IDL/Java Language Mapping

Joint Revised Submission

Digital Equipment Corporation
Expersoft Corporation
Hewlett-Packard Company
IONA Technologies ,Ltd.
International Business Machines Corporation
Netscape Communications Corporation
Novell, Inc.
Oracle Corporation
Sun Microsystems, Inc.
Visigenic Software, Inc.
Xerox Corporation

OMG TC Document orbos/97-03-01
(orbos/97-02-01 with Errata)

March 19, 1997 7:48 pm

Copyright 1997 by Digital Equipment Corporation
Copyright 1997 by Expersoft Corporation
Copyright 1997 by Hewlett-Packard Company
Copyright 1997 by International Business Machines Corporation
Copyright 1997 by IONA Technologies, Ltd.
Copyright 1997 by Netscape Communications, Inc.
Copyright 1997 by Oracle Corporation
Copyright 1997 by Novell, Inc.
Copyright 1997 by Sun Microsystems, Inc.
Copyright 1997 by Visigenic Software, Inc.
Copyright 1997 by Xerox Corporation

The submitting companies listsed above have all contributed to this “merged” submission. These companies recognize that this draft joint submission is the joint intellectual property of all the submitters, and may be used by any of them in the future, regardless of whether they ultimately participate in a final joint submission.

The companies listed above hereby grant a royalty-free license to the Object Management Group, Inc. (OMG) for worldwide distribution of this document or any derivative works thereof, so long as the OMG reproduces the copyright notices and the below paragraphs on all distributed copies.

The material in this document is submitted to the OMG for evaluation. Submission of this document does not represent a commitment to implement any portion of this specification in the products of the submitters.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE,THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The companies listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material. The information contained in this document is subject to change without notice.

This document contains information which is protected by copyright. All Rights Reserved. Except as otherwise provided herein, no part of this work 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 the permission of one of the copyright owners. All copies of this document must include the copyright and other information contained on this page.

The copyright owners grant member companies of the OMG permission to make a limited number of copies of this document (up to fifty copies) for their internal use as part of the OMG evaluation process.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013.

CORBA and Object Request Broker are trademarks of Object Management Group.

OMG is a trademark of Object Management Group.

Table of Contents

1 Preface 7

1.1 Cosubmitting Companies 7
1.2 Guide to the Submission. 7
1.3 Missing Items 8
1.4 Conventions 8
1.5 Submission Contact Points 8

2 Proof of Concept 11

3 Response to RFP Requirements 13

3.1 Map All of IDL. 13
3.2 Map All the PIDL. 13
3.3 Design Rationale 14
3.4 Java Version 14

4 Overall Design Rationale 15

4.1 Introduction 15
4.1.1 Goals. 15
4.1.2 Java Version 16
4.1.3 Reverse mapping issues 16
4.1.4 Java Stub and Skeleton ORB Interfaces. 16
4.2 Names 17
4.3 Mapping of Module 18
4.4 Mapping for Basic Types 18
4.5 Helper Classes 18

4.6 Mapping for Constant . . . 18
4.7 Mapping for Enum . . . 19
4.8 Mapping for Struct . . . 19
4.9 Mapping for Union . . . 19
4.10 Mapping for Sequence . . . 19
4.11 Mapping for Array . . . 19
4.12 Mapping for Interface . . . 19
4.13 Mapping for Exception . . . 20
4.14 Mapping for the Any Type . . . 20
4.15 Mapping for Certain Nested Types . . . 20
4.16 Mapping for Typedef . . . 20
4.17 Pseudo Interface Mapping . . . 20

5 Mapping IDL to Java . . . 23

5.1 Introduction . . . 23
5.2 Names . . . 23
5.2.1 Reserved Names . . . 24
5.3 Mapping of Module . . . 24
5.3.1 Example . . . 24
5.4 Mapping for Basic Types . . . 25
5.4.1 Introduction . . . 25
5.4.2 Boolean . . . 29
5.4.3 Character Types . . . 29
5.4.4 Octet . . . 29
5.4.5 String Types . . . 29
5.4.6 Integer Types . . . 30
5.4.7 Floating Point Types . . . 30
5.4.8 Future Fixed Point Types . . . 30
5.4.9 Future Long Double Types . . . 30
5.5 Helper Classes . . . 30
5.5.1 Examples . . . 32
5.6 Mapping for Constant . . . 32
5.6.1 Constants Within An Interface . . . 32
5.6.2 Constants Not Within An Interface . . . 33
5.7 Mapping for Enum . . . 33
5.7.1 Example . . . 35
5.8 Mapping for Struct . . . 35
5.8.1 Example . . . 36
5.9 Mapping for Union . . . 36
5.9.1 Example . . . 38
5.10 Mapping for Sequence . . . 39
5.10.1 Example . . . 39

5.11 Mapping for Array 40
5.11.1 Example 40
5.12 Mapping for Interface. 41
5.12.1 Basics 41
5.12.2 Parameter Passing Modes. 43
5.13 Mapping for Exception. 44
5.13.1 User Defined Exceptions 45
5.13.2 System Exceptions 46
5.14 Mapping for the Any Type 48
5.15 Mapping for Certain Nested Types 51
5.15.1 Example 51
5.16 Mapping for Typedef 52
5.16.1 Simple IDL types 52
5.16.2 Complex IDL types 52

6 Mapping Pseudo-Objects to Java 53

6.1 Introduction 53
6.1.1 Pseudo Interface 53
6.2 Certain Exceptions 54
6.3 Environment 55
6.4 NamedValue 55
6.5 NVList 56
6.6 ExceptionList 57
6.7 Context 57
6.8 ContextList 58
6.9 Request 59
6.10 ServerRequest and Dynamic Implementation 60
6.11 TypeCode 61
6.12 ORB 65
6.13 CORBA::Object 69
6.14 Current 69
6.15 Principal 70

7 Server-Side Mapping 71

7.1 Introduction 71
7.2 Transient Objects 71
7.2.1 Servant Base Class 71
7.2.2 Servant Class 72
7.2.3 Creating A Transient ORB Object 72
7.2.4 Connecting a Transient ORB Object 72
7.2.5 Disconnecting a Transient ORB Object 72

7.3 Persistent Objects 73

8 Java ORB Portability Interfaces 75

8.1 Introduction 75
8.1.1 Design Goals 75
8.1.2 Portability Package. 76
8.2 Architecture 76
8.3 Streamable APIs 78
8.4 Streaming APIs. 78
8.5 Portability Stub Interfaces 81
8.5.1 Stub Design 81
8.5.2 Portable ObjectImpl 81
8.6 Delegate 83
8.7 Skeleton 83
8.8 ORB Initialization 84
8.8.1 Standard Properties 84
8.8.2 ORB Initialization Methods 85

9 Conformance Issues 87

9.1 Introduction 87
9.2 Compliance. 87

10 Changes to CORBA 2.0 89

10.1 Request PIDL 89
10.2 Certain Exceptions 89

Preface 1

1.1 Cosubmitting Companies

The following companies are pleased to jointly submit this specification in response to the OMG Java Language Mapping RFP (Document ORBOS/96-08-01):

- Digital Equipment Corporation
- Expersoft Corporation
- Hewlett-Packard Company
- International Business Machines Corporation
- IONA Technologies, Ltd.
- Netscape Communications Corporation
- Novell, Inc.
- Oracle Corporation
- Sun Microsystems, Inc.
- Visigenic Software, Inc.
- Xerox Corporation

1.2 Guide to the Submission

This submission specifies a comprehensive approach to mapping IDL to the Java language. Elements include:

- Design Rationale
- Mapping of IDL
- Mapping of Pseudo Objects
- Server-Side Mapping

- Java Stub/Skeleton ORB Interfaces

1.3 Missing Items

The following topics are completely missing from this submission, pending completion and recommendation by the ORBOS Task Force of the Server-side Portability RFP:

- Portable Object Adpator
- possible new PIDL and other required mappings
- dynamic any type management

It is the intention of the submitters to ask the ORBOS Task Force to recommend approval for a possible revised submission and to leave this RFP open for a later submission which addresses these important topics.

Previous drafts of this submission have had a stub and skeleton portability layer that was not based upon the DII and DSI. The short amount of time between the initial and revised submission due dates did not allow for sufficient time to investigate and benchmark various alternatives which, because they would be specially tuned, might be smaller and more efficient for stub based invocations. Since the RFP needs to be left open to deal with the Portability RFP issues, it is the intention of the submitters to request that this RFP also be left open (for the same period of time) to address this topic.

1.4 Conventions

IDL appears using this font.

Java code appears using this font.

Please note that any change bars have no semantic meaning. They show the places that errata were discovered since the last submission (orbos/97-02-01) was submitted to the OMG. They are present for the convenience of readers and submitters (and the editor who didn't want to have to re-edit the entire document to remove change bars and maintain two synchronized copies) so that the final edits can be identified.

1.5 Submission Contact Points

All questions about the joint submission should be directed to:

Dan Frantz
Digital Equipment Corporation
ZK02-2/R80
110 Spitbrook Road
Nashua, NH 03062
USA
phone: +1 603 881 2272
fax: +1 603 881 0120
email: frantz@send.enet.dec.com

Dave Frye
Expersoft Corporation
5825 Oberlin Drive #300
San Diego, CA 92121
USA
phone: +1 619 824 4137
fax: +1 619 824 4110
email: didge@expersoft.com

Randy Schnier
International Business Machines, Inc.
Deptartment 143
3605 Highway 52 North
Rochester, MN
USA
phone: +1 507 253 2565
email: rschnier@vnet.ibm.com

Martin Chapman
IONA Technologies
The IONA Building
8-10 Lower Pembroke Street
Dublin 2, Ireland
phone: +353 1 662 5255
fax: +353 1 662 5244
email: mchapman@iona.com

Peter de Jong
Internet Technology Lab
Hewlett-Packard Company
300 Apollo Drive
Chelmsford, MA 01824
phone: +1 508 436 5372
fax: +1 508 436 5176
email: dejong@apollo.hp.com

Jim Gellman
Netscape Communications Corporation
501 E. Middlefield Road
Mountain View, CA 94043
USA
phone: +1 415 254 1900
email: jgellman@netscape.com

Mark Ericson
Novell, Inc.
122 East 1700 South
M/S C-11-1
Provo, UT 84606-6194
phone: +1 801 429 5486
email: marke@novell.com

Jeff Nisewanger
SunSoft, Inc.
2550 Garcia Avenue

MS UMPK18-209
Mountain View, CA 94043
USA
phone: +1 415 786 4867
fax: +1 415 786 4101
email: jdn@eng.sun.com

Joe Z. Ye
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065
USA
phone: +1 415 506 9332
email: zye@us.oracle.com

Jeff Mischkinsky
Visigenic Software, Inc.
951 Mariner's Island Blvd. Suite 120
San Mateo, CA 94404
USA
phone: +1 415 312 5158
email: jeffm@visigenic.com

Chris Jacobi
Xerox Corporation
3333 Coyote Hill Road
Palo Alto, CA 94304
USA
phone: +1 415 812 4000
email: jacobi@xerox.parc.com

Proof of Concept 2

The specification presented here is based on the extensive experience the submitting companies have had over the past year with their “experimental” mappings. Many of the alternative designs that were considered have actually been implemented and tried by many users. The final choices that are embodied in this submission were made based upon user and vendor experience.

Shipping product which implements this specification can be expected to be made available almost concurrently with its final approval.

Response to RFP Requirements 3

The following is a list of the requirements from the Java Language Mapping RFP (OMG orbos/96-08-01), specifying how this submission is responsive to the RFP.

3.1 Map All of IDL

- *The proposed mapping shall map the entire IDL language and provide access to all the feature of the CORBA. In particular the DII/DSI and server side mapping must be covered.*

The entire IDL language is mapped with the exception of some of the new IDL extended types.

The Java class definitions used to support **wchar** and **wstring** are now stable and are therefore included in this submission, even though they may not be fully implemented in all IDL compilers for some time.

The Java class definitions that would be used to support long double and fixed are neither fully stable, nor have they been widely implemented as yet. We therefore chose to leave these as suggestions for a future revision.

3.2 Map All the PIDL

- *The mapping of PIDL constructs of the CORBA 2.0 specification shall be described and shall be consistent with the IDL mapping.*

The following lists the PIDL that is not mapped by this submission. Conforming implementations may ignore this listed PIDL:

the BOA

The reasons for not mapping include the following. The Portability RFP process is about to deprecate the BOA and replace it with a Portable Object Adapter (POA). Because of the serious shortcomings of the current server side mappings, there is no

reason to force vendors to implement necessarily proprietary mappings. Instead we propose a simple server side mapping for registration of transient objects. The Portability submission process is expected to complete before the close of this RFP process. Hence we expect that the revised submissions will specify a mapping for the new portability features.

3.3 Design Rationale

- *The language mapping should be prefaced by an explanation of the design rationale.*

See the chapter on design rationale

3.4 Java Version

- *The version of the Java language specification to be used is 1.0 Beta, unless superseded by a revised specification issued before the submission is due.*

This specification is based on the version of the Java language (and implementation) as defined and released by JavaSoft as part of the JDK 1.0.2 release.

Overall Design Rationale

This chapter discusses some of the rationale behind the choices that were made for this mapping.

General issues are discussed in the Introduction. The remaining sections parallel the sections in Chapter 5, “Mapping IDL to Java", the actual mapping specification.

4.1 Introduction

This submission strikes a balance between several different and often conflicting design centers.

An assumption is that CORBA is used in a multi-platform world. Distributed clients and servers will be developed using a variety of computing platforms and languages. The common denominator will be the IDL that describes the services being implemented. Consistency and uniformity between the various IDL language mappings will facilitate the learning and use of CORBA. Consequently a conscious attempt has been made not to introduce gratuitous differences in the way standard CORBA IDL identifiers and Services are mapped into Java. E.G., the names of the standard CORBA system exceptions are mapped directly, rather than changing them to be more “Java friendly”.

A strong attempt has also been made to make the mapping seem as natural and convenient to Java programmers, except when there are conflicts with the above two goals. E.g. IDL interface names map directly into Java interface names.

4.1.1 Goals

- Client-side and server-side source code portability
- ORB replaceability
- Binary compatibility between client stubs (and server skeletons) and ORBs.

4.1.2 Java Version

This specification is based upon the version of Java released by JavaSoft as JDK 1.0.2.

It does not rely upon or use features that are just starting to be released as part of the JDK 1.1beta program. While it is attractive to consider using that release because of its larger feature set, the fact remains that it will not be widely available until the end of 1997.

Because JDK 1.1 will be fully upwardly compatible with JDK 1.0.2, no changes to this mapping will be required in order for it be used with JDK1.1. After JDK 1.1 stabilizes and vendors gain some experience with it, the OMG may wish to add extensions to this mapping in order to take advantage of new Java features.

4.1.3 Reverse mapping issues

The submitters feel that it will be desirable in a future OMG RFP, possibly even the pending Call By Value RFP, to be able to implement a so-called reverse mapping from Java to IDL. Using it, it should be possible to generate OMG CORBA IDL that describes a Java interface. A desirable characteristic of such a mapping, is that it be idempotent so that the result of mapping a Java interface to IDL, and then mapping it back to Java (using this specification) results in the same Java interface. This specification was designed so as not to preclude this property.

4.1.4 Java Stub and Skeleton ORB Interfaces

This submission includes a specification for the interfaces between Java stubs and skeletons and the Java ORB runtime. The unique properties of Java and the way it is used within browsers makes the specification of these interfaces interesting and necessary for a number of reasons.

Many Java applications are run as applets in browsers. The applet is usually downloaded over the Internet or a LAN. If the applet communicates with a CORBA object using a static interface, then the stub for invoking operations must be downloaded as part of the applet. Similarly, a developer implementing a server, may wish to download the skeleton for a method. In both cases, it is highly desirable that the required amount of code to be downloaded be kept to a minimum, particularly in the case of a browser. Stubs (and skeletons) generated by compliant IDL compilers (or other compliant tools) which use these interfaces will be able to run in a browser environment which already contains an ORB which is compliant with this specification.

It is expected that support for these standardized interfaces will be “baked into” the popular Internet browsers and the JDK. Thus applets, either pure client or acting themselves as CORBA servers (along with their associated stubs and skeletons), can be created and compiled once, and be guaranteed to run in all these environments without the necessity of further downloads.

As a result of the adoption of this specification, third-party tool vendors will be able to generate applets that will work with any compliant Java ORB. Without this specification, tool vendors would be forced to generate ORB-specific stubs and skeletons for every environment in which they expect their applet to run.

Binary Compatibility

Stubs and skeletons that implement this specification will not need to be recompiled when run with any ORB which supports the Java Stub/Skeleton ORB Interface. Thus CORBA client applets (applets that invoke operations on distributed CORBA objects) can be written, compiled once into Java bytecodes, downloaded, and run in any environment that includes an ORB supporting these interfaces.

Without the specification of these stub/skeleton to ORB interfaces, a CORBA client or server applet is not guaranteed to be compatible with any given Java ORB implementation. Stubs and skeletons that are not compatible with the resident Java ORB (because they use proprietary stub/skeleton ORB interfaces), must first have the compatible ORB downloaded, before they can be used. While in some environments, the time to download (and possibly cache) may be acceptable, in certain increasingly common situations such as those involving Internet browsers, the cost and increased complexity is unacceptable.

This issue is unique to Java, because for other languages (e.g. C++), there is no cross-platform binary compatibility at the language level. For those other languages, source code compatibility at the stub/skeleton signature level is sufficient since, in general, recompilation is necessary in order to move between platforms.

Value Added Extensions

Specification of Java stub/skeleton ORB interfaces does not preclude ORB vendors from continuing to innovate and add value. Vendors are free to extend the stub/skeleton ORB interfaces, and use alternative ORB interfaces (proprietary or ORB-specific) in order to provide functionality not supported by these interfaces. However users will have a choice (with compliant tools) as to whether they wish to use these extensions or not.

The required ORB components can be downloaded if not already present, as would be the case even if these interfaces were not standardized.

4.2 Names

This is the first IDL language mapping submission which explicitly takes into account possible name collisions and provides rules for dealing with them.

The name collision rule was chosen so that it would only affect highly unlikely cases, and not disadvantage most users. We expect that it will be invoked only rarely. The alternative of always pre-pending underscores would have affected all programmers and forced them to use an unnatural mapping in order to avoid uncommon cases.

Collisions will normally only occur when the Java programmer has existing classes which use names corresponding to generated support classes, or are the same as a new IDL interface. Collisions will also occur if using an IDL interface named with a Java reserved word.

The emphasis here is on keeping the client-side mappings simple and natural. Hence the decision not to simply prepend underscores everywhere there would be a possibility of a name clash. On the server-side we use a prepended underscore for such things as the implementation base class. We would also expect that future extensions would continue with this policy.

4.3 Mapping of Module

Modules are a name-scoping mechanism in IDL. The corresponding name-scoping mechanism in Java is the Java package.

4.4 Mapping for Basic Types

OMG *char* could have been mapped to *byte* which would have avoided range checks at runtime. However, it was rejected because

- Java *chars* are more natural to Java programmers.
- According to the Java Language Specification, Section 5.1.3, conversions from *byte* to *char* are narrowing primitive conversions because *byte*s are signed quantities while *char*s are unsigned. Mapping IDL *chars* to *byte* would preclude the safe usage of any of the methods in *java.lang.Character.*

Mapping unsigned integer types to the next larger size does not scale all the way up to *unsigned long long*. Also, that is by no means a specification-preserving mapping. The most intuitive, under these circumstances, is to map IDL integer types to the correspondingly sized Java types.

4.5 Helper Classes

The helper classes have necessary methods that we do not want to appear on the “regular” mapped classes. This is to make reverse mapping simpler.

4.6 Mapping for Constant

Java programmers typically use static final fields in classes or interfaces for constants. Thus, constants in IDL interfaces are best mapped to static final fields in corresponding Java interfaces.

Constants declared at module scope or globally generates a class per IDL constant. **This is done so that global constants may be supported easily.** These classes are only required at compile time since the generated Java bytecode would not depend upon the constant classes at runtime and they can be optimized out by intelligent tools.

4.7 Mapping for Enum

The proposed mapping provides two representations for each case label in order to provide strong type checking as well as allow the use of the enumeration labels in switch statements.

4.8 Mapping for Struct

An alternate mapping was to have instance variables as private fields and have public accessor functions. Since the purpose of IDL structs is to define data that will be directly accessible, we left the fields exposed. Not having accessor methods also implies smaller generated code, and faster and more straightforward access.

4.9 Mapping for Union

An alternative mapping that would ensure strong-typing is to map the union to an abstract class and each of the branches to concrete subclasses. This was rejected because the use requires casts at runtime and there are too many classes generated.

4.10 Mapping for Sequence

Alternative choices included mapping bounded sequences to a class which had methods to access parts of the sequence. Such a mapping would have been specification-preserving as well as safer. However, that would not be natural to Java programmers who are used to accessing array components using the subscript operator ([]). It would also have made reverse mapping much more difficult.

4.11 Mapping for Array

An IDL array is mapped the same way as an IDL bounded sequence. In Java, the natural Java subscripting operator is applied to the mapped array. The bounds for the array are checked when the array is marshaled as an argument to an IDL operation. If you want the length of the array to be available in Java, bound the array with an IDL constant, which will be available through its mapping.

4.12 Mapping for Interface

We considered prepending “get” and “set” to the attribute methods instead of overloading the attribute name. Doing this, however, often generates method names which have awkward capitalization or style. The overloaded style is consistent with the C++ and Smalltalk mappings.

4.13 Mapping for Exception

The exception hierarchies are set up so that only user-defined exceptions have to caught as a (large) convenience to Java programmers.

4.14 Mapping for the Any Type

Since predefined types do not map to Java classes, we felt it was unnecessary to introduce Java classes for predefined types just to hold the insertion and extraction operations. They are more naturally implemented and used by having insertion and extraction operations for predefined types on the Any class itself.

The static (for user-defined types) and non-static (for predefined types) asymmetry was a conscious decision. It was felt that statics on the Any class for insertion and extraction of predefined types would be tedious.

The read and write stream methods are mostly meant to provide support for the Java Stub and Skeleton portable ORB interfaces. It is not envisaged that they will be used by ordinary programmers.

4.15 Mapping for Certain Nested Types

Since the same IDL identifier would be used for both the class name and the package name (and this is illegal in Java) we append Package to use as the package name.

4.16 Mapping for Typedef

Mapping for typedef is somewhat problematic since Java does not support the notion of typedef.

In general, we unwind typedef definitions to their basic types and thus avoid the tricky semantics of defining the exact meaning of the phrase “a typedef introduces a distinct IDL type”.

4.17 Pseudo Interface Mapping

A conscious decision was made to map PIDL to Java public abstract classes that do not inherit nor extend from other classes or interfaces, rather than mapping them to Java interfaces. The primary reason was to support “interface” evolution of PIDL and not sacrifice the goal of allowing ORB implementations to be downloaded into a browser, even if it already has a “baked in” ORB. (The user may need to use a specific ORB implementation in order to take advantage of features that particular ORB implementation supports.)

As OMG specifications evolve, it is quite likely that new methods will be added to existing pseudo objects.If Java interfaces, rather than abstract classes were used, then adding a new method would not allow older ORBs to be downloaded. An example might be adding a new method to the Request pseudo object called useFedEx() to indicate that the request was to be guaranteed to get there by 10:00 AM the next

morning. Let's assume that the next release of a Browser ships with support for this newly updated Request with the new useFedEx() method. Older ORB implementations, because they don't know about and hence don't implement the new method, will not load. By mapping the pseudo object to an abstract class, new methods can be added over time by giving the new methods default implementations which throw a NO_IMPLEMENT Java exception. Newer ORB implementations then override the default implementation with a real implementation. Old ORB implementations (and their users) never called the new method in the first place.

Our judgment is that the benefits of insuring upward compatibility in the face of future evolutionary change far outweighs the cost of mapping to abstract classes instead of interfaces.

Mapping IDL to Java 5

5.1 Introduction

This section describes the complete mapping of IDL into the Java language.

The rationale for design decisions can be found in Chapter 4, “Overall Design Rationale".

In most cases examples of the mapping are provided. It should be noted that the examples are code fragments that try to illustrate only the language construct being described. Normally they will be embedded in some module and hence will be mapped into a Java package.

5.2 Names

In general IDL names and identifiers are mapped to Java names and identifiers with no change. If a name collision could be generated in the mapped Java code, the name collision is resolved by prepending an underscore (_) to the mapped name.

In addition, because of the nature of the Java language, a single IDL construct may be mapped to several (differently named) Java constructs. The “additional” names are constructed by appending a descriptive suffix. For example, the IDL interface **foo** is mapped to the Java interface **foo**, and additional Java classes **fooHelper** and **fooHolder**.

In those exceptional cases that the “additional” names could conflict with other mapped IDL names, the resolution rule described above is applied to the other mapped IDL names. I.e., the naming and use of required “additional” names takes precedence.

For example, an interface whose name is **fooHelper** or **fooHolder** is mapped to **_fooHelper** or **_fooHolder** respectively, regardless of whether an interface named **foo** exists. The helper and holder classes for interface **fooHelper** are named **_fooHelperHelper** and **_fooHelperHolder**.

IDL names that would normally be mapped unchanged to Java identifiers that conflict with Java reserved words will have the collision rule applied.

5.2.1 Reserved Names

The mapping in effect reserves the use of several names for its own purposes. These are:

- The Java class ***<type>*Helper**, where ***<type>*** is the name of IDL user defined type.
- The Java class ***<type>*Holder**, where ***<type>*** is the name of an IDL defined type (with certain exceptions such as typedef aliases).
- The Java classes ***<basicJavaType>*Holder**, where ***<basicJavaType>*** is one of the Java primitive datatypes that is used by one of the IDL basic datatypes (Section 5.4.1.2, “Holder Classes).
- The nested scope Java package name ***<interface>*Package**, where ***<interface>*** is the name of an IDL interface (Section 5.15, “Mapping for Certain Nested Types).
- The keywords in the Java language:

abstract	**default**	**if**	**private**	**throw**
boolean	**do**	**implements**	**protected**	**throws**
break	**double**	**import**	**public**	**transient**
byte	**else**	**instanceof**	**return**	**try**
case	**extends**	**int**	**short**	**void**
catch	**final**	**interface**	**static**	**volatile**
char	**finally**	**long**	**super**	**while**
class	**float**	**native**	**switch**	
const	**for**	**new**	**synchronized**	
continue	**goto**	**package**	**this**	

The use of any of these names for a user defined IDL type or interface (assuming it is also a legal IDL name) will result in the mapped name having an (_) prepended.

5.3 Mapping of Module

An IDL module is mapped to a Java package with the same name. All IDL type declarations within the module are mapped to corresponding Java class or interface declarations within the generated package.

IDL declarations not enclosed in any modules are mapped into the (unnamed) Java global scope.

5.3.1 Example

```
// IDL
module Example {...}
```

```
// generated Java
package Example;
    ...
```

5.4 Mapping for Basic Types

5.4.1 Introduction

The following table shows the basic mapping. In some cases where there is a potential mismatch between an IDL type and its mapped Java type, the Exceptions column lists the standard CORBA exceptions that may be (or is) raised. See Section 5.13, “Mapping for Exception for details on how IDL system exceptions are mapped.

The potential mismatch can occur when the range of the Java type is “larger” than IDL. The value must be effectively checked at runtime when it is marshaled as an in parameter (or on input for an inout), e.g. Java chars are a superset of IDL chars.

Users should be careful when using unsigned types in Java. Because there is no support in the Java language for unsigned types, a user is responsible for ensuring that large unsigned IDL type values are handled correctly as negative integers in Java.

Figure 5-1 BASIC TYPE MAPPINGS

IDL Type	Java type	Exceptions
boolean	**boolean**	
char	**char**	**CORBA::DATA_CONVERSION**
wchar	**char**	
octet	**byte**	
string	**java.lang.String**	**CORBA::MARSHAL** **CORBA::DATA_CONVERSION**
wstring	**java.lang.String**	**CORBA::MARSHAL**
short	**short**	
unsigned short	**short**	
long	**int**	
unsigned long	**int**	
long long	**long**	
unsigned long long	**long**	
float	**float**	
double	**double**	

Additional details are described in the sections following.

5.4.1.1 Future Support

In the future we expect the "new" extended IDL types fixed, and possibly long double, to be supported directly by Java. Currently there is no support for them in JDK 1.0.2, and as a practical matter, they are not yet widely supported by ORB vendors. We would expect them to be mapped as follows:

IDL Type	Java type	Exceptions
long double	**???**	
fixed	**java.math.BigDecimal**	**CORBA::DATA_CONVERSION**

A future revision of this specification should make support of this mapping normative.

5.4.1.2 Holder Classes

Support for out and inout parameter passing modes requires the use of additional "holder" classes. These classes are available for all of the basic IDL datatypes in the **org.omg.CORBA** package and are generated for all named user defined types except those defined by typedefs.

For user defined IDL types, the holder class name is constructed by appending **Holder** to the mapped (Java) name of the type.

For the basic IDL datatypes, the holder class name is the Java type name (with its initial letter capitalized) to which the datatype is mapped with an appended **Holder**, e.g. **IntHolder**. See Section 5.2, "Names" for a description of the implications on possible collisions with user defined names.

Each holder class has a constructor from an instance, a default constructor, and has a public instance member, **value**, which is the typed value. The default constructor sets the value field to the default value for the type as defined by the Java language: **false** for boolean, **0** for numeric and char types, **null** for strings, null for object references.

In order to support portable stubs and skeletons, holder classes for user defined types also have to implement the **org.omg.CORBA.portable.Streamable** interface.

The holder classes for the basic types are defined below. Note that they do not implement the **Streamable** interface. They are in the **org.omg.CORBA** package.

```
// Java

package org.omg.CORBA;

final public class ShortHolder {
    public short value;
    public ShortHolder() {}
    public ShortHolder(short initial) {
        value = initial;
    }
}
```

```
final public class IntHolder {
    public int value;
    public IntHolder() {}
    public IntHolder(int initial) {
        value = initial;
    }
}

final public class LongHolder {
    public long value;
    public LongHolder() {}
    public LongHolder(long initial) {
        value = initial;
    }
}

final public class ByteHolder {
    public byte value;
    public ByteHolder() {}
    public ByteHolder(byte initial) {
        value = initial;
    }
}

final public class FloatHolder {
    public float value;
    public FloatHolder() {}
    public FloatHolder(float initial) {
        value = initial;
    }
}

final public class DoubleHolder {
    public double value;
    public DoubleHolder() {}
    public DoubleHolder(double initial) {
        value = initial;
    }
}

final public class CharHolder {
    public char value;
    public CharHolder() {}
    public CharHolder(char initial) {
        value = initial;
    }
}
```

```
final public class BooleanHolder {
    public boolean value;
    public BooleanHolder() {}
    public BooleanHolder(boolean initial) {
        value = initial;
    }
}

final public class StringHolder {
    public java.lang.String value;
    public StringHolder() {}
    public StringHolder(java.lang.String initial) {
        value = initial;
    }
}

final public class ObjectHolder {
    public org.omg.CORBA.Object value;
    public ObjectHolder() {}
    public ObjectHolder(org.omg.CORBA.Object initial) {
        value = initial;
    }
}

final public class AnyHolder {
    public Any value;
    public AnyHolder() {}
    public AnyHolder(Any initial) {
        value = initial;
    }
}

final public class TypeCodeHolder {
    public TypeCode value;
    public typeCodeHolder() {}
    public TypeCodeHolder(TypeCode initial) {
        value = initial;
    }
}

final public class PrincipalHolder {
    public Principal value;
    public PrincipalHolder() {}
    public PrincipalHolder(Principal initial) {
        value = initial;
    }
}
```

The Holder class for a user defined type <foo> is shown below:

```
// Java
final public class <foo>Holder
    implements org.omg.CORBA.portable.Streamable {

    public <foo> value;
    public <foo>Holder() {}
    public <foo>Holder(<foo> initial) {}
    public void _read(org.omg.CORBA.portable.InputStream i)
        {...}
    public void _write(org.omg.CORBA.portable.OutputStream o)
        {...}
    public org.omg.CORBA.TypeCode _type() {...}
    }
```

5.4.1.3 *Use of Java* **null**

The Java **null** may only be used to represent the "null" object reference. For example, a zero length string, rather than **null** must be used to represent the empty string. Similarly for arrays.

5.4.2 *Boolean*

The IDL boolean constants **TRUE** and **FALSE** are mapped to the corresponding Java boolean literals **true** and **false**.

5.4.3 *Character Types*

IDL characters are 8-bit quantities representing elements of a character set while Java characters are 16-bit unsigned quantities representing Unicode characters. In order to enforce type-safety, the Java CORBA runtime asserts range validity of all Java **chars** mapped from IDL **chars** when parameters are marshaled during method invocation. If the **char** falls outside the range defined by the character set, a **CORBA::DATA_CONVERSION** exception shall be thrown.

The IDL **wchar** maps to the Java primitive type **char**.

5.4.4 *Octet*

The IDL type **octet**, an 8-bit quantity, is mapped to the Java type **byte**.

5.4.5 *String Types*

The IDL **string**, both bounded and unbounded variants, are mapped to **java.lang.String**. Range checking for characters in the string as well as bounds checking of the string shall be done at marshal time. Character range violations cause a **CORBA::DATA_CONVERSION** exception to be raised. Bounds violations cause a **CORBA:: MARSHAL** exception to be raised.

The IDL **wstring**, both bounded and unbounded variants, are mapped to **java.lang.String**. Bounds checking of the string shall be done at marshal time. Bounds violations cause a **CORBA:: MARSHAL** exception to be raised.

5.4.6 Integer Types

The integer types map as shown in Figure 5-1.

5.4.7 Floating Point Types

The IDL float and double map as shown in Figure 5-1.

5.4.8 Future Fixed Point Types

The IDL **fixed** type is mapped to the Java **java.math.BigDecimal** class. Size violations raises a **CORBA::DATA_CONVERSION** exception.

This is left for a future revision.

5.4.9 Future Long Double Types

There is no current support in Java for the IDL **long double** type. It is not clear at this point whether and when this type will be added either as a primitive type, or as a new package in **java.math.***, possibly as **java.math.BigFloat**.

This is left for a future revision.

5.5 Helper Classes

All user defined IDL types have an additional “helper” Java class with the suffix **Helper** appended to the type name generated. Several static methods needed to manipulate the type are supplied. These include **Any** insert and extract operations for the type, getting the repository id, getting the typecode, and reading and writing the type from and to a stream.

For any user defined IDL type, <**typename**>, the following is the Java code generated for the type. In addition, the helper class for a mapped IDL interface also has a narrow operation defined for it.

```
// generated Java helper

public class <typename>Helper {
    public static void
        insert(org.omg.CORBA.Any a, <typename> t) {...}
    public static <typename> extract(Any a) {...}
    public static org.omg.CORBA.TypeCode type() {...}
    public static String id() {...}
    public static <typename> read(
                    org.omg.CORBA.portable.InputStream istream)
        {...}
    public static void write(
                    org.omg.CORBA.portable.OutputStream ostream,
                    <typename> value)
        {...}

    // only for interface helpers
    public static
        <typename> narrow(org.omg.CORBA.Object obj);
}
```

The helper class associated with an IDL interface also has the narrow method (see Section 5.12, “Mapping for Interface).

5.5.1 Examples

```
// IDL - named type
struct st {long f1; string f2;};

// generated Java
public class stHelper {
    public static void insert(org.omg.CORBA.Any any,
        st s) {...}
    public static st extract(Any a) {...}
    public static org.omg.CORBA.TypeCode type() {...}
    public static String id() {...}
    public static st read(org.omg.CORBA.InputStream is) {...}
    public static void write(org.omg.CORBA.OutputStream os,
        st s) {...}
}

// IDL - typedef sequence
typedef sequence <long> IntSeq;

// generated Java helper

public class IntSeqHelper {
    public static void insert(org.omg.CORBA.Any any,
                                        int[] seq);
    public static int[] extract(Any a){...}
    public static org.omg.CORBA.TypeCode type(){...}
    public static String id(){...}
    public static int[] read(
                                org.omg.CORBA.portable.InputStream is)
        {...}
    public static void write(
                                org.omg.CORBA.portable.OutputStream os,
                                int[] seq)
        {...}
}
```

5.6 Mapping for Constant

Constants are mapped differently depending upon the scope in which they appear.

5.6.1 Constants Within An Interface

Constants declared within an IDL interface are mapped to **public static final** fields in the Java interface corresponding to the IDL interface.

5.6.1.1 Example

```
// IDL

module Example {
        interface Face {
        const long aLongerOne = -321;
    };
};

// generated Java

package Example;
public interface Face {
    public static final int aLongerOne = (int) (-321L);
}
```

5.6.2 Constants Not Within An Interface

Constants not declared within an IDL interface are mapped to a **public interface** with the same name as the constant and containing a **public static final** field, named **value**, that holds the contant's value. Note that the Java compiler will normally inline the value when the class is used in other Java code.

5.6.2.1 Example

```
// IDL

module Example {
    const long aLongOne = -123;
};

package Example;
public interface aLongOne {
    public static final int value = (int) (-123L);
}
```

5.7 Mapping for Enum

An IDL **enum** is mapped to a Java **final class** with the same name as the enum type which declares a value method, two static data members per label, an integer conversion method, and a private constructor as follows:

```
// generated Java

public final class <enum_name> {

    // one pair for each label in the enum
    public static final int _<label> = <value>;
    public static final <enum_name> <label> =
        new <enum_name>(_<label>);

    public int value() {...}

    // get enum with specified value
    public static <enum_name> from_int(int value);

    // constructor
    private <enum_name>(int) { ... }

}
```

One of the members is a **public static final** that has the same name as the IDL enum label. The other has an underscore (_) prepended and is intended to be used in switch statements.

The value method returns the integer value. Values are assigned sequentially starting with 0. Note: there is no conflict with the **value()** method in Java even if there is a label named value

There shall be only one instance of an enum. Since there is only one instance, equality tests will work correctly. I.E. the default java.lang.Object implementation of **equals()** and **hash()** will automatically work correctly for an enum's singleton object.

The Java class for the enum has an additional method **from_int()**, which returns the enum with the specified value.

The holder class for the enum is also generated. Its name is the enum's mapped Java classname with **Holder** appended to it as follows:

```
public class <enum_name>Holder implements
        org.omg.CORBA.portable.Streamable {
    public <enum_name> value;
    public <enum_name>Holder() {}
    public <enum_name>Holder(<enum_name> initial) {...}
    public void _read(org.omg.CORBA.portable.InputStream i)
        {...}
    public void _write(org.omg.CORBA.portable.OutputStream o)
        {...}
    public org.omg.CORBA.TypeCode _type() {...}
}
```

5.7.1 Example

```
// IDL
enum EnumType {a, b, c};

// generated Java

public final class EnumType {

    public static final int _a = 0;
    public static final EnumType a = new EnumType(_a);

    public static final int _b = 1;
    public static final EnumType b = new EnumType(_b);

    public static final int _c = 2;
    public static final EnumType c = new EnumType(_c);

    public int value() {...}
    public static EnumType from_int(int value) {...};

    // constructor
    private EnumType(int) {...}

    };
```

5.8 Mapping for Struct

An IDL **struct** is mapped to a final Java class with the same name that provides instance variables for the fields in IDL member ordering and a constructor for all values. A null constructor is also provided so that the fields can be filled in later.

The holder class for the struct is also generated. Its name is the struct's mapped Java classname with **Holder** appended to it as follows:

```
final public class <class>Holder implements
        org.omg.CORBA.portable.Streamable {
    public <class> value;
    public <class>Holder() {}
    public <class>Holder(<class> initial) {...}
    public void _read(org.omg.CORBA.portable.InputStream i)
        {...}
    public void _write(org.omg.CORBA.portable.OutputStream o)
        {...}
    public org.omg.CORBA.TypeCode _type() {...}
}
```

5.8.1 Example

```
// IDL
struct StructType {
    long field1;
    string field2;
};

// generated Java
final public class StructType {
    // instance variables
    public int field1;
    public String field2;
    // constructors
    public StructType() {}
    public StructType(int field1, String field2)
        {...}
    }

final public class StructTypeHolder
        implements org.omg.CORBA.portable.Streamable {
    public StructType value;
    public StructTypeHolder() {}
    public StructTypeHolder(StructType initial) {...}
    public void _read(org.omg.CORBA.portable.InputStream i)
        {...}
    public void _write(org.omg.CORBA.portable.OutputStream o)
        {...}
    public org.omg.CORBA.TypeCode _type() {...}
```

5.9 Mapping for Union

An IDL **union** is mapped to a final Java class with the same name that has:

- a default constructor
- an accessor method for the discriminator, named **discriminator()**

- an accessor method for each branch
- a modifier method for branch
- a modifier method for each branch which has more than one case label.
- a default modifier method if needed

The normal name conflict resolution rule is used (prepend an "_") for the discriminator if there is a name clash with the mapped uniontype name or any of the field names.

The branch accessor and modifier methods are overloaded and named after the branch. Accessor methods shall raise the **CORBA::BAD_OPERATION** system exception if the expected branch has not been set.

If there is more than one case label corresponding to a branch, the simple modifier method for that branch sets the discriminant to the value of the first case label. In addition, an extra modifier method which takes an explicit discriminator parameter is generated.

If the branch corresponds to the **default** case label, then the modifier method sets the discriminant to a value that does not match any other case labels.

It is illegal to specify a union with a default case label if the set of case labels completely covers the possible values for the discriminant. It is the responsibility of the Java code generator (e.g., the IDL complier, or other tool) to detect this situation and refuse to generate illegal code.

A default modifier method, named **default()** (**_default()** if name conflict) is created if there is no explicit default case label, and the set of case labels does not completely cover the possible values of the discriminant. It will set the value of the union to be an out-of-range value.

The holder class for the union is also generated. Its name is the union's mapped Java classname with **Holder** appended to it as follows:

```
final public class <union_class>Holder
            implements org.omg.CORBA.portable.Streamable {
    public <union_class> value;
    public <union_class>Holder() {}
    public <union_class>Holder(<union_class> initial) {...}
    public void _read(org.omg.CORBA.portable.InputStream i)
        {...}
    public void _write(org.omg.CORBA.portable.OutputStream o)
        {...}
    public org.omg.CORBA.TypeCode _type() {...}
}
```

5.9.1 Example

```
// IDL
union UnionType switch (EnumType) {
        case first: long win;
        case second: short place;
        case third:
        case fourth: octet show;
        default:    boolean other;
};

// generated Java
final public class UnionType {
    // constructor
    public UnionType() {....}

    // discriminator accessor
    public <switch-type> discriminator() {....}

    // win
    public int win() {....}
    public void win(int value) {....}

    // place
    public short place() {....}
    public void place(short value) {....}

    // show
    public byte show() {....}
    public void show(byte value) {....}
    public void show(int discriminator, byte value){....}

    // other
    public boolean other() {....}
    public void other(boolean value) {....}

}

final public class UnionTypeHolder
        implements org.omg.CORBA.portable.Streamable {
    public UnionType value;
    public UnionTypeHolder() {}
    public UnionTypeHolder(UnionType initial) {...}
    public void _read(org.omg.CORBA.portable.InputStream i)
        {...}
    public void _write(org.omg.CORBA.portable.OutputStream o)
        {...}
    public org.omg.CORBA.TypeCode _type() {...}
}
```

5.10 Mapping for Sequence

An IDL **sequence** is mapped to a Java array with the same name. In the mapping, everywhere the sequence type is needed, an array of the mapped type of the sequence element is used. Bounds checking shall be done on bounded sequences when they are marshaled as parameters to IDL operations, and an IDL **CORBA::MARSHAL** is raised if necessary.

The holder class for the sequence is also generated. Its name is the sequence's mapped Java classname with **Holder** appended to it as follows:

```
final public class <sequence_class>Holder {
    public <sequence_element_type>[] value;
    public <sequence_class>Holder() {};
    public <sequence_class>Holder(
                            <sequence_element_type>[] initial) {...};
    public void _read(org.omg.CORBA.portable.InputStream i)
        {...}
    public void _write(org.omg.CORBA.portable.OutputStream o)
        {...}
    public org.omg.CORBA.TypeCode _type() {...}
}
```

5.10.1 Example

```
// IDL
typedef sequence< long > UnboundedData;
typedef sequence< long, 42 > BoundedData;

// generated Java

final public class UnboundedDataHolder
            implements org.omg.CORBA.portable.Streamable {
    public int[] value;
    public UnboundedDataHolder() {};
    public UnboundedDataHolder(int[] initial) {...};
    public void _read(org.omg.CORBA.portable.InputStream i)
        {...}
    public void _write(org.omg.CORBA.portable.OutputStream o)
        {...}
    public org.omg.CORBA.TypeCode _type() {...}
}
```

```
final public class BoundedDataHolder
                implements org.omg.CORBA.portable.Streamable {
    public int[] value;
    public BoundedDataHolder() {};
    public BoundedDataHolder(int[] initial) {...};
    public void _read(org.omg.CORBA.portable.InputStream i)
        {...}
    public void _write(org.omg.CORBA.portable.OutputStream o)
        {...}
    public org.omg.CORBA.TypeCode _type() {...}
}
```

5.11 Mapping for Array

An IDL array is mapped the same way as an IDL bounded sequence. In the mapping, everywhere the array type is needed, an array of the mapped type of the array element is used. In Java, the natural Java subscripting operator is applied to the mapped array. The bounds for the array are checked when the array is marshaled as an argument to an IDL operation and a **CORBA::MARSHAL** exception is raised if an bounds violation occurs. The length of the array can be made available in Java, by bounding the array with an IDL constant, which will be mapped as per the rules for constants.

The holder class for the array is also generated. Its name is the array's mapped Java classname with **Holder** appended to it as follows:

```
final public class <array_class>Holder
        implements org.omg.CORBA.portable.Streamable {
    public <array_element_type>[] value;
    public <array_class>Holder() {}
    public <array_class>Holder(
                        <array_element_type>[] initial) {...}
    public void _read(org.omg.CORBA.portable.InputStream i)
        {...}
    public void _write(org.omg.CORBA.portable.OutputStream o)
        {...}
    public org.omg.CORBA.TypeCode _type() {...}
}
```

5.11.1 Example

```
// IDL

const long ArrayBound = 42;
typedef long larray[ArrayBound];
```

```
// generated Java

final public class larrayHolder
        implements org.omg.CORBA.portable.Streamable {
    public int[] value;
    public larrayHolder() {}
    public larrayHolder(int[] initial) {...}
    public void _read(org.omg.CORBA.portable.InputStream i)
        {...}
    public void _write(org.omg.CORBA.portable.OutputStream o)
        {...}
    public org.omg.CORBA.TypeCode _type() {...}
}
```

5.12 Mapping for Interface

5.12.1 Basics

An IDL **interface** is mapped to a public Java interface with the same name, and an additional "helper" Java class with the suffix **Helper** appended to the interface name. The Java interface extends the (mapped) base **org.omg.CORBA.Object** interface.

The Java interface contains the mapped operation signatures. Methods can be invoked on an object reference to this interface.

The helper class holds a static narrow method that allows a **org.omg.CORBA.Object** to be narrowed to the object reference of a more specific type. The IDL exception **CORBA::BAD_PARAM** is thrown if the narrow fails.

There are no special "nil" object references. Java **null** can be passed freely wherever an object reference is expected.

Attributes are mapped to a pair of Java accessor and modifier methods. These methods have the same name as the IDL attribute and are overloaded. There is no modifier method for IDL **readonly** attributes.

The holder class for the interface is also generated. Its name is the interface's mapped Java classname with **Holder** appended to it as follows:

```
final public class <interface_class>Holder
        implements org.omg.CORBA.portable.Streamable {
    public <interface_class> value;
    public <interface_class>Holder() {}
    public <interface_class>Holder(
                        <interface_class> initial) {
        value = initial;
    public void _read(org.omg.CORBA.portable.InputStream i)
        {...}
    public void _write(org.omg.CORBA.portable.OutputStream o)
        {...}
    public org.omg.CORBA.TypeCode _type() {...}
}
```

Interface inheritance expressed in IDL is reflected directly in the Java interface hierarchy.

5.12.1.1 Example

```
// IDL

module Example {
    interface Face {
        long method (in long arg) raises (e);
        attribute long assignable;
        readonly attribute long nonassignable;
    }
}

// generated Java

package Example;

public interface Face extends org.omg.CORBA.Object {
    int method(int arg)
        throws Example.e;
    int assignable();
    void assignable(int i);
    int nonassignable();
}

public class FaceHelper {

    // ... other standard helper methods

    public static Face narrow(org.omg.CORBA.Object obj)
        {...}
}
```

```
final public class FaceHolder
        implements org.omg.CORBA.portable.Streamable {
    public Face value;
    public FaceHolder() {}
    public FaceHolder(Face initial) {...}
    public void _read(org.omg.CORBA.portable.InputStream i)
        {...}
    public void _write(org.omg.CORBA.portable.OutputStream o)
        {...}
    public org.omg.CORBA.TypeCode _type() {...}
}
```

5.12.2 Parameter Passing Modes

IDL **in** parameters which implement call-by-value semantics, are mapped to normal Java actual parameters. The results of IDL operations are returned as the result of the corresponding Java method.

IDL **out** and **inout** parameters, which implement call-by-result and call-by-value/result semantics, cannot be mapped directly into the Java parameter passing mechanism. This mapping defines additional holder classes for all the IDL basic and user-defined types which are used to implement these parameter modes in Java. The client supplies an instance of the appropriate holder Java class that is passed (by value) for each IDL out or inout parameter. The contents of the holder instance (but not the instance itself) are modified by the invocation, and the client uses the (possibly) changed contents after the invocation returns.

5.12.2.1 Example

```
// IDL

module Example {
    interface Modes {
        long operation(in long inArg,
                       out long outArg,
                       inout long inoutArg);
    };
};
```

```
// Generated Java

package Example;

public interface Modes {
    int operation(int inArg,
                  IntHolder outArg,
                  IntHolder inoutArg);
}
```

In the above, the result comes back as an ordinary result and the actual in parameters only an ordinary value. But for the out and inout parameters, an appropriate holder must be constructed. A typical use case might look as follows:

```
// user Java code

// select a target object
Example.Modes target = ...;

// get the in actual value
int inArg = 57;

// prepare to receive out
IntHolder outHolder = new IntHolder();

// set up the in side of the inout
IntHolder inoutHolder = new IntHolder(131);

// make the invocation
int result =target.operation(inArg, outHolder, inoutHolder);

// use the value of the outHolder
... outHolder.value ...

// use the value of the inoutHolder
... inoutHolder.value ...
```

Before the invocation, the input value of the inout parameter must be set in the holder instance that will be the actual parameter. The inout holder can be filled in either by constructing a new holder from a value, or by assigning to the value of an existing holder of the appropriate type. After the invocation, the client uses the outHolder.value to access the value of the out parameter, and the inoutHolder.value to access the output value of the inout parameter. The return result of the IDL operation is available as the result of the invocation.

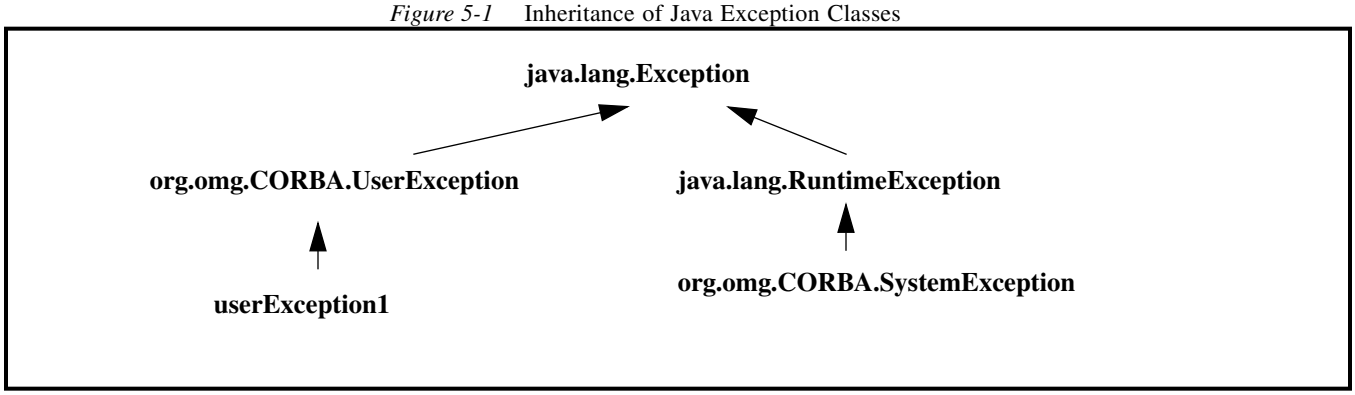
5.13 Mapping for Exception

IDL exceptions are mapped very similarly to structs. They are mapped to a Java class that provides instance variables for the fields of the exception and constructors.

CORBA system exceptions are unchecked exceptions. They inherit (indirectly) from **java.lang.RuntimeException**.

User defined exceptions are checked exceptions. They inherit (indirectly) from **java.lang.Exception**

Figure 5-1 Inheritance of Java Exception Classes



5.13.1 User Defined Exceptions

User defined exceptions are mapped to final Java classes that extend **org.omg.CORBA.UserException** and are otherwise mapped just like the IDL **struct** type, including the generation of Helper and Holder classes.

If the exception is defined within a nested IDL scope (essentially within an interface) then its Java class name is defined within a special scope. See Section 5.15, “Mapping for Certain Nested Types for more details. Otherwise its Java class name is defined within the scope of the Java package that corresponds to the exception’s enclosing IDL module.

5.13.1.1 Example

```
// IDL

module Example {
    exception ex1 { string reason; };
};

// Generated Java

package Example;
final public class ex1 extends org.omg.CORBA.UserException {
    public String reason;                    // instance
    public ex1() {...}                       // default constructor
    public ex1(String r) {...}               // constructor
}
```

```
final public class ex1Holder
        implements org.omg.CORBA.portable.Streamable {
    public ex1 value;
    public ex1Holder() {}
    public ex1Holder(ex1 initial) {...}
    public void _read(org.omg.CORBA.portable.InputStream i)
        {...}
    public void _write(org.omg.CORBA.portable.OutputStream o)
        {...}
    public org.omg.CORBA.TypeCode _type() {...}
}
```

5.13.2 System Exceptions

The standard IDL system exceptions are mapped to final Java classes that extend **org.omg.CORBA.SystemException** and provide access to the IDL major and minor exception code, as well as a string describing the reason for the exception.Note there are no public constructors for **org.omg.CORBA.SystemException**; only classes that extend it can be instantiated.

The Java class name for each standard IDL exception is the same as its IDL name and is declared to be in the **org.omg.CORBA** package. The default constructor supplies 0 for the minor code, COMPLETED_NO for the completion code, and "" for the reason string. There is also a constructor which takes the reason and uses defaults for the other fields, as well as one which requires all three parameters to be specified. The mapping from IDL name to Java class name is listed in the table below:

Table 5-1 Mapping of IDL Standard Exceptions

IDL Exception	Java Class Name
CORBA::UNKNOWN	org.omg.CORBA.UNKNOWN
CORBA::BAD_PARAM	org.omg.CORBA.BAD_PARAM
CORBA::NO_MEMORY	org.omg.CORBA.NO_MEMORY
CORBA::IMP_LIMIT	org.omg.CORBA.IMP_LIMIT
CORBA::COMM_FAILURE	org.omg.CORBA.COMM_FAILURE
CORBA::INV_OBJREF	org.omg.CORBA.INV_OBJREF
CORBA::NO_PERMISSION	org.omg.CORBA.NO_PERMISSION
CORBA::INTERNAL	org.omg.CORBA.INTERNAL
CORBA::MARSHAL	org.omg.CORBA.MARSHAL
CORBA::INITIALIZE	org.omg.CORBA.INITIALIZE
CORBA::NO_IMPLEMENT	org.omg.CORBA.NO_IMPLEMENT
CORBA::BAD_TYPECODE	org.omg.CORBA.BAD_TYPECODE
CORBA::BAD_OPERATION	org.omg.CORBA.BAD_OPERATION

Table 5-1 Mapping of IDL Standard Exceptions

IDL Exception	Java Class Name
CORBA::NO_RESOURCES	org.omg.CORBA.NO_RESOURCES
CORBA::NO_RESPONSE	org.omg.CORBA.NO_RESPONSE
CORBA::PERSIST_STORE	org.omg.CORBA.PERSIST_STORE
CORBA::BAD_INV_ORDER	org.omg.CORBA.BAD_INV_ORDER
CORBA::TRANSIENT	org.omg.CORBA.TRANSIENT
CORBA::FREE_MEM	org.omg.CORBA.FREE_MEM
CORBA::INV_IDENT	org.omg.CORBA.INV_IDENT
CORBA::INV_FLAG	org.omg.CORBA.INV_FLAG
CORBA::INTF_REPOS	org.omg.CORBA.INTF_REPOS
CORBA::BAD_CONTEXT	org.omg.CORBA.BAD_CONTEXT
CORBA::OBJ_ADAPTER	org.omg.CORBA.OBJ_ADAPTER
CORBA::DATA_CONVERSION	org.omg.CORBA.DATA_CONVERSION
CORBA::OBJECT_NOT_EXIST	org.omg.CORBA.OBJECT_NOT_EXIST
CORBA::TRANSACTIONREQUIRED	org.omg.CORBA.TRANSACTIONREQUIRED
CORBA::TRANSACTIONROLLEDBACK	org.omg.CORBA.TRANSACTIONROLLEDBACK
CORBA::INVALIDTRANSACTION	org.omg.CORBA.INVALIDTRANSACTION

The definitions of the relevant classes are specified below.

```
// from org.omg.CORBA package

package org.omg.CORBA;

public final class CompletionStatus {
    // Completion Status constants
    public static final int _COMPLETED_YES = 0,
                                        _COMPLETED_NO = 1,
                                        _COMPLETED_MAYBE = 2;
    public static final CompletionStatus COMPLETED_YES =
        new CompletionStatus(_COMPLETED_YES);
    public static final CompletionStatus COMPLETED_NO =
        new CompletionStatus(_COMPLETED_NO);
    public static final CompletionStatus COMPLETED_MAYBE =
        new CompletionStatus(_COMPLETED_MAYBE);
    public int value() {...}
    public static final CompletionStatus from_int(int) {...}
    private CompletionStatus(int) {...}
}
```

```
abstract public class
    SystemException extends java.lang.RuntimeException {
        public int minor;
        public CompletionStatus completed;
        // constructor
        protected SystemException(String reason,
                                  int minor,
                                  CompletionStatus status) {
            super(reason);
            this.minor = minor;
            this.status = status;
        }
    }

final public class
    UNKNOWN extends org.omg.CORBA.SystemException {
    public UNKNOWN() ...
    public UNKNOWN(int minor, CompletionStatus completed) ...
    public UNKNOWN(String reason) ...
    public UNKNOWN(String reason, int minor,
                               CompletionStatus completed)...
}

...

// there is a similar definition for each of the standard
// IDL system exceptions listed in the table above
```

5.14 Mapping for the Any Type

The IDL type **Any** maps to the Java class **org.omg.CORBA.Any**. This class has all the necessary methods to insert and extract instances of predefined types. If the extraction operations have a mismatched type, the **CORBA::BAD_OPERATION** exception is raised.

In addition, insert and extract methods which take a holder class are defined in order to provide a high speed interface for use by portable stubs and skeletons. There is an insert and extract method defined for each primitive IDL type as well as a pair for a generic streamable to handle the case of non-prmitive IDL types. Note that to preserve unsigned type information unsigned methods (which use the normal holder class) are defined where appropriate.

The insert operations set the specified value and reset the any's type if necessary.

Setting the typecode via the **type()** accessor wipes out the value. An attempt to extract before the value is set will result in a **CORBA::BAD_OPERATION** exception being raised. This operation is provided primarily so that the type may be set properly for IDL **out** parameters.

```
package org.omg.CORBA;

abstract public class Any {

abstract public boolean equal(org.omg.CORBA.Any a);

// type code accessors
abstract public org.omg.CORBA.TypeCode type();
abstract public void type(org.omg.CORBA.TypeCode t);

// read and write values to/from streams
//          throw excep when typecode inconsistent with value
abstract public void read_value(
      org.omg.CORBA.portable.InputStream is,
      org.omg.CORBA.TypeCode t) throws org.omg.CORBA.MARSHAL;
abstract public void
      write_value(org.omg.CORBA.portable.OutputStream os);

abstract public org.omg.CORBA.portable.OutputStream
                create_output_stream();
abstract public org.omg.CORBA.portable.InputStream
                create_input_stream();

// insert and extract each primitive type

abstract public short               extract_short()
      throws org.omg.CORBA.BAD_OPERATION;
abstract public void                insert_short(short s);

abstract public int                 extract_long()
      throws org.omg.CORBA.BAD_OPERATION;
abstract public void                insert_long(int i);

abstract public long                extract_longlong()
       throws org.omg.CORBA.BAD_OPERATION;
abstract public void                insert_longlong(long l);

abstract public short               extract_ushort()
      throws org.omg.CORBA.BAD_OPERATION;
abstract public void                insert_ushort(short s);

abstract public int                 extract_ulong()
      throws org.omg.CORBA.BAD_OPERATION;
abstract public void                insert_ulong(int i);

abstract public long                extract_ulonglong()
      throws org.omg.CORBA.BAD_OPERATION;
abstract public void                insert_ulonglong(long l);
```

```
abstract public float               extract_float()
    throws org.omg.CORBA.BAD_OPERATION;
abstract public void                insert_float(float f);

abstract public double              extract_double()
    throws org.omg.CORBA.BAD_OPERATION;
abstract public void                insert_double(double d);

abstract public boolean             extract_boolean()
    throws org.omg.CORBA.BAD_OPERATION;
abstract public void                insert_boolean(boolean b);

abstract public char                extract_char()
    throws org.omg.CORBA.BAD_OPERATION;
abstract public void                insert_char(char c)
    throws org.omg.CORBA.DATA_CONVERSION;

abstract public char                extract_wchar()
    throws org.omg.CORBA.BAD_OPERATION;
abstract public void                insert_wchar(char c);

abstract public byte                extract_octet()
    throws org.omg.CORBA.BAD_OPERATION;
abstract public void                insert_octet(byte b);

abstract public org.omg.CORBA.Any  extract_any()
    throws org.omg.CORBA.BAD_OPERATION;
abstract public void                insert_any(org.omg.CORBA.Any a);

abstract public org.omg.CORBA.Object    extract_Object()
    throws org.omg.CORBA.BAD_OPERATION;
abstract public void               insert_Object(
                                        org.omg.CORBA.Object o);
//          throw excep when typecode inconsistent with value
abstract public void               insert_Object(
                                        org.omg.CORBA.Object o,
                                        org.omg.CORBA.TypeCode t)
                                            throws org.omg.CORBA.MARSHAL;

abstract public String              extract_string()
    throws org.omg.CORBA.BAD_OPERATION;
abstract public void                insert_string(String s)
    throws org.omg.CORBA.DATA_CONVERSION, org.omg.CORBA.MARSHAL;

abstract public String              extract_wstring()
    throws org.omg.CORBA.BAD_OPERATION;
abstract public void                insert_wstring(String s)
    throws org.omg.CORBA.MARSHAL;
```

```
// insert and extract typecode

abstract public org.omg.CORBA.TypeCode extract_TypeCode()
    throws org.omg.CORBA.BAD_OPERATION;
abstract public voidinsert_TypeCode(
                                        org.omg.CORBA.TypeCode t);

// insert and extract Principal

abstract public org.omg.CORBA.Principal extract_Principal()
    throws org.omg.CORBA.BAD_OPERATION;
abstract public void insert_Principal(
                                    org.omg.CORBA.Principal p);

// insert non-primitive IDL types

abstract public void insert_Streamable(
                            org.omg.CORBA.portable.Streamable s);

}
```

5.15 Mapping for Certain Nested Types

IDL allows type declarations nested within interfaces. Java does not allow classes to be nested within interfaces. Hence those IDL types that map to Java classes and that are declared within the scope of an interface must appear in a special “scope” package when mapped to Java.

IDL interfaces that contain these type declarations will generate a scope package to contain the mapped Java class declarations. The scope package name is constructed by appending **Package** to the IDL type name.

5.15.1 Example

```
// IDL

module Example {
    interface Foo {
        exception e1 {};
    };
};

// generated Java

package Example.FooPackage;
final public class e1 extends org.omg.CORBA.UserException
    {...}
```

5.16 Mapping for Typedef

Java does not have a typedef construct.

5.16.1 Simple IDL types

IDL types that are mapped to simple Java types may not be subclassed in Java. Hence any typedefs that are type declarations for simple types are mapped to the original (mapped type) everywhere the typedef type appears.

The IDL types covered by this rule are described in Section 5.4, “Mapping for Basic Types.

Helper classes are generated for all typedefs.

5.16.2 Complex IDL types

Typedefs for non arrays and sequences are “unwound” to their original type until a simple IDL type or user-defined IDL type (of the non typedef variety) is encountered.

Holder classes are generated for sequence and array typedefs only.

5.16.2.1 Example

```
// IDL

struct EmpName {
    string firstName;
    string lastName;
};
typedef EmpName EmpRec;

// generated Java
//        regular struct mapping for EmpName
//        regular helper class mapping for EmpRec

final public class EmpName {
    ...
}

public class EmpRecHelper {
    ...
}
```

Mapping Pseudo-Objects to Java 6

6.1 Introduction

Pseudo objects are constructs whose definition is usually specified in “IDL”, but whose mapping is language specified. A pseudo object is not (usually) a regular CORBA object. Often it exposed to either clients and/or servers as a process, or a thread local, programming language construct.

For each of the standard IDL pseudo-objects we either specify a specific Java language construct or we specify it as a **pseudo interface**.

This mapping is based on the revised version 1.1 C++ mapping.

We have chosen the option allowed in the IDL specification section 4.1.3 to define Status as void and have eliminated it for the convenience of Java programmers.

6.1.1 Pseudo Interface

The use of **pseudo interface** is a convenient device which means that most of the standard language mapping rules defined in this specification may be mechanically used to generate the Java. However, in general the resulting construct is not a CORBA object. Specifically it is:

- not represented in the Interface Repository
- no helper classes are generated
- no holder classes are generated
- mapped to a Java **public abstract class** that does not extend or inherit from any other classes or interfaces

Note: The specific definition given for each piece of PIDL may override the general guidelines above. In such a case, the specific definition takes precedence.

All of the pseudo interfaces are mapped as if they were declared in:

```
module org {
    module omg {
        module CORBA {
                                        ...
```

That is, they are mapped to the **org.omg.CORBA** Java package.

6.2 Certain Exceptions

The standard CORBA PIDL uses several exceptions, **Bounds**, **BadKind**, and **InvalidName**.

No holder and helper classes are defined for these exceptions, nor are they in the interface repository., However so that users can treat them as “normal exceptions” for programming purposes, they are mapped as normal user exceptions.

They are defined within the scopes that they are used. A **Bounds** and **BadKind** exception are defined in the **TypeCodePackage** for use by **TypeCode**. A **Bounds** exception is defined in the standard CORBA module for use by **NVList**, **ExceptionList**, and **ContextList**. An **InvalidName** exception is defined in the **ORBPackage** for use by **ORB**.

```
// Java

package org.omg.CORBA;

final public class Bounds
    extends org.omg.CORBA.UserException {
    public Bounds() {...}
}

package org.omg.CORBA.TypeCodePackage;

final public class Bounds
    extends org.omg.CORBA.UserException {
    public Bounds() {...}
}
final public class BadKind
    extends org.omg.CORBA.UserException {
    public BadKind() {...}

}

package org.omg.CORBA.ORBPackage;

final public class InvalidName
    extends org.omg.CORBA.UserException {
    public InvalidName() {...}
}
```

6.3 Environment

The **Environment** is used in request operations to make exception information available.

```
// Java code

package org.omg.CORBA;

public abstract class Environment {
    void exception(java.lang.Exception except);
    java.lang.Exception exception();
    void clear();
}
```

6.4 NamedValue

A **NamedValue** describes a name, value pair. It is used in the DII to describe arguments and return values, and in the context routines to pass property, value pairs.

In Java it includes a name, a value (as an any), and an integer representing a set of flags.

```
typedef unsigned long Flags;
typedef string Identifier;
const Flags ARG_IN = 1;
const Flags ARG_OUT = 2;
const Flags ARG_INOUT = 3;
const Flags CTX_RESTRICT_SCOPE = 15;

pseudo interface NamedValue {
    readonly attribute Identifier name;
    readonly attribute any value;
    readonly attribute Flags flags;
};
```

```
// Java

package org.omg.CORBA;

public interface ARG_IN {
    public static final int value = 1;
}
public interface ARG_OUT {
    public static final int value = 2;
}
public interface ARG_INOUT {
    public static final int value = 3;
}

public interface CTX_RESTRICT_SCOPE {
    public static final int value = 15;
}

public abstract class NamedValue {
    public abstract String name();
    public abstract Any value();
    public abstract int flags();
}
```

6.5 NVList

A **NVList** is used in the DII to describe arguments, and in the context routines to describe context values.

In Java it maintains a modifiable list of **NamedValue**s.

```
pseudo interface NVList {
    readonly attribute unsigned long count;
    NamedValue add(in Flags flags);
    NamedValue add_item(in Identifier item_name, in Flags flags);
    NamedValue add_value(in Identifier item_name,
                         in any val,
                         in Flags flags);
    NamedValue item(in unsigned long index) raises (CORBA::Bounds);
    void remove(in unsigned long index) raises (CORBA::Bounds);
};
```

```
// Java

package org.omg.CORBA;

public abstract class NVList {
    public abstract int count();
    public abstract NamedValue add(int flags);
    public abstract NamedValue add_item(String item_name, int flags);
    public abstract NamedValue add_value(String item_name, Any val,
                                         int flags);
    public abstract NamedValue item(int index)
                        throws org.omg.CORBA.Bounds;
    public abstract void remove(int index) throws org.omg.CORBA.Bounds;
}
```

6.6 ExceptionList

An **ExceptionList** is used in the DII to describe the exceptions that can be raised by IDL operations.

It maintains a list of modifiable list of **TypeCode**s.

```
pseudo interface ExceptionList {
    readonly attribute unsigned long count;
    void add(in TypeCode exc);
    TypeCode item (in unsigned long index) raises (CORBA::Bounds);
    void remove (in unsigned long index) raises (CORBA::Bounds);
};

// Java

package org.omg.CORBA;

public abstract class ExceptionList {
    public abstract int count();
    public abstract void add(TypeCode exc);
    public abstract TypeCode item(int index)
                    throws org.omg.CORBA.Bounds;
    public abstract void remove(int index)
                    throws org.omg.CORBA.Bounds;
}
```

6.7 Context

A **Context** is used in the DII to specify a context in which context strings must be resolved before being sent along with the request invocation.

```
pseudo interface Context {
    readonly attribute Identifier context_name;
    readonly attribute Context parent;
    Context create_child(in Identifier child_ctx_name);
    void set_one_value(in Identifier propname, in any propvalue);
    void set_values(in NVList values);
    void delete_values(in Identifier propname);
    NVList get_values(in Identifier start_scope,
                      in Flags op_flags,
                      in Identifier pattern);
};

// Java

package org.omg.CORBA;

public abstract class Context {
    public abstract String context_name();
    public abstract Context parent();
    public abstract Context create_child(String child_ctx_name);
    public abstract void set_one_value(String propname,
                                       Any propvalue);
    public abstract void set_values(NVList values);
    public abstract void delete_values(String propname);
    public abstract NVList get_values(String start_scpe, int op_flags,
                                      String pattern);
}
```

6.8 ContextList

```
pseudo interface ContextList {
    readonly attribute unsigned long count;
    void add(in string ctx);
    string item(in unsigned long index) raises (CORBA::Bounds);
    void remove(in unsigned long index) raises (CORBA::Bounds);
};
```

```
// Java

package org.omg.CORBA;

public abstract class ContextList {
    public abstract int count();
    public abstract void add(String ctx);
    public abstract String item(int index)
        throws org.omg.CORBA.Bounds;
    public abstract void remove(int index)
        throws org.omg.CORBA.Bounds;
}
```

6.9 Request

```
pseudo interface Request {
    readonly attribute Object target;
    readonly attribute Identifier operation;
    readonly attribute NVList arguments;
    readonly attribute NamedValue result;
    readonly attribute Environment env;
    readonly attribute ExceptionList exceptions;
    readonly attribute ContextList contexts;

    attribute Context ctx;

    any add_in_arg();
    any add_named_in_arg(in string name);
    any add_inout_arg();
    any add_named_inout_arg(in string name);
    any add_out_arg();
    any add_named_out_arg(in string name);
    void set_return_type(in TypeCode tc);
    any return_value();

    void invoke();
    void send_oneway();
    void send_deferred();
    void get_response();
    boolean poll_response();
};
```

```
// Java

package org.omg.CORBA;

public abstract class Request {

    public abstract Object target();
    public abstract String operation();
    public abstract NVList arguments();
    public abstract NamedValue result();
    public abstract Environment env();
    public abstract ExceptionList exceptions();
    public abstract ContextList contexts();

    public abstract Context ctx();
    public abstract void ctx(Context c);

    public abstract Any add_in_arg();
    public abstract Any add_named_in_arg(String name);
    public abstract Any add_inout_arg();
    public abstract Any add_named_inout_arg(String name);
    public abstract Any add_out_arg();
    public abstract Any add_named_out_arg(String name);
    public abstract void set_return_type(TypeCode tc);
    public abstract Any return_value();

    public abstract void invoke();
    public abstract void send_oneway();
    public abstract void send_deferred();
    public abstract void get_response();
    public abstract boolean poll_response();
}
```

6.10 ServerRequest and Dynamic Implementation

```
pseudo interface ServerRequest {
    Identifier op_name();
    Context ctx();
    void params(in NVList parms);
    void result(in Any res);
    void   except(in Any ex);
} ;
```

```
// Java

package org.omg.CORBA;

public abstract class ServerRequest {
    public abstract String op_name();
    public abstract Context ctx();
    public abstract void params(NVList parms);
    public abstract void result(Any a);
    public abstract void except(Any a);
```

The **DynamicImplementation** interface defines the interface such a dynamic server is expect to implement. Note that it inherits from the base class for stubs and skeletons (see Section 8.5.2, “Portable ObjectImpl").

```
// Java

package org.omg.CORBA;

public abstract class DynamicImplementation
        extends org.omg.CORBA.portable.ObjectImpl {
    public abstract void invoke(org.omg.CORBA.ServerRequest request);
}
```

6.11 TypeCode

The deprecated **parameter** and **param_count** methods are not mapped.

```
enum TCKind {
    tk_null, tk_void,
    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
};

// Java

package org.omg.CORBA;

public final class TCKind {
    public static final int _tk_null = 0;
    public static final
        TCKind tk_null = new TCKind(_tk_null);
    public static final int _tk_void = 1;
        TCKind tk_void = new TCKind(_tk_void);
    public static final int _tk_short = 2;
        TCKind tk_short = new TCKind(_tk_short);
    public static final int _tk_long = 3;
        TCKind tk_long = new TCKind(_tk_long);
    public static final int _tk_ushort = 4;
        TCKind tk_ushort = new TCKind(_tk_ushort);
    public static final int _tk_ulong = 5;
        TCKind tk_ulong = new TCKind(_tk_ulong);
    public static final int _tk_float = 6;
        TCKind tk_float = new TCKind(_tk_float);
    public static final int _tk_double = 7;
        TCKind tk_double = new TCKind(_tk_double);
    public static final int _tk_boolean = 8;
        TCKind tk_boolean = new TCKind(_tk_boolean);
    public static final int _tk_char = 9;
        TCKind tk_char = new TCKind(_tk_char);
    public static final int _tk_octet = 10;
        TCKind tk_octet = new TCKind(_tk_octet);
    public static final int _tk_any = 11;
        TCKind tk_any = new TCKind(_tk_any);
    public static final int _tk_TypeCode = 12;
        TCKind tk_TypeCode = new TCKind(_tk_TypeCode);
    public static final int _tk_Principal = 13;
        TCKind tk_Principal = new TCKind(_tk_Principal);
    public static final int _tk_objref = 14;
        TCKind tk_objref = new TCKind(_tk_objref);
    public static final int _tk_stuct = 15;
        TCKind tk_stuct = new TCKind(_tk_stuct);
    public static final int _tk_union = 16;
        TCKind tk_union = new TCKind(_tk_union);
```

```
        public static final int _tk_enum = 17;
            TCKind tk_enum = new TCKind(_tk_enum);
        public static final int _tk_string = 18;
            TCKind tk_string = new TCKind(_tk_string);
        public static final int _tk_sequence = 19;
            TCKind tk_sequence = new TCKind(_tk_sequence);
        public static final int _tk_array = 20;
            TCKind tk_array = new TCKind(_tk_array);
        public static final int _tk_alias = 21;
            TCKind tk_alias = new TCKind(_tk_alias);
        public static final int _tk_except = 22;
            TCKind tk_except = new TCKind(_tk_except);
        public static final int _tk_longlong = 23;
            TCKind tk_longlong = new TCKind(_tk_longlong);
        public static final int _tk_ulonglong = 24;
            TCKind tk_ulonglong = new TCKind(_tk_ulonglong);
        public static final int _tk_longdouble = 25;
            TCKind tk_longdouble = new TCKind(_tk_longdouble);
        public static final int _tk_wchar = 26;
            TCKind tk_wchar = new TCKind(_tk_wchar);
        public static final int _tk_wstring = 27;
            TCKind tk_wstring = new TCKind(_tk_wstring);
        public static final int _tk_fixed = 28;
            TCKind tk_fixed = new TCKind(_tk_fixed);

        public int value() {...}
        public static TCKind from_int(int value) {...}
        private TCKind(int value) {...}
    }

    pseudo interface TypeCode {

        exception Bounds {};
        exception BadKind {};

        // for all TypeCode kinds
        boolean equal(in TypeCode tc);
        TCKind kind();

        // for objref, struct, union, enum, alias, and except
        RepositoryID id() raises (BadKind);
        RepositoryId name() raises (BadKind);

        // for struct, union, enum, and except
        unsigned long member_count() raises (BadKind);
        Identifier member_name(in unsigned long index)
            raises (BadKind, Bounds);

        // for struct, union, and except
        TypeCode member_type(in unsigned long index)
            raises (BadKind, Bounds);
```

```
        // for union
        any member_label(in unsigned long index) raises (BadKind, Bounds);
        TypeCode discriminator_type() raises (BadKind);
        long default_index() raises (BadKind);

        // for string, sequence, and array
        unsigned long length() raises (BadKind);
        TypeCode content_type() raises (BadKind);

        }

// Java

package org.omg.CORBA;

public abstract class TypeCode {

        // for all TypeCode kinds
        public abstract boolean equal(TypeCode tc);
        public abstract TCKind kind();

        // for objref, struct, unio, enum, alias, and except
        public abstract String id() throws TypeCodePackage.BadKind;
        public abstract String name() throws TypeCodePackage.BadKind;

        // for struct, union, enum, and except
        public abstract int member_count() throws TypeCodePackage.BadKind;
        public abstract String member_name(int index)
                throws TypeCodePackage.BadKind;

        // for struct, union, and except
        public abstract TypeCode member_type(int index)
                throws TypeCodePackage.BadKind,
                        TypeCodePackage.Bounds;

        // for union
        public abstract Any member_label(int index)
                throws TypeCodePackage.BadKind,
                        TypeCodePackage.Bounds;
        public abstract TypeCode discriminator_type()
                throws TypeCodePackage.BadKind;
        public abstract int default_index() throws TypeCodePackage.BadKind;

        // for string, sequence, and array
        public abstract int length() throws TypeCodePackage.BadKind;
        public abstract TypeCode content_type() throws TypeCodePackage.BadKind;
}
```

6.12 ORB

Issue – create_named_value() is incorrect in the CORBA spec. We have used the correct definition here. It is indicated as a change to CORBA 2.0

The **UnionMemeberSeq**, **EnumMemberSeq**, and **StructMemberSeq** typedefs are real IDL and bring in the Interface Repository. Rather than tediously list its interfaces, and other assorted types, suffice it to say that it is all mapped following the rules for IDL set forth in this specification in Chapter 5, “Mapping IDL to Java".

StructMember[], UnionMember[], EnumMember[]

```
pseudo interface ORB {

    exception InvalidName {};

    typedef string ObjectId;
    typedef sequence<ObjectId> ObjectIdList;

    ObjectIdList list_initial_services();
    Object resolve_initial_references(in ObjectId object_name)
        raises(InvalidName);

    string object_to_string(in Object obj);
    Object string_to_object(in string str);

    NVList create_list(in long count);
    NVList create_operation_list(in OperationDef oper);
    NamedValue create_named_value(in String name, in Any value,
                    in Flags flags);
    ExceptionList create_exception_list();
    ContextList create_context_list();

    Context get_default_context();
    Environment create_environment();

    void send_multiple_requests_oneway(in RequestSeq req);
    void send_multiple_requests_deferred(in RequestSeq req);
    boolean poll_next_response();
    Request get_next_response();

// typecode creation

    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);

        TypeCode create_sequence_tc (       in unsigned long bound,
                                            in TypeCode element_type);

        TypeCode create_recursive_sequence_tc(     in unsigned long bound,
                                            in unsigned long offset);

        TypeCode create_array_tc (          in unsigned long length,
                                            in TypeCode element_type);

        Current get_current();
    // Additional operations for Java mapping

        TypeCode get_primitive_tc(in TCKind tcKind);
        Any create_any();
        OutputStream create_output_stream();
        void connect(Object obj);
        void disconnect(Object obj);

    // additional  methods for ORB initialization go here, but only
    //    appear  in the mapped Java (seeSection 8.8, “ORB Initialization" )
```

```
// Java signatures
//    public static ORB init(Strings[] args, Properties props);
//    public static ORB init(Applet app, Properties props);
//    public static ORB init();
// abstract protected void set_parameters(String[] args,
//                                        java.util.Properties props);
// abstract protected void set_parameters(java.applet.Applet app,
//                                        java.util.Properties props);

}

// Java

package org.omg.CORBA;

public abstract class ORB {

    public abstract String[] list_initial_services();
    public abstract org.omg.CORBA.Object resolve_initial_references(
        String object_name)
        throws org.omg.CORBA.ORBPackage.InvalidName;

    public abstract String object_to_string(org.omg.CORBA.Object obj);
    public abstract org.omg.CORBA.Object string_to_object(String str);

    public abstract NVList create_list(int count);
    public abstract NVList create_operation_list(OperationDef oper);
    public abstract NamedValue create_named_value(String name,
                                                  Any value,
                                                  int flags);
    public abstract ExceptionList create_exception_list();
    public abstract ContextList create_context_list();

    public abstract Context get_default_context();
    public abstract Environment create_environment();

    public abstract void send_multiple_requests_oneway(Request[] req);
    public abstract void sent_multiple_requests_deferred(Request[] req);
    public abstract boolean poll_next_response();
    public abstract Request get_next_response();
```

```
        // typecode creation

        public abstract TypeCode create_struct_tc(String id,
                                                  String name,
                                                  StructMember[] members);
        public abstract TypeCode create_union_tc(String id,
                                                  String name,
                                                  TypeCode discriminator_type,
                                                  UnionMember[] members);
        public abstract TypeCode create_enum_tc(String id,
                                                  String name,
                                                  EnumMember[] members);
        public abstract TypeCode create_alias_tc(String id,
                                                  String name,
                                                  TypeCode original_type);
        public abstract TypeCode create_exception_tc(String id,
                                                  String name,
                                                  StructMember[] members);
        public abstract TypeCode create_interface_tc(String id,
                                                  String name);
        public abstract TypeCode create_string_tc(int bound);
        public abstract TypeCode create_wstring_tc(int bound);
        public abstract TypeCode create_sequence_tc(int bound,
                                                  TypeCode element_type);
        public abstract TypeCode create_recursive_sequence_tc(int bound,
                                                  int offset);
        public abstract TypeCode create_array_tc(int length,
                                                  TypeCode element_type);

        public abstract Current get_current();

        // additional methods for IDL/Java mapping

        public abstract TypeCode get_primitive_tc(TCKind tcKind);
        public abstract Any create_any();
        public abstract org.omg.CORBA.portable.OutputStream
            create_output_stream();
        public abstract void connect(           org.omg.CORBA.Object obj);
        public abstract void disconnect(        org.omg.CORBA.Object obj);

        // additional static methods for ORB initialization

        public static ORB init(Strings[] args, Properties props);
        public static ORB init(Applet app, Properties props);
        public static ORB init();
        abstract protected void set_parameters(String[] args,
                                                  java.util.Properties props);
        abstract protected void set_parameters(java.applet.Applet app,
                                                  java.util.Properties props);
    }
```

6.13 CORBA::Object

The IDL **Object** type is mapped to the **org.omg.CORBA.Object and org.omg.CORBA.ObjectHelper** classes as shown below.

The Java interface for each user defined IDL **interface** extends **org.omg.CORBA.Object**, so that any object reference can be passed anywhere a **org.omg.CORBA.Object** is expected.

```
// Java

package org.omg.CORBA;

public interface Object {
    boolean _is_a(String Identifier);
    boolean _is_equivalent(Object that);
    boolean _non_existent();
    int _hash(int maximum);
    org.omg.CORBA.Object _duplicate();
    void _release();
    ImplementationDef _get_implementation();
    InterfaceDef _get_interface();
    Request _request(String s);
    Request _create_request(Context ctx,
                            String operation,
                            NVList arg_list,
                            NamedValue result);
    Request _create_request(Context ctx,
                            String operation,
                            NVList arg_list,
                            NamedValue result,
                            ExceptionList exclist,
                            ContextList ctxlist);
}
```

6.14 Current

```
pseudo interface Current {
}
```

```
// Java

public abstract class Current
        extends org.omg.CORBA.portable.ObjectImpl {
}
```

6.15 Principal

```
pseudo interface Principal {
attribute sequence<octet> name;
}

// Java

public abstract class Principal {
    public abstract byte[] name();
    public abstract void name(byte[] name);
}
```

Server-Side Mapping 7

7.1 Introduction

This chapter discusses how implementations create and register objects with the ORB runtime.

It will be patterned after the server framework architecture to be described by the final submission to the Server Side Portability RFP.

Issue – This chapter is subject to revision pending the outcome of the Portability RFP submission

7.2 Transient Objects

For this initial submission only a minimal API to allow application developers to implement transient ORB objects is described. We do not expect there to be major changes as a result of the Portability work.

7.2.1 Servant Base Class

For each IDL interface ***<interface_name>*** the mapping defines a Java class as follows:

```
// Java

public class _<interface_name>ImplBase implements <interface_name> {
}
```

7.2.2 Servant Class

For each interface, the developer must write a servant class. Instances of the servant class implement ORB objects. Each instance implements a single ORB object, and each ORB object is implemented by a single servant.

Each object implementation implements ORB objects that supports a most derived IDL interface. If this interface is **<interface_name>**, then the servant class must extend ***_<interface_name>*ImplBase**.

The servant class must define public methods corresponding to the operations and attributes of the IDL interface supported by the object implementation, as defined by the mapping specification for IDL interfaces. Providing these methods is sufficient to satisfy all abstract methods defined by ***_<interface_name>*ImplBase**.

7.2.3 Creating A Transient ORB Object

To create an instance of an object implementation, the developer instantiates the servant class.

7.2.4 Connecting a Transient ORB Object

Object implementations (object references) may be explicitly connected to the ORB by calling the ORB's **connect()** method (seeSection 6.12, "ORB").

An object implementation may also be automatically and implicitly connected to the ORB if it is passed as a (mapped IDL) parameter to a (mapped) IDL operation that is itself not implemented as a local (Java) object. I.e., it has to be marshaled and sent outside of the process address space. Note, a vendor is free to connect such an object implementation "earlier" (e.g. upon instantiation), but it must connect the implementation to the ORB when it is passed as described above.

Note that calling **connect()** when an object is already connected has no effect.

7.2.5 Disconnecting a Transient ORB Object

The servant may disconnect itself from the ORB by invoking the ORB's **disconnect()** method (see Section 6.12, "ORB"). After this method returns, incoming requests will be rejected by the ORB by raising the **CORBA::OBJECT_NOT_EXIST** exception. The effect of this method is to cause the ORB object to appear to be destroyed from the point of view of remote clients.

Note that calling **disconnect()** when the object is not connected has no effect.

Note however, that requests issued using the servant directly (e.g. using the implementatioon's this pointer) do not pass through the ORB; these requests will continue to be processed by the servant.

7.3 Persistent Objects

Issue – dependent upon the Portability specification

Java ORB Portability Interfaces 8

8.1 Introduction

The APIs specified here provide the minimal set of functionality to allow portable stubs and skeletons to be used with a Java ORB. The interoperability requirements for Java go beyond that of other languages. Because Java classes are often downloaded and come from sources that are independent of the ORB in which they will be used, it is essential to define the interfaces that the stubs and skeletons use. Otherwise, use of a stub (or skeleton) will require: either that it have been generated by a tool that was provided by the ORB vendor (or is compatible with the ORB being used), or that the entire ORB runtime be downloaded with the stub or skeleton. Both of these scenarios are unacceptable.

8.1.1 Design Goals

The design balances several goals:

- Size
  Stubs and skeletons must have a small bytecode footprint in order to make downloading fast in a browser environment and to minimize memory requirements when bundled with a Java VM, particularly in specialized environments such as set-top boxes.
- Performance
  Obviously, the runtime performance of the generated stub code must be excellent. In particular, care must be taken to minimize temporary Java object creation during invocations in order to avoid Java VM garbage collection overhead.
- Reverse Mapability
  The design does not require adding methods to user-defined types such as structures and exceptions to ensure that stubs and skeletons generated by IDL to Java compilers and reverse Java to IDL mapping tools are interoperable and binary compatible.

A very simple delegation scheme is specified here. Basically, it allows ORB vendors maximum flexibility for their ORB interfaces, as long as they implement the interface APIs. Of course vendors are free to add proprietary extensions to their ORB runtimes. Stubs and skeletons which require proprietary extensions will not necessarily be portable or interoperable and may require download of the corresponding runtime.

8.1.2 Portability Package

The APIs needed to implement portability are found in the **org.omg.CORBA.portable** package

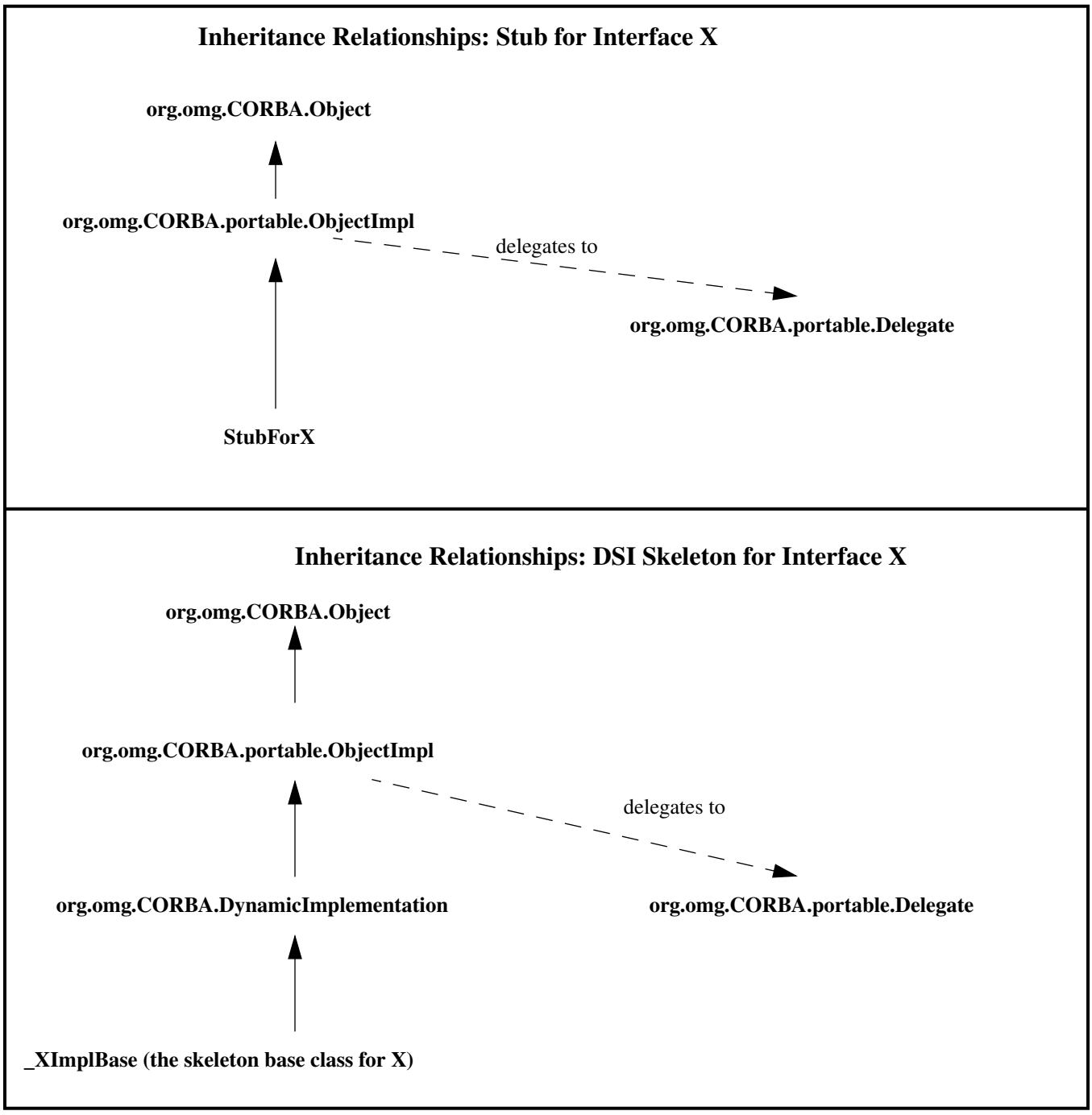
The portability package contains interfaces and classes that are designed for and intended to be used by ORB implementors. It exposes the publicly defined APIs that are used to connect stubs and skeletons to the ORB.

8.2 Architecture

The stub and skeleton portability architecture allows the use of the DII and DSI as its portability layer. The mapping of the DII and DSI PIDL have operations that support the efficient implementation of portable stubs and skeletons

All stubs shall inherit from a common base class **org.omg.CORBA.portable.ObjectImpl**. The class is responsible for delegating shared functionality such as is_a() to the vendor specific implementation. This model provides for a variety of vendor dependent implementation choices, while reducing the client-side and server “code bloat”.

All DSI-based skeletons inherit from **org.omg.CORBA.DynamicImplementation**.


Inheritance Relationships: Stub for Interface X
org.omg.CORBA.Object
org.omg.CORBA.portable.ObjectImpl
delegates to
org.omg.CORBA.portable.Delegate
StubForX
Inheritance Relationships: DSI Skeleton for Interface X
org.omg.CORBA.Object
org.omg.CORBA.portable.ObjectImpl
delegates to
org.omg.CORBA.DynamicImplementation
org.omg.CORBA.portable.Delegate
_XImplBase (the skeleton base class for X)

8.3 Streamable APIs

The Streamable Interface API provides the support for the reading and writing of complex data types. It is implemented by static methods on the Helper classes. They are also used in the Holder classes for reading and writing complex data types passed as out and inout parameters.

```
package org.omg.CORBA.portable;

public interface Streamable {
    void _read(org.omg.CORBA.portable.InputStream istream);
    void _write(org.omg.CORBA.portable.OutputStream ostream);
    org.omg.CORBA.TypeCode _type();
}
```

8.4 Streaming APIs

The streaming APIs are Java interfaces that provide for the reading and writing of all of the mapped IDL types to and from streams. Their implementations are used inside the ORB to marshal parameters and to insert and extract complex datatypes into and from **Any**s.

The streaming APIs are found in the **org.omg.CORBA.portable** package.

The ORB object is used as a factory to create an output stream. An input stream may be created from an output stream.

```
package org.omg.CORBA;

interface ORB {
        OutputStream        create_output_stream();
};
```

```
package org.omg.CORBA.portable;

public abstract class InputStream {
    public abstract boolean   read_boolean();
    public abstract char      read_char();
    public abstract char      read_wchar();
    public abstract byte      read_octet();
    public abstract short     read_short();
    public abstract short     read_ushort();
    public abstract int       read_long();
    public abstract int       read_ulong();
    public abstract long      read_longlong();
    public abstract long      read_ulonglong();
    public abstract float     read_float();
    public abstract double    read_double();
    public abstract String    read_string();
    public abstract String    read_wstring();
    public abstract void      read_boolean_array(boolean[] value,
                                                 int offset, int length);
    public abstract void      read_char_array(char[] value,
                                              int offset, int length);
    public abstract void      read_wchar_array(char[] value,
                                               int offset, int length);
    public abstract void      read_octet_array(byte[] value,
                                               int offset, int length);
    public abstract void      read_short_array(short[] value,
                                               int offset, int length);
    public abstract void      read_ushort_array(short[] value,
                                                int offset, int length);
    public abstract void      read_long_array(int[] value,
                                              int offset, int length);
    public abstract void      read_ulong_array(int[] value,
                                               int offset, int length);
    public abstract void      read_longlong_array(long[] value,
                                                  int offset, int length);
    public abstract void      read_ulonglong_array(long[] value,
                                                   int offset, int length);
    public abstract void      read_float_array(float[] value,
                                               int offset, int length);
    public abstract void      read_double_array(double[] value,
                                                int offset, int length);
    public abstract org.omg.CORBA.Object      read_Object();
    public abstract org.omg.CORBA.TypeCode    read_TypeCode();
    public abstract org.omg.CORBA.Any         read_any();
    public abstract org.omg.CORBA.Principal   read_Principal();
}
```

```
public abstract class OutputStream {
    public abstract InputStream create_input_stream();
    public abstract void write_boolean     (boolean value);
    public abstract void write_char        (char value);
    public abstract void write_wchar       (char value);
    public abstract void write_octet       (byte value);
    public abstract void write_short       (short value);
    public abstract void write_ushort      (short value);
    public abstract void write_long        (int value);
    public abstract void write_ulong       (int value);
    public abstract void write_longlong    (long value);
    public abstract void write_ulonglong   (long value);
    public abstract void write_float       (float value);
    public abstract void write_double      (double value);
    public abstract void write_string      (String value);
    public abstract void write_wstring     (String value);
    public abstract void write_boolean_array(boolean[] value,
                                             int offset, int length);
    public abstract void write_char_array(char[] value,
                                             int offset, int length);
    public abstract void write_wchar_array(char[] value,
                                             int offset, int length);
    public abstract void write_octet_array(byte[] value,
                                             int offset, int length);
    public abstract void write_short_array(short[] value,
                                             int offset, int length);
    public abstract void write_ushort_array(short[] value,
                                             int offset, int length);
    public abstract void write_long_array(int[] value,
                                             int offset, int length);
    public abstract void write_ulong_array(int[] value,
                                             int offset, int length);
    public abstract void write_longlong_array(long[] value,
                                             int offset, int length);
    public abstract void write_ulonglong_array(long[] value,
                                             int offset, int length);
    public abstract void write_float_array(float[] value,
                                             int offset, int length);
    public abstract void write_double_array(double[] value,
                                             int offset, int length);
    public abstract void write_Object(org.omg.CORBA.Object value);
    public abstract void write_TypeCode(org.omg.CORBA.TypeCode value);
    public abstract void write_any (org.omg.CORBA.Any value);
    public abstract void write_Principal(org.omg.CORBA.Principal value);
}
```

8.5 Portability Stub Interfaces

8.5.1 Stub Design

The stub class is implemented on top of the DII..

8.5.2 Portable ObjectImpl

The ObjectImpl class is the base class for stubs and skeletons. It provides the basic delegation mechanism.

The method _ids() returns an array of repository ids that an object implements. The string at the zero index shall represent the most derived interface. The last id, for the generic CORBA object (i.e. “IDL:omg.org/CORBA/Object:1.0”), is implied and not present.

```
package org.omg.CORBA.portable;

abstract public class ObjectImpl implements
                                        org.omg.CORBA.Object {

    private Delegate __delegate;

    public Delegate _get_delegate() {
        if (__delegate == null) {
            throw new org.omg.CORBA.BAD_OPERATION();
        }
        return _delegate;
    }

    public void _set_delegate(Delegate delegate) {
        __delegate = delegate;
    }

    public abstract String[] _ids() {...}

// methods for standard CORBA stuff

    public org.omg.CORBA.ImplementationDef
                                        _get_implementation() {
        return _get_delegate().get_implementation(this);
    }

    public org.omg.CORBA.InterfaceDef
                                        _get_interface() {
        return _get_delegate().get_interface(this);
    }
```

```
    public org.omg.CORBA.Object _duplicate() {
        return _get_delegate().duplicate(this);
    }

    public void _release() {
        _get_delegate().release(this);
    }

    public boolean _is_a(String repository_id) {
        return _get_delegate().is_a(this, repository_id);
    }

    public boolean _is_equivalent(org.omg.CORBA.Object rhs) {
        return _get_delegate().is_equivalent(this, rhs);
    }

    public boolean _non_existent() {
        return _get_delegate().non_existent(this);
    }

    public int _hash(int maximum) {
        return _get_delegate().hash(this, maximum);
    }

    public org.omg.CORBA.Request _request(String operation) {
        return _get_delegate().request(this, operation);
    }

    public org.omg.CORBA.Request _create_request(
                            org.omg.CORBA.Context ctx,
          String operation,
          org.omg.CORBA.NVList arg_list,
          org.omg.CORBA.NamedValue result) {
        return _get_delegate().create_request(this, ctx,
                                    operation, arg_list, result);
    }

    public Request _create_request(
                                org.omg.CORBA.Context ctx,
          String operation,
          org.omg.CORBA.NVList arg_list,
          org.omg.CORBA.NamedValue result,
            org.omg.CORBA.ExceptionList exceptions,
          org.omg.CORBA.ContextList contexts) {
        return _get_delegate().create_request(this, ctx, operation,
                                arg_list, result,exceptions, contexts);
    }

}
```

8.6 Delegate

The delegate class provides the ORB vendor specific implementation of CORBA object.

```
// Java

package org.omg.CORBA.portable;

public abstract class Delegate {

    public abstract org.omg.CORBA ImplementationDef get_implementation(
                                        org.omg.CORBA.Object self);
    public abstract org.omg.CORBA.InterfaceDef get_interface(
                                        org.omg.CORBA.Object self);
    public abstract org.omg.CORBA.Object duplicate(
                                        org.omg.CORBA.Object self);
    public abstract void release(org.omg.CORBA.Object self);
    public abstract boolean is_a(org.omg.CORBA.Object self,
                                        String repository_id);
    public abstract boolean non_existent(org.omg.CORBA.Object self);
    public abstract boolean is_equivalent(org.omg.CORBA.Object self,
                                        org.omg.CORBA.Object rhs);
    public abstract int hash(org.omg.CORBA.Object self
                                        int max);
    public abstract org.omg.CORBA.Request request(org.omg.CORBA.Object self,
                                        String operation);
    public abstract org.omg.CORBA.Request create_request(
                                        org.omg.CORBA.Object self,
                                        org.omg.CORBA.Context ctx,
                                        String operation,
                                        org.omg.CORBA.NVList arg_list,
                                        org.omg.CORBA.NamedValue result);
    public abstract org.omg.CORBA.Request create_request(
                                        org.omg.CORBA.Object self,
                                        org.omg.CORBA.Context ctx,
                                        String operation,
                                        org.omg.CORBA.NVList arg_list,
                                        org.omg.CORBA.NamedValue result,
                                        org.omg.CORBA.ExceptionList excepts,
                                        org.omg.CORBA.ContextList contexts);
}
```

8.7 Skeleton

The skeleton uses the DynamicImplementation (seeSection 6.10, “ServerRequest and Dynamic Implementation”).

See Section 7.2.2, “Servant Class” for more information.

8.8 ORB Initialization

The ORB class represents an implementation of a CORBA ORB. Vendors specific ORB implementations can extend this class to add new features.

There are several cases to consider when creating the ORB instance. An important factor is whether an applet in a browser or an stand-alone Java application is being used.

In any event, when creating an ORB instance, the class names of the ORB implementation are located using the following search order:

- check in Applet parameter or application string array, if any
- check in properties parameter, if any
- check in the System properties
- fall back on a hardcoded default behavior

8.8.1 Standard Properties

The OMG standard properties are defined in the following table.

Table 8-1 Standard ORB properties

Property Name	Property Value
org.omg.CORBA.ORBClass	class name of an ORB implementation
org.omg.CORBA.ORBSingletonClass	class name of the singleton ORB implementation

8.8.2 ORB Initialization Methods

There are three forms of initialization as shown below. In addition the actual ORB implementation (subclassed from **ORB**) must implement the **set_parameters()** methods so that the initialization parameters will be passed into the ORB from the initialization methods.

```
// Java

package org.omg.CORBA;

abstract public class ORB {

    // Application init

    public static ORB init(String[] args,
                                        java.util.Properties props) {
        // call to: set_parameters(args, props);
        ...
        }

    // Applet init

    public static ORB init(java.applet.Applet app,
                                        java.util.Properties props) {
        // call to: set_parameters(app, props);
        ...
        }

    // Default (singleton) init

    public static ORB init()
        {...}

    // Implemented by subclassed ORB implementations
    //    and called by init methods to pass in their params

    abstract protected void set_parameters(String[] args,
                                        java.util.Properties props);
    abstract protected void set_parameters(Applet app,
                                        java.util.Properties props);

}
```

8.8.2.1 Default initialization

The default initialization method returns the singleton ORB. If called multiple times it will always return the same the Java object.

The primary use of the no-argument version of **ORB.init()** is to provide a factory for **TypeCode**s for use by Helper classes implementing the **type()** method, and to create Any instances that are used to describe union labels as part of creating a union **TypeCode**. These Helper classes may be baked-in to the browser (e.g. for the interface repository stubs or other wildly popular IDL) and so may be shared across untrusted applets downloaded into the browser. The returned ORB instance is shared across all applets and therefore must have sharply restricted capabilities so that unrelated applets can be isolated from each other. It is not intended to be used directly by applets. Therefore, the ORB returned by **ORB.init()**, if called from a Java applet, may only be used to create **Typecode**s. An attempt to invoke other "regular" ORB operations shall raise a system exception.

If called from an application a fully functional ORB object is returned.

8.8.2.2 Application initialization

The application initialization method should be used from a stand-alone Java application. It is passed a array of strings which are the command arguments and a list of Java properties. Either the argument array or the properties may be **null**.

It returns a new fully functional ORB Java object each time it is called.

8.8.2.3 Applet initialization

The applet initialization method should be used from an applet. It is passed "the applet" and a list of Java properties. Either the applet or the properties may be **null**.

It returns a new fully functional ORB Java object each time it is called.

Conformance Issues 9

9.1 Introduction

This chapter specifies the compliance points for this specification

9.2 Compliance

In order to be conformant with this specification, all the specified mappings and language features must be supported and implemented using the specified semantics.

- There are no optional interfaces
- There are no optional compliance points
- Note: PIDL specifically excluded by this specification is not required to be mapped.

Changes to CORBA 2.0

10

This submission's primary change to the CORBA specification is to add a new language mapping. It makes relatively minor (textual) changes to the rest of the specification, all of which are detailed in this chapter.

- Add new section for Java Language mapping
- Update the Table of Contents and Index as necessary

10.1 Request PIDL

It appears that there an error was introduced in the PIDL defining **Request** by the C+1.1 Revision that is now part of CORBA 2.0. Overloaded versons of the add arg routines were added without modifying the PIDL. Since IDL does not support overloading the new operations would have to have new names in the PIDL. We have specified the Java mapping as if this were the case and also added the new operations to the PIDL (seeSection 6.9, "Request").

10.2 Certain Exceptions

The CORBA 2.0 specification is ambiguous as to which scope the Bounds, BadKind, and InvalidName are declared.

This submission places Bounds and BadKind in the TypeCode scope. It also places a different Bounds in the CORBA module, as well as placing InvalidName in the ORB module.