IDL/Java Language Mapping

Joint Submission

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

OMG TC Document orbos/96-12-08

December 23, 1996 1:30 pm

Table of Contents

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
- Netscape Communications Corporation
- Novell, Inc.
- Oracle Corporation
- Sun Microsystems, Inc.
- Visigenic Software, Inc.

1.2 Guide to the Submission

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

- Persistent Object Framework
- new PIDL and other required mappings

If that work converges quickly enough, then specifications of the required mappings (for the Portable Object Adapter) as well as a Persistent Object Framework will be included in the revised submission. If that work does not converge quickly enough for us add it the revised submission in a timely manner, then it is the intention of the submitters to ask the ORBOS Task Force to recommend approval for a possible revised submission and to leave the RFP open for a later submission which addresses these important topics.

1.4 Conventions

IDL appears using this font.

`Java code appears using this font.`

In various places a few issues are highlighted. These are mostly areas where we have discovered that some additional clarification may be needed. These issues will all be addressed by the revised submission deadline, if not before.

Please note that the change bars have no semantic meaning. They show the places that final edits were applied to the last reviewed draft submission. They are present for the convenience of the 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

Patrick Ravenal
Expersoft Corporation
6620 Mesa Ridge Road, Suite 100
San Diego, CA 92121
USA

phone: +1 619 824-4173
fax: +1 619 824-4110
email: pat@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

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 Avenume
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

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 a 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 PIDL has been formally deprecated in recently adopted OMG specifications (e.g. principal by the Security Service specification). The Portability RFP process is about to deprecate the PIDL. In

the latter case, we expect that 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 Java 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 the be 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 being started 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.

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 to not 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. 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 Import

The import statement is needed to allow the importation of type definitions for use without the generation of target code.

4.3 Names

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 an attempt is made to use an IDL interface with a name that is a Java reserved word.

4.4 Mapping of Module

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

4.5 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.

4.6 Mapping for Constants

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 uses 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 lesser code, faster and 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

One goal was to avoid loading the Any methods if they are not used. A simpler approach would be to have the Any methods on the class with the same name as the IDL type (without the ``Helper" suffix). However, that would mean the Any methods are loaded even if they are not actually used (which is usually the case); the user, in this case, would end up having to pay the penalty of loading up the Any methods even though they are not being used.

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 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 apply the general name collision avoidance rule and prepend an underscore in front of the package name.

4.16 Mapping for Typedef

Mapping for typedef is somewhat problematic since Java does not support the notion of typedef. It is also somewhat the semantic implications of “distinct IDL type” is somewhat unclear.

We have proposed a solution for the basic IDL types. Several approaches to more complex types are still be considered.

4.17 Helper Classes

One goal was to avoid loading the Any methods if they are not used. A simpler approach would be to have the Any methods on the class with the same name as the IDL type (without the ``Helper" suffix). However, that would mean the Any methods are loaded even if they are not actually used (which is usually the case); the user, in this case, would end up having to pay the penalty of loading up the Any methods even though they are not being used.

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.

Mapping IDL to Java 5

5.1 Introduction

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

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 Import

An **import** statement will be added to the IDL language to support the importing of IDL declarations into the name space of an IDL file for use. No code or interface repository objects are generated by the use of this statement.

Issue -

Addition of this statement is being considered by the Portability RFP submitters. It is essential that such a statement be added to IDL. Without it there is no reliable way to indicate that no code (stubs/skeletons) be generated for types which are only referenced. In Java it is important that the generated classes only be generated once and in the correct scope.

In addition to specifying a text file, the syntax should also leave open the possibility of importing the definition from other places such as an Interface Repository, or other environmentally specific objects.

One suggestion is to allow a string containing a URL, with the requirement that at least URL's that specify a file, i.e. FILE://..., be supported.

5.3 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. If that could generate a collision, then additional underscores are prepended, as necessary.

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 either by appending a suffix or prepending a prefix. In those cases that the "additional" names would 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.

An main example of this occurs in the mapping of IDL interface names (Section 5.12, "Mapping for Interface), where an additional "helper" class (which could potentially conflict) is generated. Hence an IDL interface named **fooHelper** is mapped to two Java classes **`_fooHelper`** and **`_fooHelperHelper`** because it could conflict with the mapping of an IDL interface name **foo** (which would map to **`foo`** and **`fooHelper`**).

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

5.3.1 Reserved Names

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

- The Java class **`Constants`** (Section 5.6, "Mapping for Constants)
- The Java class **`fooHelper`**, where foo is the name of IDL interface (Section 5.12, "Mapping for Interface)
- The Java class **`fooHolder`**, where foo is the name of an IDL user defined type
- 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 5.5.1.2, "Holder Classes)

Issue – This list needs to have a few more items added to it.

5.4 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 (unamed) Java global scope.

5.4.1 Example

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

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

5.5 Mapping for Basic Types

5.5.1 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 Exception 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. There are basically two cases:

- 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 negative integers in Java are handled correctly as large unsigned values.

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::DATA_CONVERSION
wstring	`java.lang.String`	CORBA::DATA_CONVERSION

IDL Type	Java type	Exceptions
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.

5.5.1.1 *Future Support*

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

IDL Type	Java type	Exceptions
long double	`long double`	
fixed	`java.lang.Bignum`	CORBA::DATA_CONVERSION

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

5.5.1.2 *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.

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 to which the datatype is mapped, with **`Holder`** appended. Note: these class names are fixed, so that name collision avoidance does not change the mappings for the holder classes (see Section 5.3, “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.

The holder classes for the basic types are defined below. They are in the **`org.omg.CORBA`** package.

```
// Java

package org.omg.CORBA;

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

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

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

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

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

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

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

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

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

public class ObjectHolder {
   public java.lang.Object value;
   public ObjectHolder() {};
   public ObjectHolder(java.lang.String initial) {
      value = initial;
   }
}

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

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

5.5.2 Boolean

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

5.5.3 Character Types

IDL characters are 8-bit quantities typically representing elements of the ISO 8859.1 character set while Java characters are 16-bit unsigned quantities representing Unicode characters. In order the 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 ISO 8859.1, a **CORBA::DATA_CONVERSION** shall be thrown.

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

5.5.4 Octet

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

5.5.5 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 may be done at marshal time. Character range violation and bounds violation cause a **CORBA:: DATA_CONVERSION** exception to be raised.

The IDL **wstring**, 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 may be done at marshal time. Character range violation cause a **CORBA::DATA_CONVERSION** exception to be thrown.

5.5.6 Integer Types

The integer types map as shown.

5.5.7 Floating Point Types

The IDL float and double map to the Java IEEE 754 compliant floating point types

5.5.8 Future Fixed Point Types

The IDL **fixed** point type is mapped to the Java **`java.lang.Bignum`** class. Size violations raises a **CORBA::DATA_CONVERSION** exception.

This is left for a future revision.

5.6 Mapping for Constants

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

5.6.1 Constants Within A Module

Constants declared within an IDL module 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.

5.6.1.1 Example

```
// IDL

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

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

5.6.2 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.

5.6.2.1 Example

```
// IDL

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

```
// generated Java

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

5.7 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 and two static data members per label 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() {...}
}
```

One of the members is a **public static final** 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 value. Values are assigned sequentially starting with 0.

The holder class for enum is **<enum_name>Holder**.

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 = 0;
   public static final EnumType b = new EnumType(_b);


   public static final int _c = 0;
   public static final EnumType none = new EnumType(_c);

   public int value() {...}
   };
```

5.8 Mapping for Struct

An IDL **struct** is mapped to a Java class with the same name that provides instance variables for the fields 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:

```
public class <class>Holder {
   public <class> value;
   public <class>Holder() {};
   public <class>Holder(<class> initial) {
      value = initial;
   }
```

5.8.1 Example

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

// generated Java
public class StructType {
   // instance variables
   public int field1;
   public String field2;
   // constructors
   public StructType() {}
   public StructType(int _field1, String _field2)
      {...}
   }

public class StructTypeHolder {
   public StructType value;
   public StructTypeHolder() {};
   public StructTypeHolder(StructType initial) {
      value = initial;
   }
```

5.9 Mapping for Union

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

- a default constructor
- an accessor method for the 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.

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 the discriminant. It is the responsiblity of the Java code generator (e.g., the IDL complier, or other tool) to detect this situation and refuse to generate illegal code.

Issue – More clarification for the exact rules for union needs to be added

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:

```
public class <union_class>Holder {
   public <union_class> value;
   public <union_class>Holder() {};
   public <union_class>Holder(<union_class> initial) {
      value = initial;
   }
```

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
public class UnionType {
   // constructor
   public UnionType() {....}
```

```
   // discriminator accessor
   public int discriminator() {....}

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

 public class UnionTypeHolder {
   public UnionType value;
   public UnionTypeHolder() {};
   public UnionTypeHolder(UniontType initial) {
      value = initial;
   }
```

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 is done on bounded sequences when they are marshaled as parameters to IDL operations, and an IDL **CORBA::MARSHAL** exception 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:

```
public class <sequence_class>Holder {
   public <sequence_element_type>[] value;
   public <sequence_class>Holder() {};
   public <sequence_class>Holder(
                  <sequence_element_type>[] initial) {
      value = initial;
   }
```

5.10.1 Example

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

// generated Java

public int[] UnboundedData;
public int[] BoundedData;

public class UnboundedDataHolder {
   public int[] value;
   public UnboundedDataHolder() {};
   public UnboundedDataHolder(int[] initial) {
      value = initial;
   }

public class BoundedDataHolder {
   public int[] value;
   public BoundedDataHolder() {};
   public BoundedDataHolder(int[] initial) {
      value = initial;
   }
```

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 arrays's mapped Java classname with **`Holder`** appended to it as follows:

```
public class <array_class>Holder {
   public <array_element_type>[] value;
   public <array_class>Holder() {};
   public <array_class>Holder(
                  <array_element_type>[] initial) {
      value = initial;
   }
```

5.11.1 Example

```
// IDL

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

// generated Java

public int[] larray;

public class larrayHolder {
   public int[] value;
   public larrayHolder() {};
   public larrayHolder(int[] initial) {
      value = initial;
   }
```

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:

```
public class <interface_class>Holder {
   public <interface_class> value;
   public <interface_class>Holder() {};
   public <interface_class>Holder(
                  <interface_class> initial) {
      value = initial;
   }
```

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;
        attribute readonly 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 {
   public static Face narrow(org.omg.CORBA.Object obj)
      {...}
}

public class FaceHolder {
   public Face value;
   public FaceHolder() {};
   public FaceHolder(
                  Face initial) {
      value = initial;
   }
```

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.

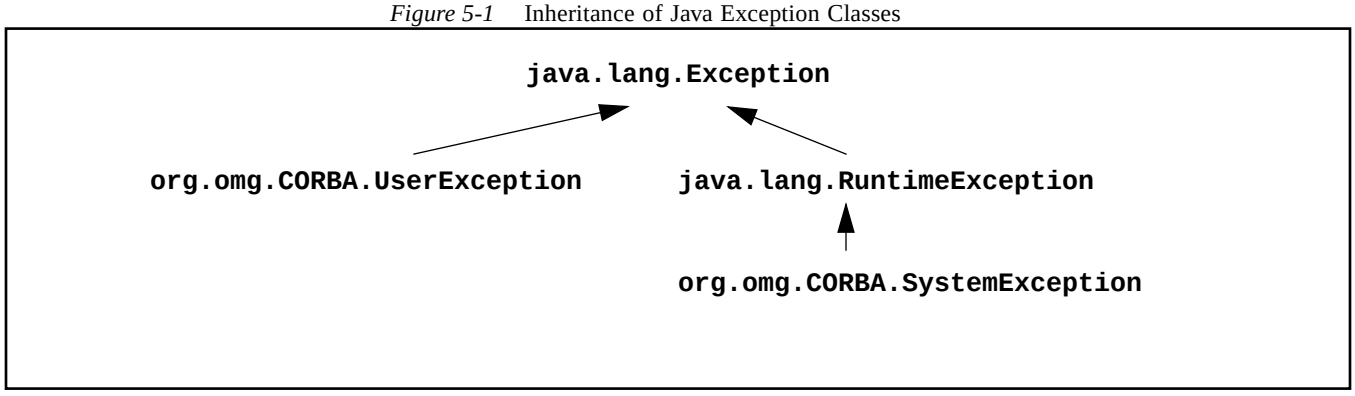
5.13 Mapping for Exception

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 Java classes that extend **`org.omg.CORBA.UserException`** and are otherwise mapped just like the IDL **struct** type.

If the exception is not defined within a nested IDL scope (essentially within an interface) then the Java class name is the same as the IDL exception name and is defined within the scope of the Java package that corresponds to its enclosing IDL module.

If the exception is defined within a nested IDL scope (essentially within an interface) then its Java class name is the same as the IDL exception name with an underscore (_) prepended to it. It is defined within a special scope. See Section 5.15, “Mapping for Certain Nested Types for more details.

5.13.1.1 Example

```
// IDL

module Example {
    exception ex1 { string reason; }
}

// Generated Java

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

5.13.2 System Exceptions

The standard IDL system exceptions are mapped to 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
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.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::TX1	org.omg.CORBA.<new tx 1>
CORBA::TX2	org.omg.CORBA.<new tx 2>
CORBA::TX3	org.omg.CORBA.<new tx 3>

The definitions of the relevant classes are specified below.

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

package org.omg.CORBA;

public class CompletionStatus {
   // Completion Status constants
   public static final int COMPLETED_YES = 0,
                           COMPLETED_NO = 1,
                           COMPLETED_MAYBE = 2;
   public static final boolean narrow(int x) {...}
}

public class
   SystemException extends org.omg.CORBA.RuntimeException
   {
   public int minor;
   public int completed;
   }

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

`create`() creates the any with a type of **TK_NULL** and no value. (Note that an attempt to extract will result in a **CORBA::MARSHAL** exception being raised.)

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::MARSHAL** exception being raised. This operation is provided primarily so that the type may be set properly for IDL **out** parameters.

Issue – Define semantics of equals

```
package org.omg.CORBA;

abstract public class Any {

abstract public boolean equals(org.omg.CORBA.Any)
static public Any create();

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

// read and write to/from streams
abstract public void read_stream(org.omg.CORBA.InputStream)
      raises org.omg.CORBA:MARSHAL;
abstract public void
   write_stream(org.omg.CORBA.OutputStream);

// to and from each primitive type

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

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

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

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

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

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

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

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

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

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

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

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

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

abstract public org.omg.CORBA.Object extract_Object()
   throws org.omg.CORBA.BAD_OPERATION;
abstract public void    insert_Object(org.omg.CORBA.Object);
abstract public void    insert_Object(org.omg.CORBA.Object,
                                      org.omg.CORBA.TypeCode);

abstract public String   extract_string()
   throws org.omg.CORBA.BAD_OPERATION;
abstract public void     insert_string(String);

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

}
```

5.15 Mapping for Certain Nested Types

Certain IDL types

IDL allows type declarations nested within other type declarations. Java does not allow interfaces to be nested within classes. 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. The affected IDL types are constants, typedefs, and exceptions.

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 prepending an underscore (_) to the IDL type name.

5.15.1 Example

```
// IDL

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

// generated Java

package Example._Foo;
public class e1 {...}
```

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 5.5, “Mapping for Basic Types.

5.16.2 Complex IDL types

Issue – Method for mapping typedefs for complex types needs to be clarified, pending clarification of meaning of “distinct IDL types”.

5.16.2.1 Example

```
// IDL

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

5.17 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, and typecode.

The helper class also has a narrow operation defined for it.

For any user defined IDL type, **typename**, the following is the Java code generated for the type.

```
// generated Java helper

public class <typename>Helper {
   public static Any insert(<typename> t);
   public static <typename> extract(Any a);
   public static get_typecode();
   public static get_repository_identifier();
```

For constructed types that are IDL sequences or arrays the helper class is a bit simpler. The insert and extract operations are only generated if there is an IDL **typedef** naming the constructed type and the parameter or result type is an array of the element mapped Java types of the sequence or array.

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

5.17.1 Examples

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

// generated Java
public class stHelper {
   public static Any insert (st s);
   public static st extract(Any a);
   public static TypeCode get_typecode();
   public static String get_repository_identifier);
}
```

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

// generated Java helper

public class IntSeqHelper {
   public static Any insert(int[] seq);
   public static int[] extract(Any a);
   public static TypeCode get_typecode();
   public static String get_repository_identifier();
}
```

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.

Issue – Should the InvalidName, Bounds, and BadKind exceptions stay where they currently are (which is somewhat awkward) or be moved “up” into the org.omg.CORBA module?

6.1.1 Pseudo Interface

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

- not represented in the Interface Repository
- no helper classes are generated

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 Environment

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

```
// Java code

package org.omg.CORBA;

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

6.3 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;
const ARG_IN = 1;
const ARG_OUT = 2;
const ARG_INOUT = 3;

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

6.4 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 (TypeCode::Bounds);
    void remove(in unsigned long index) raises (TypeCode::Bounds);
}
```

6.5 *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 (TypeCode::Bounds);
    void remove (in unsigned long index) raises (TypeCode::Bounds);
    }
```

6.6 *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);
}
```

6.7 ContextList

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

6.8 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_in_arg(string name);
    any add_inout_arg();
    any add_inout_arg(string name);
    any add_out_arg();
    any add_out_arg(string name);
    void set_return_type(TypeCode tc);
    any return_value();

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

6.9 ServerRequest and Dynamic Implementation

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

The **`DynamicImplementation`** interface defines the interface such a dynamic server is expect to implement.

```
// Java

package org.omg.CORBA;

public interface DynamicImplementation {
   public void invoke(org.omg.CORBA.ServerRequest request);
}
```

6.10 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
};

pseudo interface TypeCode {

    exception Bounds {};
    exception BadKind {};

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

    // for objref, struct, unio, 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, seuence, and array
    unsigned long length() raises (BadKind);
    TypeCode content_type() raises (BadKind);

    }
```

Note that according to the mapping rules, the exceptions are mapped to:

```
_TypeCode.Bounds
_TypeCode.BadKind;
```

6.11 ORB

Issue – the ORB interface probably should be mapped to a Java class. This is dependent upon clarifying the initialization and Orb replaceability rules.

The **UnionMemeberSeq**, **EnumMemberSeq**, and **StructMemberSeq** typedefs 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 set forth in this specification.

```
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();
    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 get_primitive_tc(          in TCKind tcKind);

    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_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);
}
```

Note that according to the mapping rules, the exceptions are mapped to:

```
_ORB.InvalidName;
```

6.12 CORBA::Object

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

Issue – Because of the probability of name clashes, the operations on* Object *are mapped to Java methods with an underscore (_) prepended.

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 _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);
}
```

6.13 Current

```
pseudo interface Current {
                }
```

6.14 Principal

```
pseudo interface Principal {
                }
```

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> {
   public <interface_name>ImplBase(ORB orb,
                                   boolean connect);
   protected void connect_to_orb();
   protected void disconnect_from_orb();
}
```

This class is declared to implement the interface, and it must provide implementations for the methods defined by the IDL interface. Note that any of the required methods may be inherited in a given ORB implementation.

7.2.2 Servant Class

For each object implementation, 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 support 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**.

Issue – We would like to have the servant class define two constructors, one of which takes an ORB parameter and one of which does not. Providing two constructors allows the user of this class either to specify an explicit ORB or use a default ORB. Various issues relating to defining and implementing the default ORB scenario are TBD and this model is not yet supported.

7.2.3 Creating An ORB Object

To create an instance of an object implementation, the developer instantiates the servant class. The instantiation must invoke a servant base class constructor. As a consequence of invoking the servant base class constructor, the servant itself becomes a CORBA object reference that supports the **<interface_name>** interface.

The first parameter of the constructor is the ORB in which to register the servant. The second parameter is a boolean that indicates whether or not to register the servant immediately. If **connect** is true, the servant is registered immediately with the

specified ORB. If **connect** is false, then registration may be delayed until the servant is passed as an object reference outside the current Java virtual machine or until **connect_to_orb()** is called (whichever happens first). Delaying registration with the ORB may reduce resource consumption, depending upon the ORB implementation. Registering the servant with the ORB prior to passing it out as an object reference may enable nonstandard features in some ORBs.

7.2.4 Connecting an ORB Object

If a servant is created without immediate registration with an ORB (**connect** is false), then the **connect_to_orb()** method can be called to register the servant with the ORB. If the **connect_to_orb()** method is called on a servant that is already connected to the ORB, nothing happens. If the **connect_to_orb()** method is called on a servant that has been disconnected from the ORB, then the **CORBA::OBJECT_NOT_EXIST** exception is raised.

7.2.5 Disconnecting an ORB Object

The servant can disconnect itself from the ORB by invoking the **disconnect_from_orb()** method inherited from the servant base class. 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, however, that requests issued using the servant directly (i.e., within the same Java virtual machine) do not pass through the ORB; these requests will continue to be processed by the servant.

If the servant has not been connected to an ORB or has already been disconnected from the ORB, then **disconnect_from_orb()** has no effect. A servant may not be garbage collected while it is connected to the ORB.

7.3 Persistent Objects

Issue – tbd

Java Stub/Skeleton ORB Interfaces 8

Issue – This chapter is still subject to revision pending clarification of some more detailed rules. Issues include: clarification of potential coexistence of multiple orbs, more detailed specification of portable marshaling behavior, and initialization.

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.

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 their presence will not necessarily be portable or interoperable and may require download of the corresponding runtime.

8.2 Streaming APIs

```
package org.omg.CORBA;

interface ORB {
      OutputStream   create_output_stream();
};


interface InputStream {
   OutputStream      create_output_stream();
   boolean           read_boolean();
   char              read_char();
   char              read_wchar();
   byte              read_octet();
   short             read_short();
   short             read_uhort();
   int               read_long();
   int               read_ulong();
   long              read_longlong();
   long              read_ulongong();
   float             read_float();
   double            read_double();
   double            read_longdouble();
   String            read_string();
   String            read_wstring();
   java.lang.Object  read_JavaObject(
                        org.omg.CORBA.TypeCode type_code);
   org.omg.CORBA.Object    read_Object();
   org.omg.CORBA.TypeCode  read_TypeCode();
   org.omg.CORBA.Any       read_any();
   byte[]                  read_octet_array(int length);
};
```

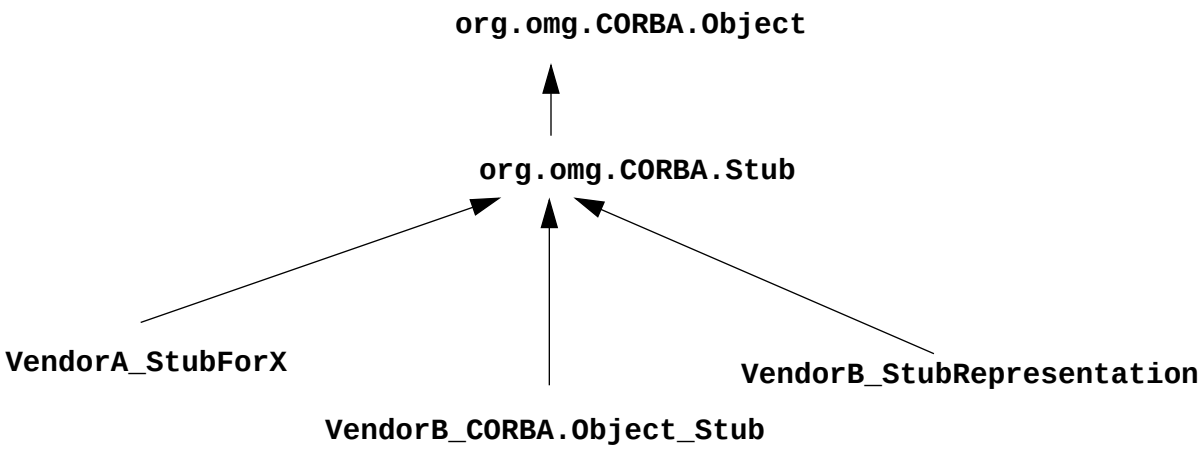
```
interface OutputStream {
   InputStream create_input_stream();
   void write_boolean      (boolean value);
   void write_char         (char value);
   void write_wchar        (char value);
   void write_octet        (byte value);
   void write_short        (short value);
   void write_ushort       (short value);
   void write_long         (int value);
   void write_ulong        (int value);
   void write_long_long    (long value);
   void write_ulonglong    (long value);
   void write_float        (float value);
   void write_double       (double value);
   void write_longdouble   (double value);
   void write_String       (String value);
   void write_Wstring      (String value);
   void write_Object       (org.omg.CORBA.Object value);
   void write_TypeCode     (org.omg.CORBA.TypeCode value);
   void write_any          (org.omg.CORBA.Any value);
   void write_octet_array  (byte[] value,
                            int offset,
                            int length);
   void write_JavaObject(java.lang.Object value,
                         org.omg.CORBA.TypeCode type_code);

};
```

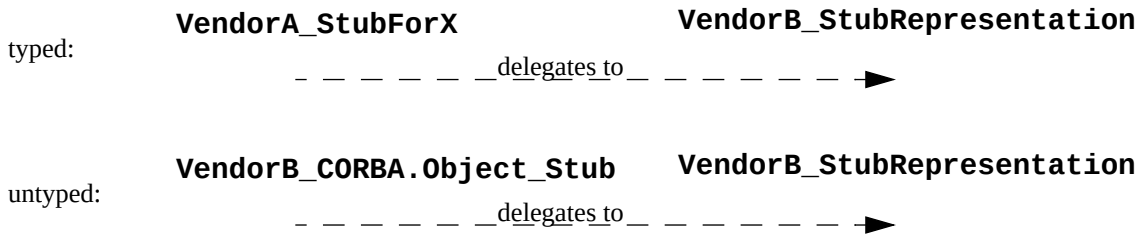
8.3 Client Side Portability Stub Interfaces

All stubs must inherit from the base Stub class. The Stub 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 “code bloat”.

In heritance Relationships: Stub for Interface X



Runtime (Dynamic) Delegation: Stub for Interface X




```
// Java

package org.omg.CORBA;

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

   private org.omg.CORBA.Stub _delegate;

   public org.omg.CORBA.Stub _delegate() {
      return _delegate;
   };

   public void _delegate(org.omg.CORBA.Stub delegate) {
      _delegate = delegate;
   };
```

```
    abstract String[] _repository_ids();

    public org.omg.CORBA.OutputStream _create_stream_request(
       String operation, boolean response_expected) {
       return _delegate()._create_stream_request(operation,
          response_expected);
    };


    public org.omg.CORBA.InputStream _invoke_stream_request(
       OutputStream output, boolean response_expected,
          String_var exception_id) {
       return _delegate()._invoke_stream_request(
          output, response_expected, exception_id);

// methods for standard CORBA stuff

    public org.omg.CORBA.Object _duplicate() {
       return _delegate()._duplicate();
    }

    public void _release() {
       return _delegate()._release();
    }

    public boolean _is_a(String repository_id) {
       return _delegate()._is_a(repository_id);
    }

    public boolean _is_equivalent(org.omg.CORBA.Object rhs) {
       return _delegate()._is_equivalent(rhs);
    }

    public boolean _non_existent() {
       return _delegate()._non_existent();
    }

    public int _hash(int maximum) {
       return _delegate()._hash(maximum);
    }

    public Request _request(String operation) {
       return _delegate()._request(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 _delegate()._create_request(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.TypeCode[] exceptions,
                  String[] contexts) {
      return _delegate()._create_request(ctx, operation,
                    arg_list, result,exceptions, contexts);
   }

   public org.omg.CORBA.InterfaceDef _get_interface() {
      return _delegate()._get_interface();
      }

   public
      org.omg.CORBA.ImplementationDef _get_implementation()
         {
         return _delegate()._get_implementation();
         }

// Java friendly convenience operations

   public int hashCode() {
      return _delegate().hashCode();
   }

   public boolean equals(org.omg.CORBA.Object rhs) {
      return _delegate().equals(rhs);
   }

   public String toString() {
      return _delegate().toString();
   }

   public java.lang.Object clone() {
      return _delegate().clone();
   }

}
```

To support narrow, the generated code shall register with the ORB the stub class to be instantiated. The narrow operation returns a new instance of “theClass” if “object” supports the interface represented by “repository_id”. Otherwise it returns null.

```
// Registering stub classes with the ORB

package org.omg.CORBA;

   abstract public class ORB {

      org.omg.CORBA.Object narrow(
                     org.omg.CORBA.Object object,
                     String repository_id,
                     java.lang.Class java_class);
   }
```

8.4 Skeletons and Method Dispatch

Issue – The main unresolved issue here is how to actually hook up the skeleton to the ORB. When one considers the special requirements for Java security, then it is clear that a more complicated framework is required. In particular when an implementation is registered with the ORB if it is a “secure” implementation then it may not share connections with “insecure” implementations. This seems to require something like the ability to support multiple POAs. or multiple ORBs

8.4.1 Method Dispatch

```
// Java

package org.omg.CORBA;

public class MethodPointer {

   public String method_name;

   public int interface_id;

   public int method_id;

   public MethodPointer() {
   }
```

```
   public MethodPointer(
      String method_name, int interface_id, int method_id
   ) {
      this.method_name = method_name;
      this.interface_id = interface_id;
      this.method_id = method_id;
   }

}

interface ORB {
   void connect (Skeleton target);
}
```

8.4.2 Server side skeleton framework

```
package org.omg.CORBA;

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

   private org.omg.CORBA.Skeleton _delegate;

   // this constructor connects to the orb automatically

   protected Skeleton(ORB orb) {
      orb.connect(this); // side effect of setting delegate
   }

   // this constructor defers connecting to the orb
   //  (i.e. the object is not yet ready)

   protected Skeleton() {
   }

   public org.omg.CORBA.Skeleton _delegate() {
      return _delegate;
   }

   public void _delegate(org.omg.CORBA.Skeleton delegate) {
      _delegate = delegate;
   }
```

```
    abstract String[] _repository_ids();

    abstract MethodPointer[] _dispatch_table();

    abstract public boolean_execute(
          org.omg.CORBA.MethodPointer methodPointer,
          org.omg.CORBA.InputStream input,
          org.omg.CORBA.OutputStream output);

  // methods for standard CORBA stuff

    public org.omg.CORBA.Object _duplicate() {
       return _delegate()._duplicate();
    }

    public void _release() {
       return _delegate()._release();
    }

    public boolean _is_a(String repository_id) {
       return _delegate()._is_a(repository_id);
    }

    public boolean _is_equivalent(org.omg.CORBA.Object rhs) {
       return _delegate()._is_equivalent(rhs);
    }

    public boolean _non_existent() {
       return _delegate()._non_existent();
    }

    public int _hash(int maximum) {
       return _delegate()._hash(maximum);
    }

    public org.omg.CORBA.Request _request(String operation) {
       return _delegate()._request(operation);
    }

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

```
   public org.omg.CORBA.Request _create_request(
                        Context ctx,
                        String operation,
                        org.omg.CORBA.NVList arg_list,
                        org.omg.CORBA.NamedValue result,
                        org.omg.CORBA.TypeCode[] exceptions,
                        String[] contexts) {
      return _delegate()._create_request(ctx, operation,
                     arg_list, result,exceptions, contexts);
   }

   public org.omg.CORBA.InterfaceDef _get_interface() {
      return _delegate()._get_interface();
      }

   public org.omg.CORBA.ImplementationDef
_get_implementation() {
      return _delegate()._get_implementation();
      }

// Java friendly convenience operations

   public int hashCode() {
      return _delegate().hashCode();
   }

   public boolean equals(java.lang.Object rhs) {
      return _delegate().equals(rhs);
   }

   public String toString() {
      return _delegate().toString();
   }

   public java.lang.Object clone() {
      return _delegate().clone();
   }

}
```

8.5 ORB Initialization

```
package org.omg.CORBA;

abstract public class ORB {

// this call does the vendor specific initialization...

abstract void _init(java.applet.Applet applet);

public static ORB init(String orb_class_name,
                       java.applet.Applet applet) {
   if (orb_class_name == null) {
      if (applet == null) {
         orb_class_name =
            System.getProperty("DEFAULT_ORB_CLASS_NAME");
      }
      else {
         orb_class_name =
            applet.getParameter("DEFAULT_ORB_CLASS_NAME");
      }
   }
   ORB orb;
   try {
      orb =(ORB)Class.forName(orb_class_name).newInstance();
   }
   catch (Throwable e) {
      throw new org.omg.CORBA.INITIALIZE
         ("Invalid ORB class: " + orb_class_name);
   }
   orb_init(applet);
   return orb;
}

public static ORB init(String orb_class_name) {
   return init(orb_class_name, null);
}

public static ORB init(java.applet.Applet applet) {
   return init(null, applet);
}

public static ORB init() {
   return init(null, null);
}
```

```
abstract org.omg.CORBA.Object string_to_object(String s);

// etc...
```

8.6 Example

Assume we have the following IDL interface:

```
interface X {
    string f (in string s);
    }
```

An example (probably generated) stub could be:

```
// Java

public class _st_X extends org.omg.CORBA.Stub implements X {

   public java.lang.String f(java.lang.String s) {
      org.omg.CORBA.OutputStream _output;
      _output = this._create_stream_request("f", true);
      _output.write_String(s);
      org.omg.CORBA.InputStream _input;
      _input = this._invoke_stream_request(
                           _output,true, null);
      java.lang.String _result;
      _result = _input.read_String();
      return _result;
   }

   public String[] _repository_ids = {
      return _repository_ids;
   }

   public String[] _repository_ids = { "IDL:X:1.0" };

}
```

An (probably generated) of the Helper class could be:

```
public class XHelper {

   private static Class _class;

   static X narrow(org.omg.CORBA.Object object) {
      if (object instance of X) {
         return (X)object;
      }
      return (X) TheGlobalOrb.narrow(object,
                                     repository_id,
                                     _class);
   }

   static { _class = Class.forName("X"); }

}
```

An example (probably generated) skeleton could be:

```
// Java

abstract public class _sk_X extends org.omg.CORBA.Skeleton
      implements X {

public MethodPointer[] _dispatch_table() {
   return _dispatch_table;
}
private static org.omg.CORBA.MethodPointer[]
   _dispatch_table = {
      new org.omg.CORBA.MethodPointer("f", 0, 0)
};

public boolean _execute(org.omg.CORBA.MethodPointer method,
                        org.omg.CORBA.InputStream input,
                        org.omg.CORBA.OutputStream output) {
   switch (method.interface_id) {
   case 0: {
      return _sk_X._execute(this, method.method_id, input,
                            output);
   }
   }
   throw new org.omg.CORBA.MARSHAL();
}
```

```
public static boolean _execute(X _self,
                        int _method_id,
                        org.omg.CORBA.InputStream input,
                        org.omg.CORBA.OutputStream output) {
   switch (_method_id) {
   case 0: {
      java.lang.String s;
      s = _input.read_String();
      java.lang.String _result = _self.f(s);
      _output.write_String(_result);
      return false /* no exception */
   }
   }
   throw new org.omg.CORBA.MARSAL();
}

public String[] _repository_ids() {
   return _respository_ids;
};

public static String[] _repository_ids = {
   "IDL:X:1.0"
};
}
```

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

Complete IDL definitions 11

This specification introduces no new standard IDL definitions.

The mapping of the standard CORBA module to Java is reproduced here as a convenience to implementors and developers. It is the result of mechanically applying the mapping specified herein to the standard CORBA IDL module and pseudo-objects with the exception of the BOA.

11.1 org.omg.CORBA Java module

This section will be included in the final revised submission.