomes disconnected without any advanced warning (it may not know that it is disconnected). This case is termed “unexpected disconnection.”
   - Upon subsequent requests, the source router consults its list of registrations. Since the destination router is currently **ACTIVE**, a send is attempted. When the send fails, the source router follows its **RetryPolicy** and keeps pinging until the **RetryPolicy** indicates the registration should be suspended (immediately if the **RetryPolicy** is **ImmediateSuspend** or never if the **RetryPolicy** is **UnlimitedPing**).
   - At some time, the destination router becomes reconnected. If the source router discovers this due to pinging, the pending requests can now be delivered. If the source router has **SUSPENDED** the registration or is in the midst of the interval between pings when the destination router re-registers itself, the registration can immediately be set to an **ACTIVE** state and pending requests can be sent to the destination router.

The “target router” is the one that synchronously delivers requests to the target. The **RouterAdmin** is also used for the administration of policies that determine when this target router will actually attempt to deliver its request. A target’s use of this interface is very similar to the way it is used for router-to-router administration described above. The analogous scenarios are re-described here for clarity:

1. An object instance is activated with support for TII. Since the target is now ready to receive requests, it is registered with some router’s **RouterAdmin** with the target’s desired retry policy and decay policy. Typically, a reference to

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

this router will also be contained in a **MessageRouting::TAG_MESSAGE_ROUTERS** component of the target's object reference.

2. A router gets a request that it can deliver directly to the target (therefore this router is considered a "target router"). Since the target router has a registration for that object, it attempts to invoke the request.

3. At some time, the target knows it is being separated from the network. This case is termed "graceful disconnection."
   - The target notifies the target router that the registration should be suspended.
   - Upon subsequent requests, the target router consults its list of registrations. Since the target is currently **SUSPENDED**, no invocation is attempted (depending on the **ResumePolicy** at the time of suspension).
   - At some later time, the target becomes reconnected. It resumes its registration and can now receive stored (and later) requests.

4. At some time, the target becomes disconnected without any advanced warning (it may not know that it is disconnected). This case is termed "unexpected disconnection."
   - Upon subsequent requests, the target router consults its list of registrations. Since the target is currently **ACTIVE**, an invocation is attempted. When this invocation fails, the target router follows its **RetryPolicy** and keeps pinging until the **RetryPolicy** indicates the registration should be suspended (immediately if the **RetryPolicy** is **ImmediateSuspend** or never if the **RetryPolicy** is **UnlimitedPing**).
   - At some time, the target once again becomes available. If the target router discovers this due to pinging, the pending requests can now be delivered. If the target router has **SUSPENDED** the registration or is in the midst of the interval between pings when the target re-registers itself, the registration can immediately be set to an **ACTIVE** state and pending requests can be invoked on the target.

17.15.1 Constants

17.15.1.1 typedef short RegistrationState

The **RegistrationState** indicates the current status of a registration for a particular destination (a router or a target). The possible values are:

- **NOT_REGISTERED** - The given destination is not registered with this **RouterAdmin**.
- **ACTIVE** - The given destination is currently registered with this **RouterAdmin** and is not in the suspended state.
- **SUSPENDED** - The given destination is currently registered with this **RouterAdmin** and has been set to the Suspended state.

17.15.2 Exceptions

17.15.2.1 exception InvalidState

The attempted operation attempts to affect a registration, which is not in a state with a valid transition to the new state dictated by the operation. The State member contains the current status of the router or target for which the operation was attempted:

- **Suspend** was attempted on a router/target either not registered or already suspended.
- **Resume** was attempted on a router/target either not registered or already active.
- **Unregister** was attempted on a router/target not registered.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.15.3 Valuetypes

17.15.3.1 RetryPolicy

This **valuetype** is the abstract base from which all retry policies are derived.

17.15.3.2 ImmediateSuspend

The registered router is placed in the **SUSPENDED** state as soon as a message send fails.

17.15.3.3 UnlimitedPing

This **valuetype** is used to parameterize a pinging behavior:

- **backoff_factor** - If **max_backoffs** is non-zero, the **backoff_factor** is the number by which the current interval between failed send attempts is multiplied to determine the interval before the next send should be attempted. For example, a **backoff_factor** of 2 will cause the interval to double between each failed attempt.
- **base_interval_seconds** - The base number of seconds between retries.
- **max_backoffs** - If zero, the same interval is used between each retry (constant interval pinging). If non-zero, the interval between retries is multiplied by the **backoff_factor** after each failed send attempt until **max_backoffs** failed attempts have been made. Once **max_backoffs** have been performed, retry attempts are made at the constant rate of the last interval used. Otherwise, the same interval is used between each retry (linear pinging).

17.15.3.4 LimitedPing

This **valuetype** is used to parameterize a pinging behavior that should be stopped after a specified number of attempts. It derives from **UnlimitedPing** and adds the following state:

- **interval_limit** - The number of attempts before the pinging should be stopped.

17.15.3.5 DecayPolicy

This **valuetype** indicates how long a given registration is valid. If the **decay_seconds** are set to the value zero, the registered destination router will only be unregistered with an invocation of **unregister_router**. Otherwise, the registered destination router will be unregistered after the specified timeout has elapsed.

17.15.3.6 ResumePolicy

This **valuetype** indicates when a suspended registration should be resumed. If the **resume_seconds** are set to the value zero, the registered destination will only become active once explicitly resumed. Otherwise, the suspended destination will be resumed after the specified timeout has passed.

17.15.4 Interfaces

17.15.4.1 RouterAdmin

The **RouterAdmin** interface provides the operations for supporting scalable connection and disconnection between source routers and their destination routers and targets.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17.15.4.2 register_destination

A registration is added for the specified target with the given policies. If the registration is marked as **is_router**, the destination will receive messages via the Router interface as described in Intermediate Request Router on page 464. Otherwise, the registration is assumed to be for a target, in which case delivery is made as described in Target Router on page 464.

17.15.4.3 suspend_destination

The specified registration is suspended. If that target is not in an **ACTIVE** state, an InvalidState exception is raised. The suspended destination will be returned to the **ACTIVE** state if an explicit **resume_destination** or **register_destination** operation is performed for that destination. If the **resume_policy** allows for **TimedResume**, this transition will occur in, at most, the specified amount of time (e.g., if an explicit resumption doesn't happen first).

17.15.4.4 resume_destination

Resume the suspended destination. An InvalidState exception is raised if the destination is not in the **SUSPENDED** state.

17.15.4.5 unregister_destination

Unregister the specified destination. An InvalidState exception is raised if the target is not registered.

17.16 CORBA Messaging IDL

17.16.1 Messaging Module

The following module has been added by CORBA Messaging:

```
// IDL
// File: Messaging.idl
#ifndef _MESSAGING_IDL_
#define _MESSAGING_IDL_

import ::CORBA;
import ::IOP;
import ::TimeBase;
module Messaging {
    typeprefix Messaging "omg.org";

    //
    // Messaging Quality of Service
    //

    typedef short RebindMode;
    const RebindMode TRANSPARENT =              0;
    const RebindMode NO_REBIND =                1;
    const RebindMode NO_RECONNECT =             2;

    typedef short SyncScope;
    const SyncScope SYNC_NONE =                 0;
    const SyncScope SYNC_WITH_TRANSPORT =       1;
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
const SyncScope SYNC_WITH_SERVER =          2;
const SyncScope SYNC_WITH_TARGET =          3;

typedef short RoutingType;
const RoutingType ROUTE_NONE =              0;
const RoutingType ROUTE_FORWARD =           1;
const RoutingType ROUTE_STORE_AND_FORWARD = 2;

typedef short Priority;

typedef unsigned short Ordering;
const Ordering ORDER_ANY =              0x01;
const Ordering ORDER_TEMPORAL =         0x02;
const Ordering ORDER_PRIORITY =         0x04;
const Ordering ORDER_DEADLINE =         0x08;

//
// Locally-Constrained Policy Objects
//

// Rebind Policy (default = TRANSPARENT)
const CORBA::PolicyType REBIND_POLICY_TYPE = 23;
interface RebindPolicy : CORBA::Policy {
    readonly attribute RebindMode     rebind_mode;
};

// Synchronization Policy (default = SYNC_WITH_TRANSPORT)
const CORBA::PolicyType SYNC_SCOPE_POLICY_TYPE = 24;
interface SyncScopePolicy : CORBA::Policy {
    readonly attribute SyncScope      synchronization;
};

// Priority Policies
const CORBA::PolicyType REQUEST_PRIORITY_POLICY_TYPE = 25;
struct PriorityRange {
    Priority min;
    Priority max;
};
interface RequestPriorityPolicy : CORBA::Policy {
    readonly attribute PriorityRange        priority_range;
};
const CORBA::PolicyType REPLY_PRIORITY_POLICY_TYPE = 26;
interface ReplyPriorityPolicy : CORBA::Policy {
    readonly attribute PriorityRange        priority_range;
};

// Timeout Policies
const CORBA::PolicyType REQUEST_START_TIME_POLICY_TYPE = 27;
interface RequestStartTimePolicy : CORBA::Policy {
    readonly attribute TimeBase::UtcT start_time;
};
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
const CORBA::PolicyType REQUEST_END_TIME_POLICY_TYPE = 28;
interface RequestEndTimePolicy : CORBA::Policy {
    readonly attribute TimeBase::UtcT end_time;
};

const CORBA::PolicyType REPLY_START_TIME_POLICY_TYPE = 29;
interface ReplyStartTimePolicy : CORBA::Policy {
    readonly attribute TimeBase::UtcT start_time;
};
const CORBA::PolicyType REPLY_END_TIME_POLICY_TYPE = 30;
interface ReplyEndTimePolicy : CORBA::Policy {


    readonly attribute TimeBase::UtcT end_time;
};

const CORBA::PolicyType RELATIVE_REQ_TIMEOUT_POLICY_TYPE = 31;
interface RelativeRequestTimeoutPolicy : CORBA::Policy {
    readonly attribute TimeBase::TimeT relative_expiry;
};

const CORBA::PolicyType RELATIVE_RT_TIMEOUT_POLICY_TYPE = 32;
interface RelativeRoundtripTimeoutPolicy : CORBA::Policy {
    readonly attribute TimeBase::TimeT relative_expiry;
};

const CORBA::PolicyType ROUTING_POLICY_TYPE = 33;
struct RoutingTypeRange {
    RoutingType min;
    RoutingType max;
};
interface RoutingPolicy : CORBA::Policy {
    readonly attribute RoutingTypeRange      routing_range;
};

const CORBA::PolicyType MAX_HOPS_POLICY_TYPE = 34;
interface MaxHopsPolicy : CORBA::Policy {
    readonly attribute unsigned short max_hops;
};

// Router Delivery-ordering Policy (default = ORDER_TEMPORAL)
const CORBA::PolicyType QUEUE_ORDER_POLICY_TYPE = 35;
interface QueueOrderPolicy : CORBA::Policy {
    readonly attribute Ordering              allowed_orders;
};

//
// Propagation of QoS Policies
//

typedef CORBA::OctetSeq PolicyData;
```


© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    struct PolicyValue {
        CORBA::PolicyType           ptype;
        PolicyData                  pvalue;
    };
    typedef sequence<PolicyValue> PolicyValueSeq;

    //
    // Exception Delivery in the Callback Model
    //

    typedef CORBA::OctetSeq MarshaledException;
    native UserExceptionBase;
    valuetype ExceptionHolder {
        void raise_exception() raises (UserExceptionBase);
        void raise_exception_with_list(
            in CORBA::ExceptionList exc_list)
            in Dynamic::ExceptionList exc_list)
        raises (UserExceptionBase);
        private boolean  is_system_exception;
        private boolean  byte_order;
        private MarshaledException marshaled_exception;
    };

    //
    // Base interface for the Callback model
    //

    interface ReplyHandler { };

    //
    // Base value for the Polling model
    //
    abstract valuetype Poller : CORBA::Pollable {
    typeid ::Messaging::Poller "IDL:omg.org/Messaging/Poller:3.1";
        readonly attribute Object       operation_target;
        readonly attribute string       operation_name;

        attribute ReplyHandler          associated_handler;
        readonly attribute boolean      is_from_poller;
    };
};
#endif
```

17.16.2 MessageRouting Module

The following module has been added for the CORBA Messaging Interoperable Routing Protocol. These definitions are only required for interoperable support of Time-Independent Invocations:

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
// IDL
// File: MessageRouting.idl
#ifndef _MESSAGE_ROUTING_IDL_
#define _MESSAGE_ROUTING_IDL_

import ::CORBA;
import::Dynamic;
import ::GIOP;
import ::IOP;
import ::Messaging;
module MessageRouting {
    typeprefix MessageRouting “omg.org”;

    //
    // Basic Routing Interoperability
    //

    interface Router;
    interface RouterAdmin;
    typedef sequence<Router> RouterList;

    typedef CORBA::OctetSeq BodyData;

    struct MessageBody {
        BodyData        body;
        boolean         byte_order;
    };

    struct RequestMessage {
        GIOP::Version giop_version;
        IOP::ServiceContextList service_contexts;
        octet response_flags;
        GIOP::RequestReserved reserved;
        IOP::ObjectKey object_key;
        string operation;
        MessageBody body;
    };

    enum ReplyDisposition { TYPED, UNTYPED };
    struct ReplyDestination {
        ReplyDisposition            handler_type;
        Messaging::ReplyHandler     handler;
    };

    struct RequestInfo {
        RouterList                  visited;
        RouterList                  to_visit;
        Object                      target;
        unsigned short              profile_index;
        ReplyDestination            reply_destination;
        Messaging::PolicyValueSeq   selected_qos;
```


© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    RequestMessage                  payload;
};
typedef sequence<RequestInfo> RequestInfoSeq;

interface Router {
    void send_request(in RequestInfo req);
    void send_multiple_requests(in RequestInfoSeq reqSeq);

    readonly attribute RouterAdmin admin;
};

//
// Polling-related interfaces
//

interface UntypedReplyHandler : Messaging::ReplyHandler {
    void reply(
            in string operation_name,
            in GIOP::ReplyStatusType reply_type,


            in MessageBody reply_body);
};

exception ReplyNotAvailable { };

interface PersistentRequest {
    readonly attribute boolean reply_available;

    GIOP::ReplyStatusType get_reply(
            in boolean blocking,
            in unsigned long timeout,
            out MessageBody reply_body)
        raises (ReplyNotAvailable);

    attribute Messaging::ReplyHandler associated_handler;

    GIOP::ReplyStatusType get_reply_with_context(
            in boolean blocking,
            in unsigned long timeout,
            out MessageBody reply_body,
            out IOP::ServiceContextList  service_contexts)
    raises (ReplyNotAvailable);
};

interface PersistentRequestRouter {
    PersistentRequest create_persistent_request(
            in unsigned short profile_index,
            in RouterList to_visit,
            in Object target,
            in CORBA::PolicyList current_qos,
```

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
            in RequestMessage payload);
    };

    //
    // Router Administration
    //

    typedef short RegistrationState;
    const RegistrationState NOT_REGISTERED   = 0;
    const RegistrationState ACTIVE           = 1;
    const RegistrationState SUSPENDED        = 2;

    exception InvalidState{
        RegistrationState registration_state;
    };

    valuetype RetryPolicy supports CORBA::Policy { };

    const CORBA::PolicyType IMMEDIATE_SUSPEND_POLICY_TYPE = 50;
    valuetype ImmediateSuspend : RetryPolicy { };

    const CORBA::PolicyType UNLIMITED_PING_POLICY_TYPE = 51;
    valuetype UnlimitedPing : RetryPolicy {
        public short max_backoffs;
        public float backoff_factor;
        public unsigned long base_interval_seconds;
    };

    const CORBA::PolicyType LIMITED_PING_POLICY_TYPE = 52;
    valuetype LimitedPing : UnlimitedPing {
        public unsigned long interval_limit;
    };

    const CORBA::PolicyType DECAY_POLICY_TYPE = 53;
    valuetype DecayPolicy supports CORBA::Policy {
        public unsigned long decay_seconds;
    };

    const CORBA::PolicyType RESUME_POLICY_TYPE = 54;
    valuetype ResumePolicy supports CORBA::Policy {
        public unsigned long resume_seconds;
    };

    interface RouterAdmin {
        void register_destination(
                in Object dest,
                in boolean is_router,
                in RetryPolicy retry,
                in DecayPolicy decay);

        void suspend_destination(
```


© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
                in Object dest,
                in ResumePolicy resumption)
            raises (InvalidState);

        void resume_destination(
                in Object dest)
            raises (InvalidState);

        void unregister_destination(
                in Object dest)
            raises (InvalidState);
    };
};
#endif
```

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex A for Clause 17
Overall Design Rationale

(normative)

A.1 QoS Abstract Model Design

This Annex describes each of the components in the Quality of Service (QoS) abstract model and their relationships. The specification defines a framework within which current QoS levels are queried and overridden. This framework is intended to be of use for CORBAServices specifiers, as well as for future revisions of CORBA. The Messaging-specific QoS are defined in terms of this framework.

NOTE: The QoS definitions specified in this specification are applied to both synchronous as well as asynchronous invocations.

A.2 Model Components

The QoS framework abstract model consists of the following components:

- **Policy** - The base interface from which all QoS objects derive.
- **PolicyList** - A sequence of Policy objects.
- **PolicyManager** - An interface with operations for querying and overriding QoS **Policy** settings.
  - Mechanisms for obtaining **Policy** override management operations at each relevant application scope:
  - The ORB's **PolicyManager** is obtained through invoking **ORB::resolve_initial_references** with the **ObjectId** "ORBPolicyManager".
  - A **CORBA::PolicyCurrent** derived from **CORBA::Current** is used for managing the thread's QoS Policies. A reference to this interface is obtained through an invocation of **ORB::resolve_initial_references** with the **ObjectId** "PolicyCurrent."
  - Accessor operations on **CORBA::Object** allow querying and overriding of QoS at the object reference scope.
  - The application of QoS on a Portable Object Adapter is done through the currently existing mechanism of passing a **PolicyList** to the **POA::create_POA** operation.
- Mechanisms for transporting Policy values as part of interoperable object references and within requests:
  - **TAG_POLICIES** - A Profile Component containing the sequence of QoS policies exported with the object reference by an object adapter.
  - **INVOCATION_POLICIES** - A Service Context containing a sequence of QoS policies in effect for the invocation.

The Messaging QoS abstract model consists of a number of **CORBA::Policy**-derived interfaces:

- Client-side Policies are applied to control the behavior of requests and replies. These include Priority, RequestEndTime, and Queueing QoS.
- Server-side Policies are applied to control the default behavior of invocations on a target. These include QueueOrder and Transactionality QoS.

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A.2.1 Component Relationships

Programmers set QoS at various levels of scope by creating a Policy-derived Messaging QoS Policy and selecting the interface for the particular scope. It is anticipated that the following is the standard use-case scenario:

- A POA is created with a certain set of QoS. When object references are created by that POA, the required and supported QoS are encoded in that object reference. Such an object reference is then exported for use by a client.
- Within a client, the ORB's **PolicyManager** interface is obtained to set QoS for the entire ORB (for the entire process when only one ORB is present) either programmatically, or administratively. The Policies set here are valid for all invocations in the process. A programmer-constructed **PolicyList** is used with this interface to actually set the QoS.
- Within that same client, the **CORBA::PolicyCurrent** is obtained to set QoS for all invocations in the current thread. This interface is derived from the **PolicyManager** interface, which can be used to change the QoS for each invocation. A programmer-constructed **PolicyList** is used with this interface to actually set the QoS.
- Within that same client, the object reference is obtained and an invocation of its **get_client_policy** operation queries the most specific QoS overrides. A programmer-constructed **PolicyList** may be passed to the Object's **set_policy_overrides** operation to obtain a new Object reference with revised QoS. Setting the QoS here applies to all invocations using the new Object reference and supersedes (if possible) those set at the ORB and thread (Current) scopes. The current set of overrides can be validated by calling the Object's pseudo-operation **validate_connection**, which will attempt to locate a target for the object reference if no target has yet been located. At this time, any Policy overrides placed at the Object, Thread or ORB scope will be reconciled with the QoS Policies established for that object reference when it was created by the POA. The current *effective* **Policy** can then be queried by invoking **get_policy**, which returns the **Policy** value that is in effect.
- Unseen by the application, the ORB (including the protocol engine) modifies its internal behavior in order to realize the quality of service indicated by the client through the first three steps. See the description of the protocol abstract design in Message Routing Abstract Model Design on page 488.

A.2.2 Component Design

Design decisions were made with respect to the following components of the QoS framework:

- Each QoS is an interface derived from **CORBA::Policy**. The design trade-offs focused on ease of application interface for setting specific QoS values, extensibility for new QoS types and values, and compactness so the QoS values can be represented efficiently in Service Contexts and IOR Profile Components. Several alternatives were considered as the basic type for each QoS entity before the decision was made to use the **Policy** interface:
  - **CORBA::NamedValue** - A pair of **string** and **any** were considered mainly due to the flexibility afforded by using an any to represent QoS values. This design was discounted due to the untyped nature of the **any** and the application development and execution costs of inserting typed data into and extracting typed data from values of type **any**. Furthermore, the presence of a full typecode within an any makes the size of such pairs too large for inclusion in compact Service Contexts and Profile Components.
  - Stateful CORBA **valuetype** - Although the **valuetype** does present a typed interface to the application program, including **valuetype**s in Service Contexts and IOR Profile Components is too expensive due to the presence of full repository identifier information when the **valuetype** is marshaled. Furthermore, there are issues associated with potential truncation of such QoS **valuetype**s when passed as formal arguments of their base type.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- Interfaces derived from **CORBA::Policy** and compact representation. In the model chosen by this specification, the QoS values are accessible through locality-constrained interfaces. Derivation from **CORBA::Policy** allows reuse of existing interfaces and operations for policy management. When certain QoS values must be marshaled in a Service Context or an IOR Profile Component, the most compact format was chosen. The type of QoS **Policy** represented is indicated by a structure containing the integral **PolicyType** and a **sequence** of **octet** holding the values for that policy.

- A generic factory for creating QoS Policies. In the *POA* specification within *CORBA*, each POA **Policy** is created through an operation on the POA itself. Although this presents a convenient typed interface for the creation of **Policy** objects, it causes serious problems when new POA Policies are introduced. To fit with the current model, operations would have to be added to the POA interface for every new type of POA **Policy**. To address this potential administrative nightmare, this specification introduces a new ORB operation **create_policy**. Rather than introducing typed operations for creating all of the Messaging QoS Policies discussed in this specification, the generic factory operation is used.

- A **RebindPolicy** client-side QoS **Policy** to ensure deterministic effective QoS. In *CORBA*, *transparent rebinding* of an object reference may take place during any invocation. Rebinding is defined here to mean changing the client-visible QoS as a result of replacing the IOR Profile used by a client's object reference with a new IOR Profile. Transparent rebinding is defined as when this happens without notice to the client application. Typically, this happens within GIOP through the use of location forwarding. The default **RebindPolicy** (and the only *CORBA* behavior) supports this transparent rebind. For an application with rigorous quality of service requirements, such transparent rebinding can cause problems. For instance, unexpected errors may occur if the application sets its QoS Policies appropriately for an object reference, and then the ORB transparently changes the application's assumptions about that reference by obtaining a new IOR. The **RebindPolicy** has been added so that applications can prevent the ORB from silently changing the IOR Profile (and therefore the server-side QoS) that have been assumed. A more rigorous value of this **Policy** even precludes the ORB from silently closing and opening connections (when IIOP is being used, for example). The specific requirements demanded by an application dictate which level of **RebindPolicy** is necessary.

A.3 AMI/TII Abstract Model Design

This sub clause describes each of the components in the Asynchronous Method Invocation /Time-Independent Invocation (AMI/TII) abstract model and the relationships between them.

The model supported by Messaging is a specialization of the general object model described in the OMA guide. All of the elements of the CORBA object model are present in the model described here. Some of the names of existing components are defined more precisely than they are in the CORBA object model. In addition, this specification adds some new components to support Messaging.

Some of the components described here have been borrowed from other specifications, which in some cases have yet to be ratified. Where this occurs, it is clearly noted.

A.3.1 Asynchronous Method Invocation Components

The abstract model for AMI/TII supported by Messaging adds the following client-side components:

- **ReplyHandler** - A **ReplyHandler** is an Object that encapsulates the functionality for handling an asynchronous reply. It is used for callback model reply handling.

- **Poller** - A Poller is a **valuetype** used by clients to obtain replies to asynchronous invocations. The Poller provides a type-specific wrapping through which a Reply is obtained.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- Asynchronous Method Invocation (AMI) - A remote method invocation that returns immediately and whose reply is handled by a **ReplyHandler**-derived class implemented by the programmer, or whose reply is obtained through a Poller **valuetype**.

A.3.2 Time-Independent Invocation Components

The abstract model for AMI/TII supported by Messaging adds the following components to support interoperability of Time-Independent Invocations:

- **PersistentRequest** - A **PersistentRequest** is an Object that encapsulates an outstanding request. It supports operations for asynchronous operations (including polling or blocking until the reply comes). The **PersistentRequest** is not a locality constrained object (as opposed to the **CORBA::Request**).
- **Persistent ReplyHandler** - A **ReplyHandler** whose Object reference is created by a POA with a **PERSISTENT LifeSpan** Policy. The Persistent **ReplyHandler** may be implemented by a process other than the one that issued the request.
- **PersistentPoller** - A Poller with state including a **PersistentRequest** reference. The **PersistentPoller** may be used by a process other than the one that issued the request.
- Time-Independent Invocation (TII) - A time-independent invocation is an AMI request whose reply may outlive the client process. This is addressed via the persistent **ReplyHandler** and Poller mechanisms.
- Router - A software routing agent that is used when the target objects (either the target of the request or the target of the reply) are not available.
- Interoperable Routing Protocol -- An interoperable routing protocol built in terms of GIOP that provides a higher level of Quality of Service with respect to message routing and delivery than is currently supported by IIOP. These extensions allow out-of-the-box interoperability and define interfaces for MOM product plug-ins to support CORBA Messaging with value-added QoS services that the particular MOM vendor brings to the market.

A.3.3 Component Relationships

Figure 17.2 denotes an abstract view of the general Messaging architecture and is not meant to imply any particular implementation.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

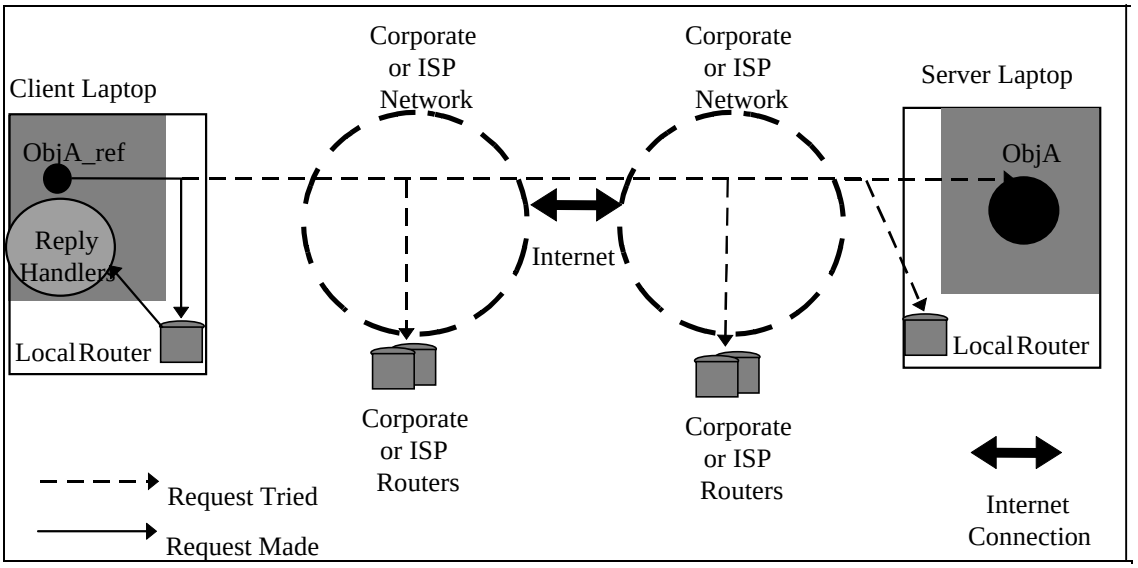


Figure 17.2 - TII: No direct connection possible

Figure 17.2 depicts the most general scenario in which a client application residing on a laptop wishes to make an asynchronous method invocation on an object in a server residing on another laptop. Each laptop typically connects to its own corporate or ISP network. Each of these networks has some set of Request/Reply Routers installed that are meant to be highly available and reliable. These Routers provide store-and-forward capabilities.

In Figure 17.2 neither client nor server laptops are currently connected to their respective networks. In this scenario, the client application makes its requests using the Time-Independent Invocation model. The dashed arrows indicate that the client always tries to make the invocation on the target object or the Request/Reply Router closest to the target. Since the client is not connected, it makes the invocation on the local router (indicated by the solid arrow).

Figure 17.3 depicts an asynchronous invocation in that the replies to the client invoke an operation on a callback object called a **ReplyHandler**. In general, the client may passivate himself, or may die while the request is outstanding. If a persistent delivery quality of service had been specified (with a long enough time-out period) the reply may be delivered when the **ReplyHandler** instance becomes available again. All object adapter features including process activation, Adapter activation and servant activation can be used in ensuring delivery of the reply to a persistent **ReplyHandler**.

Again, Figure 17.2 is meant to depict the most general case.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

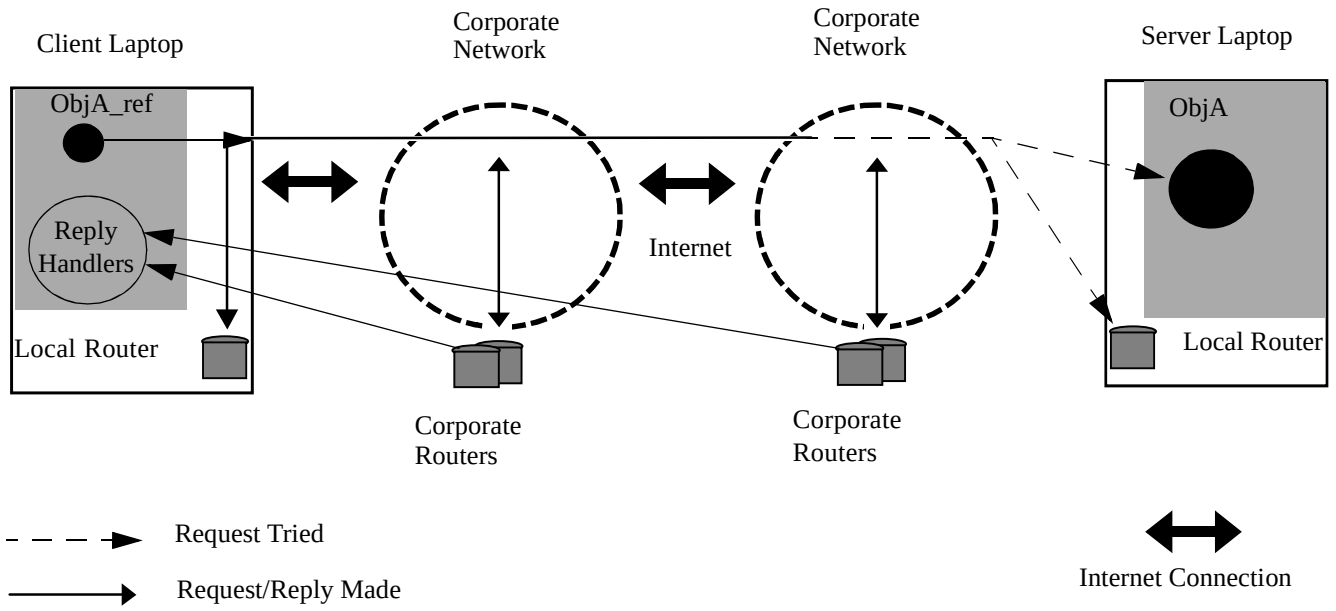


Figure 17.3 - TII: Target not available synchronously

Figure 17.3 illustrates the case where the client laptop gains an Internet connection to its corporate network. In this scenario, the Routers that are accessible exchange requests and replies always first trying to contact the target and then sending to the accessible **Router** closest to the target. In Figure 17.3, the server laptop is not accessible so the routers exchange information. Notice that Corporate Routers may have replies to invoke on the client's set of ReplyHandlers now that the client is reachable. Also, recognize that since the client laptop is now connected, there may be requests and replies for other targets, which are not currently running on the Client Laptop and so are cached in the Client Laptop's Local **Router**.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

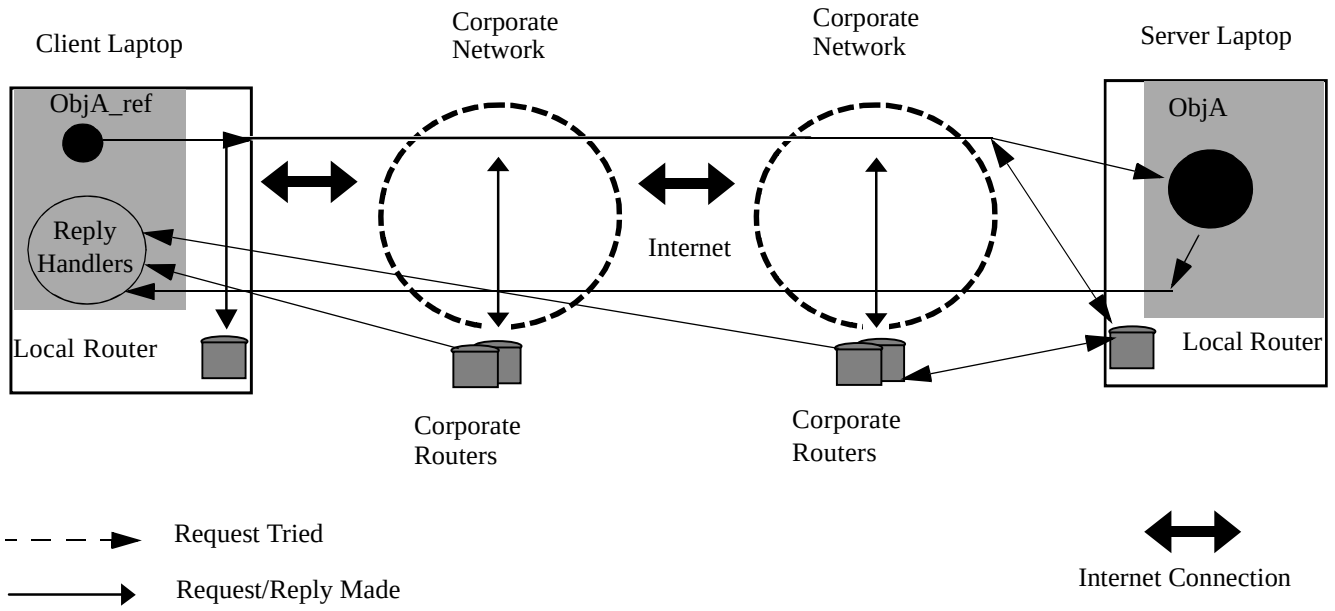


Figure 17.4 - Full connectivity available

Finally, Figure 17.4 represents full connectivity. Notice that all of the Request/Reply Routers exchange information to get previously-queued requests/replies closer to their target objects. Since there is full connectivity between the two applications, the client's async invocations can be made on the target object directly and the replies can be sent directly back to make the appropriate invocation on the **ReplyHandler** object.

If the client application has requested queued delivery, a Router is used even in the case depicted in Figure 17.4. Despite the availability of the target, the client ORB sends the request to a Router, which can queue the request prior to attempting the synchronous invocation on the target. As an optimization that limits the request to needing only a single network hop, this Router may be local to the target, but it is still a Router with all the usual responsibilities.

Notice also that since the Server Laptop is connected its Request/Reply Router exchanges information for applications that may or may not be running.

A.3.4 Callback Model Detailed Design

Several characteristics of the Callback programming model are worth extra attention:

- The **ReplyHandler** is a CORBA object that receives the reply to an AMI. The programmer writes the implementation for a type-specific **ReplyHandler**. A client obtains an object reference for this **ReplyHandler** and passes it as part of the asynchronous method invocation. When the server completes the request, its reply is delivered as an invocation on the **ReplyHandler** object. This invocation is made on the **ReplyHandler** using the normal POA techniques of servant and object activation. As a result, the callback operation may be handled in a different programming context than that in which the original request was made.
- Exception replies require special handling in the Callback model. Since the **ReplyHandler** implements an IDL interface, all arguments passed to its operations must be defined in IDL as well. However, exceptions cannot be passed as arguments to operations; exceptions can only be raised as part of a reply. To solve this problem, an **ExceptionHolder valuetype** is created to encapsulate the identity and contents of the exception that was raised. An

© ISO/IEC 2008 - All rights reserved

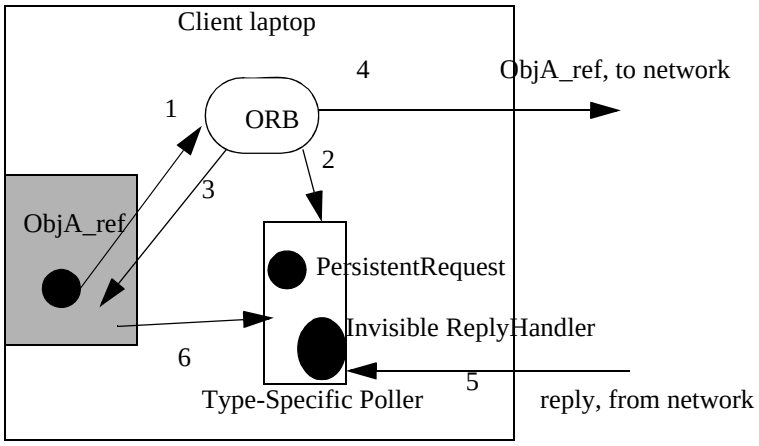
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

instance of this **ExceptionHolder** is passed as the argument to the **ReplyHandler** operation that indicates an exception was raised by the target. In addition to its exception state, the **ExceptionHolder** also has operations that raise the returned exception, so the **ReplyHandler** implementation can have the returned exception re-raised within its own context.

A.3.5 Poller/PersistentRequest Detailed Design

In the Polling model, the routing relationships are a superset of those seen in the Callback model. The differences in this model appear at both the beginning and end of the request/reply cycle. For Polling, the client application does not establish a Callback **ReplyHandler**. The events that occur when Polling are pictured in Example 17-1. The steps are as follows:

1. The client invokes the "sendp" variation of the target object's operation.
2. The ORB creates a **PersistentRequest** object and associates a reference to it with an invisible **ReplyHandler** that is wrapped in a type-specific Poller value.
3. The ORB returns this Poller to the client.
4. The ORB then proceeds as if the invocation were done with the invisible **ReplyHandler** and sends its request into the network.
5. At the very end, the invisible **ReplyHandler** receives the response and waits for a poll.
6. When the computing context holding the type-specific Poller asks for a response, the Poller obtains the response from the invisible **ReplyHandler** and delivers that response to the caller.



Example 17-1. Sequence of Steps in Polling

A client uses the **Poller** in a similar fashion as in the DII deferred synchronous model. The programmer can at any time choose to check whether or not the reply has arrived and deal with it in the current programming context. The user may also ask a Poller to block until the reply has arrived. The **PersistentRequest** reference is not visible to the client application, but is specified to enable interoperability between Messaging products.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

When a Time-Independent Invocation has been made, it is possible to poll for the reply in a client different from the one that made the initial request. An application takes advantage of this by passing the Poller from the client that made the request to the client that intends to poll for the reply (presumably by way of an Object instance that is collocated with the latter client). Since this Poller is implemented through the use of a **PersistentRequest** object implemented by the Messaging layer, that **PersistentRequest** must be accessible to whichever client uses that Poller. When the TII is used, it is possible for the polling client to obtain the reply after the original invoking client no longer exists. Since the **PersistentRequest** must be implemented in a server that is accessible to the Polling client, that **PersistentRequest** must be external to the original invoking client. A common design might be to have the **PersistentRequest** in this case be implemented by a corporate Router accessible to the invoking client as well as to the client that intends to poll for the response. The creation of **PersistentRequest** objects is discussed in detail in the Section 17.12, Section III - Introduction, on page 455.

In addition to being able to query the status of an individual Poller, the client can use the **PollableSet** interface to ask about the status of several pollers, as well as the status of any deferred synchronous requests. The client can query to find out if any of a particular set has completed or it can block until one of the set completes.

Note on CORBA AMI Support

Asynchrony is addressed in several places in *CORBA*. These items are taken into consideration by this specification and are modified in the following ways:

- oneway operations - Operations can be defined in IDL to be oneway. Such operations are by their very nature asynchronous, in that no reply is ever received from a oneway operation and no synchrony can be assumed between the client and the target. However, the definition of oneway in the *CORBA* specification does not guarantee a deterministic, portable behavior between compliant ORB products. To address this issue, the CORBA Messaging specification introduces a QoS Policy that makes the behavior of oneway operations deterministic. Note that this new Policy addresses the behavior of oneway operations regardless of the use of the new Polling and Callback stubs introduced by this specification.
- DII Deferred Synchronous - Deferred synchronous invocations are supported in *CORBA* only when the DII is used. The CORBA::Request pseudo-interface is enhanced by this specification with the additions of TII and the Callback model.

Note on Asynchrony and Narrowing of Object References

- Many programming languages map IDL interfaces to programming constructs that support inheritance. In those language mappings (such as C++ and Java) that provide a mechanism for narrowing an Object reference of a base interface to a more derived interface, the act of narrowing may require the full type hierarchy of the target. In this case, the implementation of narrow must either contact an interface repository or the target itself to determine whether or not it is safe to narrow the client's object reference. This requirement is not acceptable when a client is expecting only asynchronous communication with the target. Therefore, for the appropriate languages this specification adds an unchecked narrow operation to the IDL mappings for interface. This unchecked narrow always returns a stub of the requested type without checking that the target really implements that interface. If a client narrows the target to an unsupported interface type, invoking the unsupported operations will raise the system exception CORBA::BAD_OPERATION with standard minor code 2.

A.4 Message Routing Abstract Model Design

This sub clause describes each of the components of the Message Routing abstract model and their relationships.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A.4.1 Model Components

By and large the components of the message routing protocol are the same as those of GIOP. The differences come with respect to two issues:

- TII is essentially a store-and-forwarding mechanism. This implies the use of Request routing agents. The protocol followed by these Routers is defined in Message Routing on page 457.
- Dynamic Protocol Selection based on QoS is reconciled locally via information in the IOR and the local ORB. This implies several newly defined items at the protocol level:
  - Newly defined **IOP::ServiceContext** that contains QoS parameters.
  - Newly defined **IOP::ComponentId** tag for Messaging and a Component consisting of a representation of default QoS parameters.
  - Newly defined **IOP::ComponentId** tag and Component representing the transaction policy.
  - A newly defined **IOP::ComponentId** tag and Component containing a sequence of Request Routers. This sequence of Routers represents the preferred addressing strategy when TIIs are made on an Object.

A.4.2 Component Relationships

The relationship between the above described components is based on the following:

- QoS resolution should be performed by the client ORB if possible. Routers and/or Messaging-aware Adapters must ensure that only valid QoS have been selected.
- For efficient use of the Request/Reply Routers, their addressing information needs to be in the IOR.
- Request/Reply Routers re-route request and reply messages by explicitly sending messages between them, and then generating a regular GIOP request (and receiving a regular GIOP reply) when interfacing with the real target. To allow this routing to occur, the Router interface requires an encapsulation of a GIOP request in terms of:
  - Routing information including the message header and pertinent QoS information.
  - Message payload (the marshaled arguments and service contexts from the client).

The routers use the encapsulated QoS & re-routing information to re-route requests and replies and to decide whether to store request/reply information for a specified lifetime. The GIOP must be flexible enough to allow the Router closest to the request's destination to generate a request that looks like it was marshalled at the original client. This closest Router must be able to handle the full GIOP including the processing of a **LOCATION_FORWARD** reply without necessitating a return to the original client.

A.4.3 Router Administration Design

Several features of the Router administration design are worth note. These fall into two main areas:

- Static vs. Dynamic Routing - Routing information for an Object is available to the client ORB through a Profile Component in the object's IOR. This Component contains a sequence of Router references through which Time-Independent requests may pass on the way to the target. Therefore, portably exporting a target's preferred Routers must be done statically, at the time when the target's reference is created. This specification introduces no interfaces that suppo rt dynamic routing. It is expected that future work in CORBA Messaging will introduce portable administrative interfaces through which domains of Routers may be connected. Note that since the Router is an Object, the usual CORBA mechanisms for dynamic server relocation can certainly be used to allow migration of

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Routers and other such dynamic Routing activities.

- Minimize administrative traffic - Administrative interfaces are introduced that will allow a minimal amount of network bandwidth to be consumed when network disconnections occur. Furthermore, these administrative interfaces have been designed so that additional overhead is not consumed when Routers would normally be in an idle state. Administrative communication is only necessary when messages would otherwise have to be sent between Routers.

© ISO/IEC 2008 - All rights reserved
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex B for Clause 17
Conformance and Compatibility Issues

(normative)

This Annex specifies the points that must be met for a compliant implementation of CORBA Messaging and compatibility issues associated with this specification.

B.1 Conformance Issues

This specification can be separated into several logical components. In order to be conformant with this specification, the following mappings and features must be supported and implemented using the specified semantics:

- Changes to CORBA and Services. These changes include the modifications to GIOP, OTS, and the **SyncScopePolicy** refinements to **oneway** operations. This component includes the **Policy** management framework for Quality of Service as described in I - Introduction on page 419.
- Asynchronous Method Invocation (AMI) interfaces. This component includes the generation of asynchronous stubs (sendc/sendp operations) along with all interfaces and values upon which these stubs rely. All modifications to the DII are also included in this component.
- Quality of Service Policies for Messaging. These new Policies and their possible values are described in Messaging Quality of Service on page 417.

Implementation of the following component is not required to be conformant with this specification:

- Time-Independent Invocations (TII). This component includes the QoS **Policy** that supports TII (**RoutingTypePolicy**), the typed **PersistentPollers** described in Persistent Type-Specific Poller on page 440, and all interoperable Routing interfaces described in III - Introduction on page 460.

B.2 Compatibility Issues

B.2.1 Transaction Service

Transaction service compatibility is affected by two factors:

- Changes to existing transaction service behavior introduced as part of this specification.
- New transaction service functions introduced by this specification and the affect on existing implementations.

These are considered separately in each of the following sub clauses.

B.2.2 Changes to Current OTS Behavior

This specification deprecates the **TransactionalObject** interface defined in the *Transaction Service* specification. The **TransactionalObject** interface was defined to control propagation of the transaction context between the client and the server. An interface that inherits from **TransactionalObject** will automatically have the client's transaction context established by the server ORB before any operations on that interface are invoked.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A new mechanism for transaction propagation is independent of the use of inheritance from **TransactionalObject**. This mechanism has been defined so that *existing applications will continue to operate correctly without change* so they do not have to remove **TransactionalObject** inheritance from their existing IDL. At most, they will need to ensure that a definition of **CosTransactions::TransactionalObject** continues to be available to the IDL compiler.

The use of **TransactionalObject** inheritance had two other side effects in the *Transaction Service* specification.

- It affected the CORBA type of the interface being defined and thus the RepositoryID in the Interface Repository. This means that once interface inheritance is actually removed, transactional and non-transactional implementations of the same interface will have the same CORBA type.
- It provided for documentation within IDL of interfaces whose implementation was intended to be transactional. This enabled application developers to easily track their use of transactions.

Once **TransactionalObject** is actually removed, these side effects will no longer be present.

B.2.2.1 Effects of New OTS Functions on Existing OTS Implementations

This specification introduces new functions and behaviors to the *Transaction Service* to support the global transaction model used by messaging and to encode the transaction model in the object reference using a newly defined **TransactionPolicy**. The default for this new policy has been chosen to be compatible with existing CORBA behavior (i.e., a global transaction is associated with the target object if present) otherwise it is not. Existing applications, which will not create **TransactionPolicy** objects, will get the existing CORBA behavior.

Existing Clients with New Servers

New server applications can create object references with new **TransactionPolicy** selections that can be exported to existing clients. Depending on the **TransactionPolicy** selected, invoking methods on these objects may succeed transparently to the client or produce failures (in the form of system exceptions) existing clients will not have previously seen.

New AMI Clients with Existing Servers

Existing servers may require analysis of their existing semantics to determine the extent to which they may be able to operate with new clients, especially clients that use the new AMI request invocation model. In general the following are true and existing objects may as a result be usable without change by AMI clients:

- If transactions are not used, existing server objects will interoperate with new AMI clients.
- If transactions are used, AMI invocations will use the new queued transaction model causing invocations on the target object to be rejected with a new system exception.
- Depending on application design, it is possible that some (but not all) of these existing applications can operate successfully with AMI clients. This will require that these server objects be changed to produce new compatible object references.

It is normally true that a server application design, which depends on updating recoverable resources managed by objects at multiple sites cannot support an AMI invocation without producing different behavior. For the cases where this is not a problem the application can take advantage of new AMI clients by changing the object reference at creation time.

B.2.3 Security Service

The issues surrounding Security and Time-Independent Invocations must be addressed in a subsequent RFP. Current CORBA Security does fully support all other aspects of this specification, including typed deferred synchronous invocations.

--`,,```,,,,````-`-`,,`,,`,`,,`---

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex A
OMG IDL Tags and Exceptions

(normative)

This annex lists the standardized profile, service, component, policy tags and exception codes described in the CORBA documentation. Implementor-defined tags can also be registered in this manual. Requests to register tags with the OMG should be sent to *tag_request@omg.org*.

A.1 Profile ID Tags

Tag Name	Tag Value	Described in
ProfileId	TAG_INTERNET_IOP = 0	*CORBA 3.1, Part 2 - ORB Interoperability Architecture* clause.
ProfileId	TAG_MULTIPLE_COMPONENTS = 1	*CORBA 3.1, Part 2 - ORB Interoperability Architecture* clause.
ProfileId	TAG_SCCP_IOP = 2	CORBA/TC Interworking specification
ProfileId	TAG_UIPMC = 3	*CORBA 3.1, Part 2 - Unreliable Multicast* clause
ProfileId	TAG_MOBILE_TERMINAL_IOP = 4	Telecom Wireless specification

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A.2 Service ID Tags

Tag Name	Tag Value	Described in
ServiceId	TransactionService = 0	Object Transaction Service specification
ServiceId	CodeSets = 1	*CORBA 3.1, Part 2 - ORB Interoperability Architecture* clause.
ServiceId	ChainBypassCheck = 2	*Interoperability with non-CORBA Systems* clause: see CORBA, v3.0.3.
ServiceId	ChainBypassInfo = 3	*Interoperability with non-CORBA Systems* clause: see CORBA, v3.0.3.
ServiceId	LogicalThreadId = 4	*Interoperability with non-CORBA Systems* clause: see CORBA, v3.0.3.
ServiceId	BI_DIR_IIOP = 5	*CORBA 3.1, Part 2 - General Inter-ORB Protocol* clause.
ServiceId	SendingContextRunTime = 6	*CORBA 3.1, Part 1 - Value Type Semantics* clause.
ServiceId	INVOCATION_POLICIES = 7	*CORBA 3.1, Part 1 - CORBA Messaging* clause.
ServiceId	FORWARDED_IDENTITY = 8	Firewall Traversal specification (ptc/04-03-01)
ServiceId	UnknownExceptionInfo = 9	Java to IDL Language Mapping specification:
ServiceId	RTCorbaPriority = 10	Real-Time CORBA specification: see
ServiceId	RTCorbaPriorityRange = 11	Real-Time CORBA specification: see
ServiceId	FT_GROUP_VERSION = 12	*Fault Tolerant CORBA* clause: see CORBA, v3.0.3.
ServiceId	FT_REQUEST= 13	*Fault Tolerant CORBA* clause: see CORBA, v3.0.3.
ServiceId	ExceptionDetailMessage = 14	*CORBA 3.1, Part 2 - ORB Interoperability Architecture* chapter.
ServiceId	SecurityAttributeService = 15	*CORBA 3.1, Part 2 - Secure Interoperability* clause.
ServiceId	ActivityService = 16	Additional Structuring Mechanisms for the OTS.
ServiceId	RMICustomMaxStreamFormat = 17	Java to IDL Language Mapping specification.
ServiceId	ACCESS_SESSION_ID = 18	Telecom Service Access Subscription (TSAS) specification.
ServiceId	SERVICE_SESSION_ID = 19	Telecom Service Access Subscription (TSAS) specification.
ServiceId	FIREWALL_PATH = 20	Firewall Traversal specification (ptc/04-03-01)
ServiceId	FIREWALL_PATH_RESP = 21	Firewall Traversal specification (ptc/04-03-01)

A.3 Component ID Tags

Tag Name	Tag Value	Described in
ComponentId	TAG_ORB_TYPE = 0	*CORBA 3.1, Part 2 - ORB Interoperability Architecture* clause.
ComponentId	TAG_CODE_SETS = 1	*CORBA 3.1, Part 2 - ORB Interoperability Architecture* clause.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Tag Name	Tag Value	Described in
ComponentId	TAG_POLICIES = 2	*CORBA 3.1, Part 1 - CORBA Messaging* clause.
ComponentId	TAG_ALTERNATE_IIOP_ADDRESS = 3	*CORBA 3.1, Part 2 - General Inter-ORB Protocol* clause.
ComponentId	TAG_COMPLETE_OBJECT_KEY = 5	*The DCE ESIOP* clause: see CORBA, v3.0.3.
ComponentId	TAG_ENDPOINT_ID_POSITION = 6	*The DCE ESIOP* clause: see CORBA, v3.0.3.
ComponentId	TAG_LOCATION_POLICY = 12	*The DCE ESIOP* clause: see CORBA, v3.0.3.
ComponentId	TAG_ASSOCIATION_OPTIONS =13	Security Service specification.
ComponentId	TAG_SEC_NAME = 14	
ComponentId	TAG_SPKM_1_SEC_MECH = 15	
ComponentId	TAG_SPKM_2_SEC_MECH = 16	
ComponentId	TAG_KerberosV5_SEC_MECH = 17	
ComponentId	TAG_CSI_ECMA_Secret_SEC_MECH = 18	
ComponentId	TAG_CSI_ECMA_Hybrid_SEC_MECH = 19	
ComponentId	TAG_SSL_SEC_TRANS = 20	
ComponentId	TAG_CSI_ECMA_Public_SEC_MECH = 21	
ComponentId	TAG_GENERIC_SEC_MECH = 22	
ComponentId	TAG_FIREWALL_TRANS = 23	Firewall Traversal specification (ptc/04-03-01)
ComponentId	TAG_SCCP_CONTACT_INFO = 24	CORBA/TC Interworking and SCCP Inter-ORB Protocol specification.
ComponentId	TAG_JAVA_CODEBASE = 25	Java to IDL Language Mapping specification.
ComponentId	TAG_TRANSACTION_POLICY = 26	Object Transaction Service specification.
ComponentId	TAG_ FT_GROUP= 27	*Fault Tolerant CORBA* clause: see CORBA, v3.0.3.
ComponentId	TAG_ FT_PRIMARY= 28	*Fault Tolerant CORBA* clause: see CORBA, v3.0.3.
ComponentId	TAG_ FT_HEARTBEAT_ENABLED = 29	*Fault Tolerant CORBA* clause: see CORBA, v3.0.3.
ComponentId	TAG_MESSAGE_ROUTERS = 30	*CORBA 3.1, Part 1 - CORBA Messaging* clause.
ComponentId	TAG_OTS_POLICY = 31	Object Transaction Service specification.
ComponentId	TAG_INV_POLICY = 32	Object Transaction Service specification.
ComponentId	TAG_CSI_SEC_MECH_LIST = 33	*CORBA 3.1, Part 2 - Secure Interoperability* clause
ComponentId	TAG_NULL_TAG = 34	*CORBA 3.1, Part 2 - Secure Interoperability* clause
ComponentId	TAG_SECIOP_SEC_TRANS = 35	*CORBA 3.1, Part 2 - Secure Interoperability* clause
ComponentId	TAG_TLS_SEC_TRANS = 36	*CORBA 3.1, Part 2 - Secure Interoperability* clause
ComponentId	TAG_ACTIVITY_POLICY = 37	Additional Structuring Mechanisms for the OTS.
ComponentId	TAG_RMI_CUSTOM_MAX_STREAM_FORMAT = 38	Java to IDL Language Mapping specification.
ComponentId	TAG_GROUP = 39	*CORBA 3.1, Part 2 - Unreliable Multicast* clause
ComponentId	TAG_GROUP_IIOP = 40	*CORBA 3.1, Part 2 - Unreliable Multicast* clause
ComponentId	TAG_PASSTHRU_TRANS = 41	Firewall Traversal (ptc/04-03-01)
ComponentId	TAG_FIREWALL_PATH = 42	Firewall Traversal (ptc/04-03-01)
ComponentId	TAG_IIOP_SEC_TRANS = 43	Firewall Traversal (ptc/04-03-01)

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Tag Name	Tag Value	Described in
ComponentId	TAG_HOME_LOCATION_INFO = 44	Telecom Wireless specification.
ComponentId	TAG_DCE_STRING_BINDING = 100	*The DCE ESIOP* clause: see CORBA, v3.0.3.
ComponentId	TAG_DCE_BINDING_NAME = 101	*The DCE ESIOP* clause: see CORBA, v3.0.3.
ComponentId	TAG_DCE_NO_PIPES = 102	*The DCE ESIOP* clause: see CORBA, v3.0.3.
ComponentId	TAG_DCE_SEC_MECH = 103	Security Service specification.
ComponentId	TAG_INET_SEC_TRANS = 123	Security Service specification.

A.4 Policy Type Tags

The table below lists the standard policy types that are defined by various parts of CORBA and CORBA Services in this version of CORBA/IIOP.

Policy Type	Policy Interface	Defined in	Uses create_policy
SecClientInvocationAccess = 1	SecurityAdmin::AccessPolicy	Security Service specification.	N
SecTargetInvocationAccess = 2	SecurityAdmin::AccessPolicy		N
SecApplicationAccess = 3	SecurityAdmin::AccessPolicy		N
SecClientInvocationAudit = 4	SecurityAdmin::AuditPolicy		N
SecTargetInvocationAudit = 5	SecurityAdmin::AuditPolicy		N
SecApplicationAudit = 6	SecurityAdmin::AuditPolicy		N
SecDelegation = 7	SecurityAdmin::Delegation Policy		N
SecClientSecureInvocation = 8	SecurityAdmin::SecureInvocationPolicy		N
SecTargetSecureInvocation = 9	SecurityAdmin::SecureInvocationPolicy		N
SecNonRepudiation = 10	NRService::NRPolicy		N
SecConstruction = 11	CORBA::SecConstruction	*CORBA 3.1, Part 1 - ORB Interface* clause	N
SecMechanismPolicy = 12	SecurityLevel2::MechanismPolicy	Security Service specification.	Y
SecInvocationCredentialsPolicy = 13	SecurityLevel2::InvocationCredentials Policy		Y
SecFeaturesPolicy = 14	SecurityLevel2::FeaturesPolicy		Y
SecQOPPolicy = 15	SecurityLevel2::QOPPolicy		Y

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Policy Type	Policy Interface	Defined in	Uses create _policy
THREAD_POLICY_ID = 16	PortableServer::ThreadPolicy	*CORBA 3.1, Part 1 - Portable Object Adapter* clause	Y
LIFESPAN_POLICY_ID = 17	PortableServer::LifespanPolicy		Y
ID_UNIQUENESS_POLICY_ID = 18	PortableServer::IdUniquenessPolicy		Y
ID_ASSIGNMENT_POLICY_ID = 19	PortableServer::IdAssignmentPolicy		Y
IMPLICIT_ACTIVATION_POLICY_ID = 20	PortableServer::ImplicitActivationPolicy		Y
SERVENT_RETENTION_POLICY_ID = 21	PortableServer::ServentRetentionPolicy		Y
REQUEST_PROCESSING_POLICY_ID = 22	PortableServer::RequestProcessingPolicy		Y
REBIND_POLICY_TYPE = 23	Messaging::RebindPolicy	*CORBA 3.1, Part 1 - CORBA Messaging* clause	Y
SYNC_SCOPE_POLICY_TYPE = 24	Messaging::SyncScopePolicy		Y
REQUEST_PRIORITY_POLICY_TYPE = 25	Messaging::RequestPriorityPolicy		Y
REPLY_PRIORITY_POLICY_TYPE = 26	Messaging::ReplyPriorityPolicy		Y
REQUEST_START_TIME_POLICY_TYPE = 27	Messaging::RequestStartTimePolicy		Y
REQUEST_END_TIME_POLICY_TYPE = 28	Messaging::RequestEndTimePolicy		Y
REPLY_START_TIME_POLICY_TYPE = 29	Messaging::ReplyStartTimePolicy		Y
REPLY_END_TIME_POLICY_TYPE = 30	Messaging::ReplyEndTimePolicy		Y
RELATIVE_REQ_TIMEOUT_POLICY_ TYPE = 31	Messaging::RelativeRequestTimeoutPolicy		Y
RELATIVE_RT_TIMEOUT_POLICY_ TYPE = 32	Messaging::RelativeRoundtripTimeout Policy		Y
ROUTING_POLICY_TYPE = 33	Messaging::RoutingPolicy		Y
MAX_HOPS_POLICY_TYPE =34	Messaging::MaxHopsPolicy		Y
QUEUE_ORDER_POLICY_TYPE = 35	Messaging::QueueOrderPolicy		Y
FIREWALL_POLICY_TYPE = 36	Firewall::FirewallPolicy	Firewall Traversal specification: (ptc/04-03-01)	Y
BIDIRECTIONAL_POLICY_TYPE = 37	BiDirPolicy::BidirectionalPolicy	*CORBA 3.1, Part 2 - General Inter-ORB Protocol* clause	Y
SecDelegationDirectivePolicy = 38	SecurityLevel2::DelegtionDirectivePolicy	Security Service specification.	Y
SecEstablishTrustPolicy = 39	SecurityLevel2::EstablishTrustPolicy		Y

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Policy Type	Policy Interface	Defined in	Uses create _policy
PRIORITY_MODEL_POLICY_TYPE = 40	RTCORBA::PriorityModelPolicy	Real-Time CORBA specification.	Y
THREADPOOL_POLICY_TYPE = 41	RTCORBA::ThreadpoolPolicy		Y
SERVER_PROTOCOL_POLICY_TYPE = 42	RTCORBA::ServerProtocolPolicy		Y
CLIENT_PROTOCOL_POLICY_TYPE = 43	RTCORBA::ClientProtocolPolicy		Y
PRIVATE_CONNECTION_POLICY_ TYPE = 44	RTCORBA::PrivateConnectionpolicy		Y
PRIORITY_BANDED_CONNECTION_ POLICY_TYPE = 45	RTCORBA::PriorityBandedConnection Policy		Y
TransactionPolicyType = 46	CosTransactions::TransactionPolicy	Object Transaction Service specification.	Y
REQUEST_DURATION_POLICY_ TYPE = 47		*Fault Tolerant CORBA*: see CORBA, v3.0.3.	
HEARTBEAT_POLICY_TYPE = 48		*Fault Tolerant CORBA*: see CORBA, v3.0.3.	
HEARTBEAT_ENABLED_POLICY_ TYPE = 49			
IMMEDIATE_SUSPEND_POLICY_ TYPE = 50	valuetype MessageRouting:: ImmediateSuspend	*CORBA 3.1, Part 1 - CORBA Messaging* clause	N
UNLIMITED_PING_POLICY_TYPE = 51	valuetype MessageRouting::UnlimitedPing		N
LIMITED_PING_POLICY_TYPE = 52	valuetype MessageRouting::LimitedPing		N
DECAY_POLICY_TYPE = 53	valuetype MessageRouting::DecayPolicy		N
RESUME_POLICY_TYPE = 54	valuetype MessageRouting::ResumePolicy		N
INVOCATION_POLICY_TYPE = 55	CosTransactions::InvocationPolicy	Object Transaction Service specification.	Y
OTS_POLICY_TYPE = 56	CosTransactions::OTSPolicy		Y
NON_TX_TARGET_POLICY_TYPE = 57	CosTransactions::NonTxTargetPolicy		Y
ActivityPolicyType = 58	CORBA::PolicyType	Additional Structuring Mechanisms for the OTS.	Y
OSA_MANAGER_POLICY = 59		Security Domain Membership (orbos/01-06-01)	
ODM_MANAGER_POLICY = 60			

© ISO/IEC 2008 - All rights reserved
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Policy Type	Policy Interface	Defined in	Uses create _policy
PATH_SELECTION_POLICY_TYPE = 61		Firewall Traversal specification (ptc/04-03-01)	
PATH_INSERTION_POLICY_TYPE = 62			
PROCESSING_MODE_POLICY_TYPE = 63		*CORBA 3.1, Part 1 - Portable Interceptor* clause	

A.5 Exception Codes

If an exception that is to be raised for an error condition does not explicitly specify a specific standard minor code for that error condition, implementations can either use a minor code of zero, or use a vendor-specific minor code to convey more detail about the error.

The following table specifies standard minor exception codes that have been assigned for the standard system exceptions. The actual value that is to be found in the **minor** field of the **ex_body** structure is obtained by or-ing the values in this table with the **OMGVMCID** constant. For example "Missing local value implementation" for the exception NO_IMPLEMENT would be denoted by the **minor** value **0x4f4d0001.**

.

SYSTEM EXCEPTION	MINOR CODE	EXPLANATION
ACTIVITY_COMPLETED	1	Activity context completed through timeout, or in some way other than requested.
ACTIVITY_REQUIRED	1	Calling thread lacks required activity context.
BAD_CONTEXT	1	IDL context not found.
	2	No matching IDL context property.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

SYSTEM EXCEPTION	MINOR CODE	EXPLANATION
BAD_INV_ORDER	1	Dependency exists in IFR preventing destruction of this object.
	2	Attempt to destroy indestructible objects in IFR.
	3	Operation would deadlock.
	4	ORB has shutdown
	5	Attempt to invoke **send** or **invoke** operation of the same **Request** object more than once.
	6	Attempt to set a servant manager after one has already been set.
	7	**ServerRequest::arguments** called more than once or after a call to S**erverRequest:: set_exception**.
	8	**ServerRequest::ctx** called more than once or before **ServerRequest::arguments** or after **ServerRequest::ctx, ServerRequest::set_result** or **ServerRequest::set_exception**.
	9	**ServerRequest::set_result** called more than once or before **ServerRequest::arguments** or after **ServerRequest::set_result** or **ServerRequest::set_exception.**
	10	Attempt to send a DII request after it was sent previously.
	11	Attempt to poll a DII request or to retrieve its result before the request was sent.
	12	Attempt to poll a DII request or to retrieve its result after the result was retrieved previously.
	13	Attempt to poll a synchronous DII request or to retrieve results from a synchronous DII request.
	14	Invalid portable interceptor call.
	15	Service context add failed in portable interceptor because a service context with the given id already exists.
	16	Registration of **PolicyFactory** failed because a factory already exists for the given **PolicyType.**

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

SYSTEM EXCEPTION	MINOR CODE	EXPLANATION
	17	**POA** cannot create **POA**s while undergoing destruction
	18	Attempt to reassign priority.
	19	An OTS/XA integration **xa_start** call returned **XAER_OUTSIDE.**
	20	An OTS/XA integration **xa_** call returned **XAER_PROTO.**
	21	Transaction context of request and client threads do not match in interceptor.
	22	Poller has not returned any response yet.
	23	Registration of TaggedProfileFactory failed because a factory already exists for the given id.
	24	Registration of TaggedComponentFactory failed because a factory already exists for the given id.
	25	Iteration has no more elements.
	26	Invocation of this operation not allowed in post_init.
BAD_OPERATION	1	ServantManager returned wrong servant type.
	2	Operation or attribute not known to target object

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

SYSTEM EXCEPTION	MINOR CODE	EXPLANATION
BAD_PARAM	1	Failure to register, unregister, or lookup value factory.
	2	RID already defined in IFR.
	3	Name already used in the context in IFR.
	4	Target is not a valid container.
	5	Name clash in inherited context.
	6	Incorrect type for abstract interface.
	7	string_to_object conversion failed due to bad scheme name.
	8	string_to_object conversion failed due to bad address.
	9	string_to_object conversion failed due to bad bad schema specific part.
	10	string_to_object conversion failed due to non specific reason.
	11	Attempt to derive abstract interface from non-abstract base interface in the Interface Repository.
	12	Attempt to let a ValueDef support more than one non-abstract interface in the Interface Repository.
	13	Attempt to use an incomplete **TypeCode** as a parameter.
	14	Invalid object id passed to **POA::create_reference_by_id.**
	15	Bad **name** argument in **TypeCode** operation.
	16	Bad **RepositoryId** argument in **TypeCode** operation.
	17	Invalid member name in **TypeCode** operation.
	18	Duplicate label value in **create_union_tc**.
	19	Incompatible TypeCode of label and discriminator in **create_union_tc**.
	20	Supplied discriminator type illegitimate in **create_union_tc**.
	21	**Any** passed to **ServerRequest::set_exception** does not contain an exception.
	22	Unlisted user exception passed to **ServerRequest::set_exception**.
	23	wchar transmission code set not in service context.
	24	Service context is not in OMG-defined range.
	25	Enum value out of range.
	26	Invalid service context Id in portable interceptor.
	27	Attempt to call **register_initial_reference** with a null **Object.**
	28	Invalid component Id in portable interceptor.
	29	Invalid profile Id in portable interceptor.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

SYSTEM EXCEPTION	MINOR CODE	EXPLANATION
	30	Two or more **Policy** objects with the same **PolicyType** value supplied to **Object::set_policy_overrides** or **PolicyManager::set_policy_overrides**.
	31	Attempt to define a **oneway** operation with non-void result, **out** or **inout** parameters or user exceptions.
	32	DII asked to create request for an implicit operation.
	33	An OTS/XA integration **xa**_ call returned **XAER_INVAL**.
	34	Union branch modifier called with bad case label discriminator.
	35	Illegal IDL context property name.
	36	Illegal IDL property search string.
	37	Illegal IDL context name.
	38	Non-empty IDL context.
	39	Unsupported RMI/IDL custom value type stream format.
	40	ORB output stream does not support ValueOutputStream interface.
	41	ORB input stream does not support ValueInputStream interface.
	42	Character support limited to ISO 8859-1 for this object reference.
	43	Attempt to add a Pollable to a second PollableSet.
BAD_TYPECODE	1	Attempt to marshal incomplete **TypeCode.**
	2	Member type code illegitimate in **TypeCode** operation.
	3	Illegal parameter type.
CODESET_INCOMPATIBLE	1	Codeset negotiation failed.
	2	Codeset delivered in CodeSetContext is not supported by server as transmission codeset.
DATA_CONVERSION	1	Character does not map to negotiated transmission code set.
	2	Failure of **PriorityMapping** object.
IMP_LIMIT	1	Unable to use any profile in IOR.
INITIALIZE	1	Priority range too restricted for ORB.
INTERNAL	1	An OTS/XA integration **xa**_ call returned **XAER_RMERR**.
	2	An OTS/XA integration **xa**_ call returned **XAER_RMFAIL**.
INTF_REPOS	1	Interface Repository not available
	2	No entry for requested interface in Interface Repository
INVALID_ACTIVITY	1	Transaction or Activity resumed in wrong context, or invocation incompatible with Activity's current state.
INV_OBJREF	1	wchar Code Set support not specified.
	2	Codeset component required for type using **wchar** or **wstring** data

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

SYSTEM EXCEPTION	MINOR CODE	EXPLANATION
INV_POLICY	1	Unable to reconcile IOR specified policy with effective policy override.
	2	Invalid **PolicyType.**
	3	No **PolicyFactory** has been registered for the given **PolicyType**.
MARSHAL	1	Unable to locate value factory.
	2	**ServerRequest::set_result** called before **ServerRequest::ctx** when the operation IDL contains a context clause.
	3	**NVList** passed to **ServerRequest::arguments** does not describe all parameters passed by client.
	4	Attempt to marshal **Local** object.
	5	**wchar** or **wstring** data erroneosly sent by client over **GIOP** 1.0 connection
	6	**wchar** or **wstring** data erroneously returned by server over **GIOP** 1.0 connection.
	7	Unsupported RMI/IDL custom value type stream format.
NO_IMPLEMENT	1	Missing local value implementation.
	2	Incompatible value implementation version.
	3	Unable to use any profile in IOR.
	4	Attempt to use DII on Local object.
	5	Biomolecular Sequence Analysis iterator cannot be reset.
	6	Biomolecular Sequence Analysis metadata is not available as XML.
	7	Genomic Maps iterator cannot be rest.
	8	Operation not implemented in local object.
NO_RESOURCES	1	Portable Interceptor operation not supported in this binding.
	2	No connection for request's priority.
NO_RESPONSE	1	Reply is not available immediately in a non-blocking call.
OBJ_ADAPTER	1	System exception in **AdapterActivator::unknown_adapter**.
	2	Incorrect servant type returned by servant manager.
	3	No default servant available [POA policy].
	4	No servant manager available [POA Policy].
	5	Violation of POA policy by **ServantActivator::incarnate.**
	6	Exception in **PortableInterceptor::**IORInterceptor.components_established.
	7	Null servant returned by servant manager

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

SYSTEM EXCEPTION	MINOR CODE	EXPLANATION
OBJECT_NOT_EXIST	1	Attempt to pass an unactivated (unregistered) value as an object reference.
	2	Failed to create or locate Object Adapter.
	3	Biomolecular Sequence Analysis Service is no longer available.
	4	Object Adapter inactive.
	5	This Poller has already delivered a reply to some client.
TIMEOUT	1	Reply is not available in the Poller by the timeout set for it.
	2	End time specified in RequestEndTimePolicy or RelativeRequestTimeoutPolicy has expired.
	3	End time specified in ReplyEndTimePolicy or RelativeReplyTimeoutPolicy has expired.
TRANSACTION_ROLLEDBACK	1	An OTS/XA integration **xa**_ call returned **XAER_RB**.
	2	An OTS/XA integration **xa**_ call returned **XAER_NOTA**.
	3	OTS/XA integration **end** was called with success set to **TRUE** while transaction rollback was deferred.
	4	Deferred transaction rolled back.
TRANSIENT	1	Request discarded because of resource exhaustion in **POA**, or because POA is in discarding state.
	2	No usable profile in IOR.
	3	Request cancelled.
	4	**POA** destroyed.
UNKNOWN	1	Unlisted user exception received by client.
	2	Non-standard System Exception not supported.
	3	An unknown user exception received by a portable interceptor.

A.6 Identity Tokens

The following identity tokens are defined in the Security Context clause (CORBA Part 2: Interoperability) and the Firewall Traversal specification (ptc/04-03-01). These tokens must be powers of two.

- ITTAbsent = 0;
- ITTAnonymous = 1;
- ITTPrincipalName = 2;
- ITTX509CertChain = 4;
- ITTDistinguishedName = 8;
- ITTCompoundToken = 16;

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex B
Legal Information

(normative)

B.1 Copyright Information

Copyright © 1997-2001 Electronic Data Systems Corporation
Copyright © 1997-2001 Hewlett-Packard Company
Copyright © 1997-2001 IBM Corporation
Copyright © 1997-2001 ICON Computing
Copyright © 1997-2001 i-Logix
Copyright © 1997-2001 IntelliCorp
Copyright © 1997-2001 Microsoft Corporation
Copyright © 2008 Object Management Group
Copyright © 1997-2001 ObjecTime Limited
Copyright © 1997-2001 Oracle Corporation
Copyright © 1997-2001 Platinum Technology, Inc.
Copyright © 1997-2001 Ptech Inc.
Copyright © 1997-2001 Rational Software Corporation
Copyright © 1997-2001 Reich Technologies
Copyright © 1997-2001 Softeam
Copyright © 1997-2001 Sterling Software
Copyright © 1997-2001 Taskon A/S
Copyright © 1997-2001 Unisys Corporation

B.2 Use Of Specification - Terms, Conditions & Notices

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

B.3 Licenses

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification,

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

B.4 Patents

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

B.5 General Use Restrictions

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

B.6 Disclaimer Of Warranty

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.

IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

B.7 Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 140 Kendrick Street, Needham, MA 02494, U.S.A.

B.8 Trademarks

MDA®, Model Driven Architecture®, UML®, UML Cube logo®, OMG Logo®, CORBA® and XMI® are registered trademarks of the Object Management Group, Inc., and Object Management Group™, OMG™ , Unified Modeling Language™, Model Driven Architecture Logo™, Model Driven Architecture Diagram™, CORBA logos™, XMI Logo™, CWM™, CWM Logo™, IIOP™ , MOF™ and OMG Interface Definition Language (IDL)™ , and Systems Modeling Language (SysML™) are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

B.9 Compliance

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

© ISO/IEC 2008 - All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex C
OMG's Issue Reporting Procedure

(normative)

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex D
Acknowledgements

(normative)

The following companies submitted and/or supported parts of the specifications that were approved by the Object Management Group to become *CORBA*:

- Adiron, LLC
- Alcatel
- BEA Systems, Inc.
- BNR Europe Ltd.
- Borland International, Inc.
- Compaq Computer Corporation
- Concept Five Technologies
- Cooperative Research Centre for Distributed Systems Technology (DSTC)
- Defense Information Systems Agency
- Digital Equipment Corporation
- Ericsson
- Eternal Systems, Inc.
- Expersoft Corporation
- France Telecom
- FUJITSU LIMITED
- Genesis Development Corporation
- Gensym Corporation
- Hewlett-Packard Company
- HighComm
- Highlander Communications, L.C.
- Humboldt-University
- HyperDesk Corporation
- ICL, Plc.
- Inprise Corporation
- International Business Machines Corporation
- International Computers, Inc.
- IONA Technologies, Plc.
- Lockheed Martin Federal Systems, Inc.
- Lucent Technologies, Inc.

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- Micro Focus Limited
- MITRE Corporation
- Motorola, Inc.
- NCR Corporation
- NEC Corporation
- Netscape Communications Corporation
- Nortel Networks
- Northern Telecom Corporation
- Novell, Inc.
- Object Design, Inc.
- Objective Interface Systems, Inc.
- Object-Oriented Concepts, Inc.
- OC Systems, Inc.
- Open Group - Open Software Foundation
- Oracle Corporation
- PeerLogic, Inc.
- Persistence Software, Inc.
- Promia, Inc.
- Siemens Nixdorf Informationssysteme AG
- SPAWAR Systems Center
- Sun Microsystems, Inc.
- SunSoft, Inc.
- Sybase, Inc.
- Telefónica Investigación y Desarrollo S.A. Unipersonal
- TIBCO, Inc.
- Tivoli Systems, Inc.
- Tri-Pacific Software, Inc.
- University of California, Santa Barbara
- University of Rhode Island
- Visual Edge Software, Ltd.
- Washington University

In addition to the preceding contributors, the OMG would like to acknowledge Mark Linton at Silicon Graphics and Doug Lea at the State University of New York at Oswego for their work on the C++ mapping.

© ISO/IEC 2008 - All rights reserved
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

INDEX

A

Abstract interfaces 7, 173
Abstract Model Description
  Dynamic Skeleton Interface 311
  Implicit Activation 309
  Location Transparency 312
  Model Architecture 305
  Model Components 303
  Multi-threading 310
  Object Activation States 308
  POA Creation 306
  Reference Creation 307
  Request Processing 309
Abstract object model 3
AbstractInterfaceDef 253, 254
Activation 9
ACTIVE 470
AdapterActivator 307
add_pollable 188
AliasDef 243
AMI/TII Abstract Model Design 481
any type 62, 178
Any values
  dynamic management overview 197
array
  syntax of 68
ArrayDef 245
associated_handler 437
Async Operation Mapping 428
Asynchronous Method Invocation (AMI) 482
  Components 481
Asynchronous Method Signatures 441
asynchrony
  and narrowing of object references 487
AsyncOperation Mapping
  Callback Model Signatures (sendc) 429
  Polling Model Signatures (sendp) 431
attribute
  defined 9
attribute associated_handler 461
Attribute Declaration 74
AttributeDef 247

B

backoff_factor 471
base interface 47
base_interval_seconds 471
Basic types 5
Basic Type-Specific Poller 437
Basics
  Interface Repository Objects 225
  Names and Identifiers 224
  Structure and Navigation of the Interface Repository 226
  Types and TypeCodes 225
blocking 461
body 459
boolean is_a operation
  OMG PIDL for 108
boolean types 62

C

C++ Language
  Usage in 218
Callback Model
  exception delivery in 432
  signatures 429
Callback Model Detailed Design 485
char type 61
Client 16, 20
Client Stubs 17
client-side components
  Asynchronous Method Invocation (AMI) 482
  Poller 481
  ReplyHandler 481
client-side policies 479
Common Data Structures 177
Complex Declarator
  Arrays 68
  Deprecated Anonymous Types 69
component
  tags for 493
Component Design 480
Component Relationships 480, 482
Concrete object model 3
Constant Declaration
  Semantics 56
  Syntax 55
ConstantDef 240
Constructed Recursive Types 64
Constructed types 6
Contained interface
  OMG IDL for 231
Container interface 229
  OMG IDL for 233
CORBA
  Any values
    dynamic creation of 218
    dynamic interpretation 219
CORBA module 86
  types defined by 177
CORBA Required Object Adapter
  Portable Object Adapter 24
CORBA_free 179
create_dii_pollable 188
create_persistent_request 462
create_pollable_set 188
create_request operation 106

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Current 337

D
Data type
  basic OMG IDL 60–62
  constructed OMG IDL 62–66
  constructs for OMG IDL 59
  native 61
  OMG IDL template 66–67
deactivation 9
DecayPolicy 471
derived interface 47
DII Deferred Synchronous 487
DIIPollable interface 188
DSI
  Language Mapping 195
Dynamic creation of CORBA
  Any values 218
Dynamic interpretation of CORBA
  Any values 219
Dynamic Invocation interface
  overview of 13, 17
  parameters 177
  request routines 179
dynamic protocol selection 488
dynamic routing 488
Dynamic Skeleton Interface 14, 18, 193
DynAny
  interface 206, 215
  management overview 197
DynAny API 198
  Creating a DynAny object 204
  The DynAny interface 206
  The DynArray interface 215
  The DynEnum interface 210
  The DynFixed Interface 210
  The DynSequence interface 214
  The DynStruct interface 211
  The DynUnion interface 212
  The DynValue interface 216
  The DynValueBox interface 217
DynAny object
  basic data type values 208
  copying 207
  creating 204
  destroying 207
  generating an any value from 207
  initializing from an any value 206
  initializing from another DynAny object 206
  interface 206, 215
  TypeCode associated with 206
DynEnum interface 210
DynFixed interface 210
DynSequence interface 214
DynStruct interface 211
DynUnion interface 212
DynValue interface 216
DynValueBox interface 217

E
EnumDef 243
Enumerated types 66
Example Programmer Usage
  C++ Example of Callback Client Program 447
  C++ Example of Generated ReplyHandler 442
  C++ Example of Polling Client Program 450
  C++ Example of User-Implemented ReplyHandler 444
  C++ Example of Using PollableSet in a Client Program 452
  Client-Side C++ Example for the Asynchronous Method Signatures 441
  Client-Side C++ Example of the Callback Model 442
  Client-Side C++ Example of the Polling Model 449
  Example Programmer Usage (Examples Mapped to C++) 441
Example Programmer Usage
  Server Side 454
Exception Declaration 71
Exception replies 485
ExceptionDef 246
Exceptions 8
  InvalidState 470
  REBIND 155
  TIMEOUT 155
  TRANSACTION_UNAVAILABLE 155
ExplicitRequest State
  ServerRequestPseudo-Object 194
expression
  context 74
  raises 73

F
FixedDef 245
floating point type 61
Foreign Object Systems
  Integration of 24
Forward Declarations 64

G
Generic Poller Value
  associated_handler 437
  is_from_poller 437
  operation_name 437
  operation_target 436
get_client_policy 111
get_interface operation 107, 228
  OMG PIDL for 107
get_policy_overrides 111, 133
get_reply 461
giop_version 459
global name 88

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

and inheritance 88
and Interface Repository ScopedName 230

H
handler 459
handler_type 459
hash operation 109

I
IdAssignmentPolicy 326
identifier 45
IDL for PortableServer Module 338
IDLType interface 229
IdUniquenessPolicy 326
ImmediateSuspend 470
implementation
defined 10
model for 9
Implementation Dependencies 223
Managing Interface Repositories 223
Implementation Repository 19
Implementation Skeleton 18
ImplicitActivationPolicy 328
incarnate operation 321
Initial Request Router 463
integer tdata type 61
interface 6, 7
Interface Declaration
Forward Declaration 46
Interface Body 46
Interface Header 45
Interface Inheritance 47
Interface Inheritance Specification 46
interface object 226
Interface Repository 14, 19, 221
AliasDef, OMG IDL 243
and identifiers 230
ArrayDef, OMG IDL 245
AttributeDef, OMG IDL 247
Contained interface, OMG IDL 231
Container 229
Container interface, OMG IDL 233
ExceptionDef interface 246
IDLType 229
inserting information 223
InterfaceDef, OMG IDL 250, 257
IRObject interface 229
IRObject interface, OMG IDL 230
location of interfaces in 227
ModuleDef interface, OMG IDL 240
OMG IDL for 284
OperationDef, OMG IDL 248
overview of 19, 221
PrimitiveDef, OMG IDL 244
Repository interface, OMG IDL 238, 264
SequenceDef, OMG IDL 245
StringDef, OMG IDL 244
StructDef, OMG IDL 241
TypeCode 144
TypeCode interface, OMG IDL 138
Interface Repository Interfaces
AbstractInterfaceDef 253
AliasDef 243
ArrayDef 245
ConstantDef 240
Contained 231
Container 233
EnumDef 243
ExceptionDef 246
FixedDef 245
IDLType 238
InterfaceDef 250
IRObject 230
LocalInterfaceDef 255
ModuleDef 240
NativeDef 262
OperationDef 248
PrimitiveDef 244
Repository 238
SequenceDef 245
StringDef 244
StructDef 241
Supporting Type Definitions 229
TypedefDef 241
UnionDef 242
ValueBoxDef 262
ValueDef 257, 260
ValueMemberDef 256
WstringDef 244
interface repository objects 225
interface type 7
InterfaceDef 228, 250, 252
OMG IDL for 250, 257
Interfaces 312
AdapterActivator Interface 319
attribute associated_handler 461
create_persistent_request 462
Current Operations 337
get_reply 461
Handling LOCATION_FORWARD Replies 466
Handling of Service Contexts 466
Initial Request Router 463
interface MaxHopsPolicy 425
interface PolicyCurrent 134
interface PolicyManager 132
interface Pollable 187
interface PollableSet 188
interface QueueOrderPolicy 425
interface RelativeRequestTimeoutPolicy 423
interface RelativeRoundtripTimeoutPolicy 424
interface ReplyEndTimePolicy 423

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

interface ReplyPriorityPolicy 422
interface ReplyStartTimePolicy 423
interface RequestEndTimePolicy 422
interface RequestPriorityPolicy 421
interface RequestStartTimePolicy 422
interface RoutingPolicy 424
interfaceRebindPolicy 420
Intermediate Request Router 464
Invoking Client 462
PersistentRequest 461
PersistentRequestRouter 461
POA Interface 328
POA Policy Objects 325
POAManager Interface 314
readonly attribute reply_available 461
reply 460
ReplyHandler 460
Replying to a Type-specific ReplyHandler 466
Replying to an UntypedReplyHandler 466
Request Routing Algorithm 464
Router 460
RouterAdmin 471
Routing of Replies 466
Routing Protocol 462
send_multiple_requests 460
send_request 460
ServantActivator Interface 321
ServantLocator Interface 323
ServantManager Interface 320
SyncScopePolicy 421
Target Router 464
The Servant IDL Type 313
UntypedReplyHandler 460, 466
Intermediate Request Router 464
Interoperable Routing Protocol 482
interval_limit 471
InvalidState 470
INVOCATION_POLICIES 479
invocations
DII Deferred Synchronous 487
Invoking Client 462
IRObject 230
IRObject interface 229
OMG IDL for 230
is_equivalent operation 109
is_from_poller 437
is_ready 187

L

language mapping
overview 17
Lexical Conventions 28
Comments 31
Escaped Identifiers 32
Identifiers 31
Keywords 33
Tokens 31
LifespanPolicy 326
LimitedPing 471
Liskov substitution principle 7
List Operations 189
add_item 190
create_list 190
create_operation_list 191
free 191
free_memory 191
get_count 191
Literals
Character Literals 34
Fixed-Point Literals 36
Floating-point Literals 35
Integer Literals 34
String Literals 35
Local Interface 50
LocalInterfaceDef 255, 256
LocalObject 113
LOCATION_FORWARD Replies 466
logical_type_id string 108

M

Mapping of OMG IDL to Programming Languages 17
max_backoffs 471
MaxHopsPolicy interfaces 425
Memory Usage 179
message 460, 462
message payload 488
Message Routing 457
Message Routing Abstract Model Design 487
Message Routing Interoperability 455
MessageBody 459
Messaging module 417, 425, 472
Messaging Programming Model 426
Messaging QoS 425
propagation of 425
Messaging QoS Profile Component 426
Messaging QoS Service Context 426
Messaging Quality of Service 417
method 9
mode 8
Model Components 479
module
Messaging 425
Module Declaration 45
ModuleDef 240
modules
Messaging 417, 472
multiple inheritance 47

N

NamedValue type 177
Names and Scoping

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Qualified Names 87
Scoping Rules and Name Resolution 89
Special Scoping Rules for Type Names 91
NativeDef 262
NO_EXCEPTION 466
NO_REBIND 420
NO_RECONNECT 420
NOT_REGISTERED 470
number_left 189
NVList interface
add_item operation 190
create_list operation 190
create_operation_list 191
get_count operation 191
NVList operation
free_memory operation 191

O

object 4
implementation 10, 16
invocation 17, 18
reference 16
object adapter 15, 18, 22
functions of 23
overview of 14, 18
structure 23
Object Creation and Destruction
Abstract Interfaces 7
Attributes 9
Basic types 5
Constructed types 6
Interfaces 6
Operations 7
Types 5
Value Types 7
object duplicate operation
OMG PIDL for 107
object identifiers
and hash operation 109
Object Implementation 16
Structure 21
The Construction Model 10
The Execution Model 9
Object interface
create_request operation 106
OMG PIDL for 105
Object model 3
Object reference 4, 16
testing for equivalence 109
Object Request Broker 15
explained 11
how implemented 15
interfaces to 11
sample implementations 19, ??–21
Structure 11
Object Semantics
Objects 4
Requests 4
Object Services
tags for 493
object system 3
object_key 459
object_to_string operation 102
OMG PIDL for 102
object-level policies 131
Objects 4
octet type 62
OMG IDL
overview of 12, 17
syntax of 43
OMG IDL for Interface Repository 284
OMG IDL global name 88
OMG IDL Grammar 37
OMG IDL Specification 43
OMG IDL tags
requests to allocate 493
OMG IDL-to-programming language mapping
overview 17
OMG Interface Definition Language 17
operation 7, 459
attribute, syntax of 72
declaration, syntax of 71
defined 7
oneway 487
signature of 8
Operation Declaration
Context Expressions 74
Operation Attribute 72
Parameter Declarations 72
Raises Expressions 73
operation_name 437, 461
operation_target 436
OperationDef 248
ORB Interface 18
and create_list operation 190
and create_operation_list operation 191
ORB Operations
get_next_response and poll_next_response 185
send_multiple_requests 185
ORB-level policies 131
ORBs
Client- and Implementation-resident ORB 19
connecting 223
Library-based ORB 20
Server-based ORB 19
System-based ORB 20
ORDER_ANY 425
ORDER_DEADLINE 425
ORDER_PRIORITY 425
ORDER_TEMPORAL 425

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Ordering 425
OTS Behavior 490

P
parameter
  defined 8
parameter declaration
  syntax of 72
Persistent ReplyHandler 482
PersistentPoller 482
PersistentRequest 482
PersistentRequest interface 461
PersistentRequestRouter interface 461
POA
  location transparency 312
POA Interface 328
POA Threading Models 310
POAManager Interface 314
POA-related interfaces 312
policies
  object-level 131
  thread-level 131
Policy 479
PolicyCurrent interface 134
PolicyList 479
PolicyManager 479
PolicyManager interface 132
PolicyValue 426
Pollable interface 187
PollableSet interface 188
Poller 437, 481
Poller operations
  for Interface attributes 439
  for Interface operations 438
Poller Value
  generic 436
Poller/PersistentRequest Detailed Design 486
Poller/PersistentRequest detailed design 486
Polling 186
  Abstract Valuetype DIIPollable 188
  Abstract Valuetype Pollable 187
  interface PollableSet 188
Polling Model
  signatures 431
Portable Object Adaptor
  abstract model description 303
  AdaptorActivator interface 319
  creating 306, 329, 347
  creating object references 307
  destroying 330
  dynamic skeleton interface 311
  finding 329
  implicit activation 309
  Implicit Activation policy 328
  interface 328
  model architecture 305
  model components 303
  multi-threading 310
  overview 303
  policy objects 325
  request processing 309
  root POA 346
  ServantActivator interface 321
  ServantLocator Interface 323
  ServantManager interface 320
  SYSTEM_ID policy 347
  usage scenarios 346
PortableServer
  UML description of 344
Pragma 280
pragma directive
  and Interface Repository 277
  id 277
Prefix Pragma 278
preinvoke operation 321
prepare 184
Preprocessing 36
PrimitiveDef 244
  OMG IDL for 244
PriorityRange 421
profile
  tags for 493
profile_index 460, 462

Q
QoS Abstract Model Design 479
qualified name 87
Quality of Service (QoS) 417
QueueOrderPolicy 425
QueueOrderPolicy interface 425

R
REBIND 155
Rebind Support 419
  Interface RebindPolicy 420
  Request and Reply Priority 421
  Request and Reply Timeout 422
  Routing 424
  Synchronization Scope 420
  typedefshortRebindMode 419
RebindMode 419
RebindPolicy interface 420
register_destination 471
Registering Dynamic Implementation Routines 195
RegistrationState 470
RelativeRequestTimeoutPolicy 423
RelativeRoundtripTimeoutPolicy interface 424
release operation 107
remove 189
reply 460
reply_body 461

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

reply_destination 460
reply_type 461
ReplyDestination 459
ReplyEndTimePolicy 423
ReplyEndTimePolicy interfaces 423
ReplyHandler 434, 481, 485
ReplyHandler interface 460
ReplyHandler Operations
  for NO_EXCEPTION Replies 434
ReplyHandler operations
  for exceptional replies 435
  for NO_EXCEPTION replies 434
ReplyHandler Operations for Exceptional Replies 435
ReplyHandler Operations for NO_EXCEPTION
    Replies 434
ReplyPriorityPolicy 422
ReplyStartTimePolicy 423
Repository interface
  OMG IDL for 238, 264
RepositoryId 274
  and pragma directive 277
  DCE UUID Format 277
  format of 275
  For More Information 282
  LOCAL Format 277
  OMG IDL Format 275
  Pragma Directives for RepositoryId 277
  RepositoryIDs for OMG-Specified Types 282
  RMI Hashed Format 275
request context 8
request form 4
Request operations
  add_arg 182
  create_request 180
  delete 183
  get_response 183
  invoke 182
  poll_response 183
  prepare 184
  send 183
  sendc 184
  sendp 184
Request Routing Algorithm 464
Request/Reply Routers 488
RequestEndTimePolicy 422
RequestInfo 459
RequestMessage 459
RequestPriorityPolicy 422
RequestProcessingPolicy 327
reserved 459
response_flags 459
result
  defined 8
resume_destination 472
ResumePolicy 471
RetryPolicy 470
Return Status and Exceptions 179
ROUTE_FORWARD 424
ROUTE_NONE 424
ROUTE_STORE_AND_FORWARD 424
Router Administration 467
  Constants 470
  Exceptions 470
  Interfaces 471
  Values 470
Router interface 460
RouterAdmin interface 471
Routing Object References 456
RoutingPolicy interface 424
RoutingTypeRange 424

S

scoped name identifier 87
scoped_name 48
Security Considerations 175
Security Service 491
selected_qos 460, 462
send_multiple_requests 460
send_request 460
sendc 184, 429
sendp 184, 431
sequence type 64, 66
SequenceDef 245
ServantRetentionPolicy 327
ServerRequest pseudo-object 194
ServerRequest's Handling of Operation Parameters 195
ServerRequestPseudo-Object 194
server-side
  policies 479
  policy management 132
Service Contexts 466
service_contexts 459
set_policy_overrides 133
signature 7
static routing 488
string type 67
string_to_object operation 102
StringDef 244
StructDef 241
Structures
  MessageBody 459
  ReplyDestination 459
  RequestInfo 459
  RequestMessage 459
stub interface 17
subject 75
suspend_destination 471
SUSPENDED 470
SYNC_NONE 421
SYNC_WITH_SERVER 421

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

SYNC_WITH_TARGET 421
SYNC_WITH_TRANSPORT 421
SyncScope 420
SyncScopePolicy 421
SYSTEM_EXCEPTION 466

T
tag
  requests to allocate 493
TAG_POLICIES 479
target 460, 462
Target Router 464
thread-level policies 131
ThreadPolicy 325
TII 488
Time-Independent Invocation 482
TIMEOUT 155, 461
to_visit 460, 462
Transaction service compatibility 490
TRANSACTION_MODE exception 155
TRANSACTION_UNAVAILABLE Exception 155
TRANSPARENT 420
Type 5
  Any Type 6, 62
  Basic Types 5, 60
  Boolean Type 62
  Char Type 61
  Constructed Types 6, 62
  Floating-Point Types 61
  Integer Types 61
  Native Types 68
  Octet Type 62
  Template Types 66
  Wide Char Type 62
type specifier
  syntax of 59
TypeCode constants 142
TypeCode interface 138
TypeCodes 178
  Creating TypeCodes 143
  OMG IDL for 144
  The TypeCode Interface 138
  TypeCode Constants 142
TypedefDef 241
Type-Specific Poller Mapping
  Basic Type-Specific Poller 437
  Persistent Type-Specific Poller 439
  Poller operations for Interface attributes 439
  Poller operations for Interface operations 438
Type-specific ReplyHandler 466
Type-Specific ReplyHandler Mapping 433

U
UML Description of PortableServer 344
union type 63
UnionDef 242
UnlimitedPing 470
unregister_destination 472
UntypedReplyHandler 466
  Replying to an 466
UntypedReplyHandler interface 460
Usage Guidelines 174
Usage Scenarios
  Creating a POA 347
  Creating References before Activation 349
  Explicit Activation with POA-assigned Object Ids 347
  Explicit Activation with User-assigned Object Ids 348
  Getting the Root POA 346
  Multiple Object Ids Mapping to a Single Servant 352
  Object Activation on Demand 351
  One Servant for All Objects 352
  Persistent Objects with POA-assigned Ids 352
  Servant Manager Definition and Creation 349
  Single Servant, Many Objects and Types, Using DSI 355
USER_EXCEPTION 466

V
validate_connection 112
Value 4
  Abstract Value Type 53
  Boxed Value Type 52
  DecayPolicy 471
  ImmediatePing 470
  ImmediateSuspend 470
  LimitedPing 471
  Regular Value Type 50
  ResumePolicy 471
  Type 7, 157
  Value Forward Declaration 53
  Valuetype Inheritance 54
ValueBoxDef 262
ValueDef 257, 260
ValueMemberDef 256
Version Pragma 280
visited 460

W
WstringDef 244

© ISO/IEC 2008 - All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale