Java™ Language to IDL Mapping 1

Note – The Java Language to IDL Mapping specification is aligned with CORBA version 3.0.2.

This is OMG document ptc/2003-01-17.

Contents

This chapter contains the following sections.

Section Title	**Page**
“Overview”	1-1
“The RMI/IDL Subset of Java”	1-2
“The IDL Mapping”	1-6
“Run-Time Issues”	1-32
“Portability Interfaces”	1-42
“Application Programming Interfaces”	1-62
“Generated IDL File Structure”	1-64

1.1 Overview

The Java distributed programming community has until now been forced to choose between two different mechanisms for distributed programming, Java Remote Method Invocation (RMI) and OMG IDL.

The RMI style of distributed programming has proven extremely popular because it is easy to use and avoids the need for Java programmers to learn a separate interface definition language. However, RMI lacks interoperability with other languages and it is not currently supported over standard protocols.

The mapping from Java RMI to OMG IDL and IIOP described in this chapter is intended to unify the ease-of-programming of Java RMI with support for cross-language operation (through OMG IDL) and support for standard protocols (through IIOP).

To encourage convergence between the RMI and CORBA programming communities, it is important to define a solution that is both fully compatible with current RMI semantics and fully compatible with OMG IDL, IIOP, and the CORBA object model.

The subset of Java that meets these goals is referred to as RMI/IDL.

1.2 The RMI/IDL Subset of Java

This section describes the subset of Java RMI that is mapped to IDL and can run over GIOP.

1.2.1 Overview of Conforming RMI/IDL Types

A *conforming* RMI/IDL type is a Java type whose values may be transmitted across an RMI/IDL remote interface at run-time.

A Java data type is a conforming RMI/IDL type if it is:

- one of the Java primitive types (see Section 1.2.2, “Primitive Types,” on page 1-2).
- a conforming remote interface (as defined in Section 1.2.3, “RMI/IDL Remote Interfaces,” on page 1-3).
- a conforming value type (as defined in Section 1.2.4, “RMI/IDL Value Types,” on page 1-4).
- an array of conforming RMI/IDL types (see Section 1.2.5, “RMI/IDL Arrays,” on page 1-5).
- a conforming exception type (see Section 1.2.6, “RMI/IDL Exception Types,” on page 1-5).
- a conforming CORBA object reference type (see Section 1.2.7, “CORBA Object Reference Types,” on page 1-6).
- a conforming IDL entity type (see Section 1.2.8, “IDL Entity Types,” on page 1-6).

1.2.2 Primitive Types

All the standard Java primitive types are supported as part of RMI/IDL. These are:

- **`void`**, **`boolean`**, **`byte`**, **`char`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**

1.2.3 RMI/IDL Remote Interfaces

An RMI *remote interface* defines a Java interface that can be invoked remotely. A Java interface is a conforming RMI/IDL remote interface if:

1. The interface is or inherits from **`java.rmi.Remote`** either directly or indirectly.
2. All methods in the interface are defined to throw **`java.rmi.RemoteException`** or a superclass of **`java.rmi.RemoteException`**. Throughout this section, references to methods in the interface include methods in any inherited interfaces.
3. There are no restrictions on method arguments and result types. However at run-time, the actual values passed as arguments or returned as results must be conforming RMI/IDL types (see Section 1.2.1, “Overview of Conforming RMI/IDL Types,” on page 1-2). In addition, for each RMI/IDL remote interface reference, the actual value passed or returned must be either a stub object or a remote interface implementation object (see Section 1.2.3.1, “Stubs and remote implementation classes,” on page 1-4).
4. All checked exception classes used in method declarations (other than **`java.rmi.RemoteException`** and its subclasses) are conforming RMI/IDL exception types (see Section 1.2.6, “RMI/IDL Exception Types,” on page 1-5).[1]
5. Method names may be overloaded. However, when an interface directly inherits from several base interfaces, it is forbidden for there to be method name conflicts between the inherited interfaces. This outlaws the case where an interface A defines a method “foo,” an interface B also defines a method “foo,” and an interface C tries to inherit from both A and B.
6. Constant definitions in the form of interface variables are permitted. The constant value must be a compile-time constant of one of the RMI/IDL primitive types or **`String`**.
7. Method and constant names must not cause name collisions when mapped to IDL (see Section 1.3.2.10, “Names that would cause OMG IDL name collisions,” on page 1-10).

The following is an example of a conforming RMI/IDL interface definition:

```
// Java
public interface Wombat extends java.rmi.Remote {
   String BLEAT_CONSTANT = “bleat”;
   boolean bleat(Wombat other)
   throws java.rmi.RemoteException;
```

1. Because unchecked exception classes and **`java.rmi.RemoteException`** and its subclasses are not mapped to IDL exceptions, it is not necessary for them to be conforming RMI/IDL exception types.

```
}
```

While the following is an example of a non-conforming RMI/IDL interface:

```
// Java
// IllegalInterface fails to extend Remote!!
public interface IllegalInterface {
   // illegalExceptions fails to throw RemoteException.
   void illegalExceptions();
}
```

1.2.3.1 Stubs and remote implementation classes

At run time, when a reference to an RMI/IDL remote interface is passed across a remote interface, the class of the actual object that is passed must be either a stub class or a remote implementation class.

A stub class is a class that has been created (normally by tools) to manage a remote object reference.

A remote implementation class is a class that acts as the server side implementation for a given RMI/IDL remote interface.

A given remote implementation class may implement several distinct RMI/IDL interfaces.

1.2.4 RMI/IDL Value Types

An RMI/IDL *value type* represents a class whose values can be moved between systems. So rather than transmitting a reference between systems, the actual state of the object is transmitted between systems. This requires that the receiving system have an analogous class that can be used to hold the received value.

Value types may be passed as arguments or results of remote methods, or as fields within other objects that are passed remotely.

A Java class is a conforming RMI/IDL value type if the following applies:

1. The class must implement the **`java.io.Serializable`** interface, either directly or indirectly, and must be serializable at run-time. It may serialize references to other RMI/IDL types, including value types and remote interfaces.
2. The class may implement **`java.io.Externalizable`**. (This indicates it overrides some of the standard serialization machinery.)
3. If the class is a non-static inner class, then its containing class must also be a conforming RMI/IDL value type.
4. A value type must not either directly or indirectly implement the **`java.rmi.Remote`** interface. (If this were allowed, then there would be potential confusion between value types and remote interface references.)

5. A value type may implement any interface except for **`java.rmi.Remote.`**
6. There are no restrictions on the method signatures for a value type.
7. There are no restrictions on **`static`** fields for a value type.
8. There are no restrictions on **`transient`** fields for a value type.
9. Method, constant, and field names must not cause name collisions when mapped to IDL (see Section 1.3.2.10, “Names that would cause OMG IDL name collisions,” on page 1-10).

Here is an example of a conforming RMI/IDL value type:

```
// Java
public class Point implements java.io.Serializable {
   public final static int CONSTANT_FOO = 3+3;
   private int x;
   private int y;
   public Point(int x, y) { ... }
   public int getX() { ... }
   public int getY() { ... }
}
```

1.2.4.1 The Java String Type

The **`java.lang.String`** class is a conforming RMI/IDL value type following these rules. Note, however, that **`String`** is handled specially when mapping Java to OMG IDL (see Section 1.3.5.11, “Mapping for java.lang.String,” on page 1-21).

1.2.5 RMI/IDL Arrays

Arrays of any conforming RMI/IDL type are also conforming RMI/IDL types. So **`int[]`** and **`String[][][]`** are conforming RMI/IDL types. Similarly if **`Wombat`** is a conforming RMI/IDL interface type, then **`Wombat[]`** is a conforming RMI/IDL type.

1.2.6 RMI/IDL Exception Types

An RMI/IDL exception type is a checked exception class (as defined by the Java Language Specification). Since checked exception classes extend **`java.lang.Throwable`**, which implements **`java.io.Serializable`**, it is unnecessary for an RMI/IDL exception class to directly implement **`java.io.Serializable`**.

A type is a conforming RMI/IDL exception if the class:

- is a checked exception class.
- meets the requirements for RMI/IDL value types defined in Section 1.2.4, “RMI/IDL Value Types,” on page 1-4.

Here's an example of a conforming RMI/IDL exception type:

```
// Java
public class MammalOverload extends MammalException {
   public MammalOverload(String message) {
      super(message);
   }
}
```

1.2.7 CORBA Object Reference Types

A conforming CORBA object reference type is either

- the Java interface **`org.omg.CORBA.Object`**, or
- a Java interface that extends **`org.omg.CORBA.Object`** directly or indirectly and conforms to the rules specified in the Java Language Mapping (i.e., could have been generated by applying the mapping to an OMG IDL definition).

1.2.8 IDL Entity Types

A Java class is a conforming IDL entity type if it extends **`org.omg.CORBA.portable.IDLEntity`** and conforms to the rules specified in the Java Language Mapping (i.e., could have been generated by applying the mapping to an OMG IDL definition) and is not an OMG IDL user exception.

1.3 The IDL Mapping

1.3.1 Overview

This section defines the mapping between RMI/IDL data types and OMG IDL. It includes general rules for mapping Java names to OMG IDL and mappings for:

- Primitive types
- RMI/IDL remote interfaces
- RMI/IDL value types
- RMI/IDL arrays
- RMI/IDL exception types
- CORBA object reference types
- IDL entity types
- Java types that are referenced in RMI/IDL remote interfaces or inherited by RMI/IDL value types, but which are not themselves conforming RMI/IDL types.
- RMI/IDL abstract interfaces
- RMI/IDL implementation classes

1.3.1.1 Summary of Special Case Mappings

Some standard Java class and interface types benefit from special case mappings to specific CORBA types. These are described in the appropriate sections below, but for convenience Table 1-1 summarizes these mappings:

Table 1-1 Special Case Mappings

Java	OMG IDL
`java.lang.Object`	**::java::lang::_Object**
`java.lang.String`	**::CORBA::WStringValue** or **wstring**[1]
`java.lang.Class`	**::javax::rmi::CORBA::ClassDesc**
`java.io.Serializable`	**::java::io::Serializable**
`java.io.Externalizable`	**::java::io::Externalizable**
`java.rmi.Remote`	**::java::rmi::Remote**
`org.omg.CORBA.Object`	**Object**

1. **`String`** constants are mapped differently than **`String`** variables. See Section 1.3.5.11, “Mapping for java.lang.String,” on page 1-21.

1.3.2 Mapping Java Names to IDL Names

In general, each Java name is mapped to an equivalent OMG IDL name. However, there are some exceptions when the Java name is not a legal identifier in OMG IDL.

1.3.2.1 Mapping packages to modules

We map Java package names to OMG IDL modules. Each Java package becomes a separate OMG IDL module. Packages within packages are represented as modules within modules.

So a Java package **`a.b.c`** would turn into an OMG IDL module **::a::b::c**.

1.3.2.2 Java names that clash with IDL keywords

For Java names that collide with OMG IDL keywords, the Java names are mapped to OMG IDL by adding a leading underscore. So the Java name **`oneway`** is mapped to the OMG IDL identifier **_oneway** (an escaped identifier).

1.3.2.3 Java names with leading underscores

For Java names that have leading underscores, the leading underscore is replaced with “J_”. So **`_fred`** is mapped to **J_fred**.

1.3.2.4 Java names with illegal IDL identifier characters

Given the current lack of support for Unicode in OMG IDL, we define a simple name mangling scheme to support the mapping of Java identifiers to OMG IDL identifiers.

For Java identifiers that contain illegal OMG IDL identifier characters such as '$' or Unicode characters outside of ASCII, any such illegal characters are replaced by "U" followed by the 4 hexadecimal characters (in upper case) representing the Unicode value. So, the Java name **`a$b`** is mapped to **`aU0024b`** and **`x\u03bCy`** is mapped to **`xU03BCy`**.

1.3.2.5 Names for inner classes

When mapping names for Java inner classes, a composite name is formed by concatenating the name for the outer class, two underscores, and the name of the inner class. The corrections for illegal OMG IDL identifiers described above are then applied.

For example, an inner class **`Fred`** inside a class **`Bert`** will get mapped to an OMG IDL name of **Bert__Fred**.

1.3.2.6 Overloaded method names

If a Java RMI/IDL method isn't overloaded, then the same method name is used in OMG IDL as was used in Java.

Given the absence of overloaded methods in current OMG IDL, we define a simple name mangling for overloaded methods.

Note that a method may be uniquely defined in a base interface (and therefore its name will not be mangled in that interface) and then be overloaded in a derived interface (in which case the name will be mangled in the derived interface).

For overloaded RMI/IDL methods, the mangled OMG IDL name is formed by taking the Java method name and then appending two underscores, followed by each of the fully qualified OMG IDL types of the arguments (removing any leading "::" and replacing embedded "::" with "_") separated by two underscores. Any spaces (such as in the OMG IDL type **long long**) are replaced with underscores, and any leading underscores in OMG IDL escaped identifiers are removed.

For example, the four overloaded Java methods:

```
void hello();
void hello(int x, a.b.c y, int z);
void hello(int z[]);
void hello(Object o);
```

are mapped to the OMG IDL methods:

```
void hello__();
void hello__long__a_b_c__long(in long x, in ::a::b::c y, in long z);
```

```
void hello__org_omg_boxedRMI_seq1_long(
                    in ::org::omg::boxedRMI::seq1_long x);
void hello__java_lang_Object(in ::java::lang::_Object o);
```

1.3.2.7 Names differing only in case

While Java supports case-sensitive names, OMG IDL does not. Therefore, a general name mangling rule is provided to allow unique OMG IDL identifiers to be generated for Java names that differ only in case.

To simplify the mapping, the use of Java package names differing only in case is not supported. Nor do we support the use of class or interface names within the same package that differ only in case. Both of these are treated as errors.

For other case-sensitive collisions, the rule is that if two (or more) names that need to be defined in the same OMG IDL name scope differ only in case, then a mangled name is generated consisting of the original name followed by an underscore, followed by an underscore separated list of decimal indices into the string, where the indices identify all the upper case characters in the original string. Indices are zero based.

Thus if a Java remote interface has methods **`jack, Jack`** and **`jAcK`** these names are mapped to **jack_, Jack_0,** and **jAcK_1_3**.

1.3.2.8 Method names that collide with other names

In some cases, applying these rules for name mappings would generate OMG IDL with collisions between method names and constant or field names. This is because Java constants and fields can have the same names as methods, but OMG IDL constants and fields cannot. The following rules are used to avoid such name collisions in OMG IDL:

- Method names are mapped unchanged (subject to other mangling rules).
- Java constant or field names whose mapped name collides with the mapped name of a Java method (or would collide if the Java method were mapped to OMG IDL) are mapped with an additional trailing underscore.

For example, if a Java class has both a constant **`foo`** and a method **`foo`**, the OMG IDL method is called **foo** (if it is mapped) and the OMG IDL constant is called **foo_** (whether or not the method **foo** is mapped).

1.3.2.9 Container names that clash with their members

In some cases, applying these rules for name mappings would generate OMG IDL with collisions between a container name and members of the container. This is because a Java member can have the same name as its container, but OMG IDL members cannot. The following rules are used to avoid such name collisions in OMG IDL:

- Container names are mapped unchanged (subject to other mangling rules).
- Java method, constant, or field names whose mapped name collides with the mapped name of their Java container are mapped with an additional trailing underscore.

For example, if a remote Java interface **Foo** has a method **foo**, the OMG IDL interface is called **Foo** and the OMG IDL operation is called **foo_**.

1.3.2.10 Names that would cause OMG IDL name collisions

If the name mappings defined in this specification would produce OMG IDL method, constant, field, or attribute names that are not unique within their declared scope, this is treated as an error. For example, if a Java remote interface has methods **foo()**, **foo(int x)**, and **foo__long()**, the corresponding OMG IDL names would be **foo__**, **foo__long**, and **foo__long**, which is not legal OMG IDL.

1.3.3 Mappings for Primitive Types

Here are the OMG IDL mappings for the Java primitive types:

Java	OMG IDL
void	**void**
boolean	**boolean**
char	**wchar**
byte	**octet**
short	**short**
int	**long**
long	**long long**
float	**float**
double	**double**

The mappings for the Java **void**, **boolean**, **short**, **int**, **long**, **float**, and **double** types are straightforward as they have exact OMG IDL analogues.

The 8 bit signed Java type **byte** is mapped to the 8 bit unsigned OMG IDL type **octet**. The mapping is bit-for-bit so that Java byte value “-1” is transmitted as GIOP octet “0xFF,” and the GIOP octet “0xFF” is mapped back to the Java byte value “-1.” Thus when using this mapping, we will preserve full value and sign information when using RMI/IDL between a Java client and a Java server over GIOP.

The 16 bit Java Unicode **char** type is mapped to the OMG IDL **wchar** type.

1.3.4 Mapping for RMI/IDL Remote Interfaces

An RMI/IDL remote interface is mapped into an OMG IDL interface with the corresponding name (see Section 1.3.2, “Mapping Java Names to IDL Names,” on page 1-7) in the OMG IDL module corresponding to the Java interface’s package name (see Section 1.3.2.1, “Mapping packages to modules,” on page 1-7).

1.3.4.1 Special case for java.rmi.Remote

As a special case, any explicit use of **`java.rmi.Remote`** as a parameter, result, or field is mapped to the OMG IDL type **::java::rmi::Remote**, which is defined as follows:

```
// IDL
module java {
module rmi {
    typedef Object Remote;
};
};
```

All RMI/IDL remote interfaces inherit from **`java.rmi.Remote`**. This inheritance is represented in the RMI to OMG IDL mapping as the implicit inheritance of IDL interface types from **CORBA::Object**.

1.3.4.2 Inherited interfaces

Each inherited interface (other than **`java.rmi.Remote`**) in the Java interface is represented by an equivalent inherited interface in the OMG IDL interface. If the inherited interface is an RMI/IDL remote interface, then it is mapped as specified here. If not, it is mapped as specified in Section 1.3.11, “Mapping Abstract Interfaces,” on page 1-30.

1.3.4.3 Property accessor methods

Methods that follow the JavaBeans™ design patterns for simple read-write properties or simple read-only properties are mapped to OMG IDL interface attributes. No special mapping is done for indexed properties or write-only properties.

Read-Write properties

If an RMI/IDL remote interface has a pair of methods **`get<name>`** and **`set<name>`** where

- the **`get<name>`** method has no arguments,
- the **`set<name>`** method has a single argument and a void return type,
- the result type of the **`get<name>`** method is the same as the argument type of the **`set<name>`** method,
- **`get<name>`** and **`set<name>`** do not throw any checked exceptions except for **`java.rmi.RemoteException`** and its subclasses,

then this is mapped to an OMG IDL read-write attribute where the attribute has the OMG IDL type corresponding to the **`set<name>`** method’s argument type.

Read-only properties

If there is a **get<name>** method that

- has no arguments,
- has a non-void return type,
- does not throw any checked exceptions except for **java.rmi.RemoteException** and its subclasses,

but if there is no corresponding **set<name>** method that satisfies the rules defined in "Read-Write properties" on page 1-11, then the **get<name>** method is mapped to a read-only OMG IDL attribute whose type is obtained by mapping the method's return type.

Boolean properties

For boolean properties an **is<name>** method may take the place of the **get<name>** method. For example, a pair of methods, as shown below, define a read-write attribute foo.

```
boolean isFoo() throws java.rmi.RemoteException;
void setFoo(boolean b) throws java.rmi.RemoteException;
```

The **is<name>** method may be provided instead of a **get<name>** method, or it may be provided in addition to a **get<name>** method. In either case, if the **is<name>** method is present for a boolean property then **is<name>** will be mapped to the OMG IDL attribute **<name>** and **get<name>** (if present) will be mapped to an OMG IDL operation **get<name>**. For example, the following Java methods:

```
// Java
boolean getBar();
boolean isBar();
void setBar(boolean x);
```

are mapped to the following OMG IDL:

```
// IDL
boolean getBar();
attribute boolean bar;
```

Attribute names

The JavaBeans design pattern for property names is that the property name is obtained from the method name(s) by:

- Extracting the characters after the initial "get," "is," or "set" of the method name.
- Converting the first character to lower case unless both the first and second characters are upper case.

So the **getFoo** method implies a "foo" property, the **setX** method implies an "x" property, and the **getURL** method implies a "URL" property.

The OMG IDL attribute name is obtained by taking the JavaBeans property name and applying the normal mapping rules (see Section 1.3.2, “Mapping Java Names to IDL Names,” on page 1-7). However, if this OMG IDL attribute name conflicts with an OMG IDL method name, then an extra pair of underscores “__” is added to the end of the attribute name to attempt to disambiguate it.

1.3.4.4 Methods

Except for property accessors (see Section 1.3.4.3, “Property accessor methods,” on page 1-11), each method in the interface is mapped to an OMG IDL method where:

1. The OMG IDL method name is generated as described in Section 1.3.2.6, “Overloaded method names,” on page 1-8.
2. The Java return type is mapped to the corresponding OMG IDL return type.
3. Each Java argument is mapped to an OMG IDL **in** parameter with the corresponding OMG IDL type.
4. The OMG IDL parameters may be given arbitrary names, but it is recommended that, where possible, the OMG IDL names should be obtained by mapping the Java argument names.[2]
5. Each declared RMI/IDL exception (other than **`java.rmi.RemoteException`** and its subclasses) is mapped to the corresponding OMG IDL exception.
6. **`java.rmi.RemoteException`** and its subclasses, and unchecked exception classes, are assumed to be mapped to the implicit CORBA system exception, and are therefore not explicitly declared in OMG IDL.

1.3.4.5 Constants

Compile-time constants (“**`public final static`**” fields with compile-time constant values) for primitive types and **`Strings`** are mapped to similarly named IDL constants in the target interface with the same values, except for byte constants which are mapped bit-for-bit. For example, -1 maps to 255. Individual **wstring** and **wchar** character values may need to be escaped as defined in the OMG IDL specification.

2. This is not always possible, since Java method argument names do not appear in the .class file output from the javac compiler.

1.3.4.6 Repository ID

A **#pragma ID** is generated to assign each mapped OMG IDL interface type an RMI Hashed format repository ID derived from the Java interface name using the rules specified in *The Common Object Request Broker: Architecture and Specifications, Interface Repository* chapter, with a hash code of zero and no SUID. See Section 1.3.5.7, “Repository ID,” on page 1-18 for more information.

1.3.4.7 An example

Here is an example of an RMI/IDL remote interface:

```
// Java
package alpha.bravo;
public interface Wombat extends java.rmi.Remote,
                omega.Wallaby {
   String BLEAT_CONSTANT = “bleat”;
   void chirp(int x) throws RemoteException;
   void buzz() throws RemoteException, omega.MammalOverload;
   int getFoo() throws RemoteException;
   void setFoo(int x) throws RemoteException;
   String getURL() throws RemoteException;
   void eat() throws Exception;
   void drink() throws RemoteException,
                       java.rmi.NoSuchObjectException;
}
```

that gets mapped to the following IDL:

```
// IDL
module alpha {
module bravo {
    interface Wombat: ::omega::Wallaby {
        const wstring BLEAT_CONSTANT = “bleat”;
        void chirp(in long arg0);
        void buzz() raises (::omega::MammalOverloadEx);
        attribute long foo;
        readonly attribute ::CORBA::WStringValue URL;
        void eat() raises (::java::lang::Ex);
        void drink();
    };
#pragma ID Wombat “RMI:alpha.bravo.Wombat:0000000000000000”
};
};
```

Note that **`String`** constants are mapped differently than **`String`** variables. See Section 1.3.5.11, “Mapping for java.lang.String,” on page 1-21.

1.3.5 Mapping for RMI/IDL Value Types

This section covers the general mapping for RMI/IDL value types, including inner classes and conforming exception classes that are not RMI/IDL exception types. However, note that there are special case mappings for **`java.lang.String`** (see Section 1.3.5.11, “Mapping for java.lang.String,” on page 1-21) and **`java.lang.Class`** (see Section 1.3.5.12, “Mapping for java.lang.Class,” on page 1-21).

RMI/IDL value classes that implement **`org.omg.CORBA.portable.IDLEntity`** and **`org.omg.CORBA.portable.ValueBase`** directly or indirectly are not mapped to OMG IDL, because these Java classes correspond to existing OMG IDL value types that were mapped to Java using the OMG IDL to Java mapping. Instead, the original OMG IDL definitions are used.

Exception classes that implement **`org.omg.CORBA.portable.IDLEntity`** may appear only in Java **`throws`** clauses. This is because they correspond to existing OMG IDL **exception** types, and OMG IDL **exception** types may appear only in IDL **raises** clauses.

Each RMI/IDL value class (except for those mapped from OMG IDL using the OMG IDL to Java mapping) is mapped to an OMG IDL value type with the corresponding OMG IDL name (see Section 1.3.2, “Mapping Java Names to IDL Names,” on page 1-7) in the OMG IDL module corresponding to the Java class’s package name (see Section 1.3.2.1, “Mapping packages to modules,” on page 1-7).

1.3.5.1 Inherited base class

If the RMI/IDL class extends some base class (other than **`java.lang.Object`**), then this inheritance is represented by having the OMG IDL value type inherit from an IDL value type corresponding to the base class. See Section , “module org {,” on page 1-27 for details.

1.3.5.2 Inherited interfaces

Each inherited interface (other than **`java.io.Serializable`** and **`java.io.Externalizable`**) in the Java class is represented by an equivalent inherited or supported type in the mapped OMG IDL type. If the inherited interface is mapped to an OMG IDL abstract valuetype, then it is inherited by the mapped OMG IDL type. If the inherited interface is mapped to an OMG IDL abstract interface, then it is supported by the mapped OMG IDL type. It is not possible for the inherited interface to be mapped to a non-abstract OMG IDL interface, because RMI/IDL value types cannot implement RMI/IDL remote interfaces (see Section 1.2.4, “RMI/IDL Value Types,” on page 1-4). See Section , “module org {,” on page 1-27 for details of how inherited interfaces are mapped.

1.3.5.3 Methods

It is not required that methods in RMI/IDL value classes be mapped into OMG IDL.

This is partly due to concern that an automatic mapping would have a spaghetti effect, where referencing a single value type would result in mappings for methods that would pull in other RMI/IDL types, that would pull in other value types.

In addition, many of the methods in common Java value types cannot be mapped usefully to OMG IDL (because they reference non RMI/IDL types) or to other languages.

However, there may be cases where it is useful to map value type methods to OMG IDL and tools may choose to support options to map methods. In those cases, each mapped method in a Java value type is mapped to an OMG IDL method using the rules specified in Section 1.3.4.3, “Property accessor methods,” on page 1-11 and Section 1.3.4.4, “Methods,” on page 1-13.

Java private methods are not mapped to OMG IDL.

1.3.5.4 Constructors

As with methods, it is not required that RMI/IDL value type constructors be mapped to OMG IDL. However, in those cases where constructors are mapped to OMG IDL (including the default constructor, if any), we require that the following mapping be used:

Each mapped constructor in a Java value type is mapped to an OMG IDL initializer where:

1. If there is a single IDL initializer, its name is **create**. If there are multiple IDL initializers, this name is mangled as specified in Section 1.3.2.6, “Overloaded method names,” on page 1-8.
2. Each Java argument is mapped to an IDL **in** parameter with the corresponding IDL type.
3. The OMG IDL parameters may be given arbitrary names, but it is recommended that, where possible, the OMG IDL names should be obtained by mapping the Java argument names.
4. Each declared RMI/IDL exception type (other than **`java.rmi.RemoteException`** and its subclasses) is mapped to the corresponding OMG IDL exception.
5. **`java.rmi.RemoteException`** and its subclasses, and unchecked exception classes, are not explicitly declared in OMG IDL.

Java private constructors are not mapped to OMG IDL.

For example, the Java classes:

```
// Java
public class foo implements java.io.Serializable {
    foo(int x);
}
public class bar implements java.io.Serializable {
```

```
    bar(int x);
    bar(char y);
}
```

would be mapped to the OMG IDL valuetypes:

```
// IDL
valuetype foo {
  factory create(in long x);
};
valuetype bar {
  factory create__long(in long x);
  factory create__wchar(in long y);
};
```

1.3.5.5 Constants

Compile-time constants ("**`public final static`**" fields with compile-time constant values) for primitive types and **`Strings`** are mapped to similarly named IDL constants in the target value type with the same values. Individual **wstring** and **wchar** character values may need to be escaped as defined in the OMG IDL specification.

1.3.5.6 Data

If the class implements **`java.io.Externalizable`**, then the serialized state of the Java class is treated as an opaque type, and it is defined as an OMG IDL **"custom valuetype**." Java non-static non-transient **`public`** fields are mapped to OMG IDL **public** data members, and other Java fields are not mapped.

If the class does not implement **`java.io.Externalizable`** but does have a **`writeObject`** method,, or extends such a class directly or indirectly, then it is mapped to an OMG IDL "**custom valuetype**" using the rules for mapping data members specified below. An additional IDL custom valuetype in the module **::org::omg::customRMI** is also generated to assist with marshaling and unmarshaling instances of the class. See Section 1.3.5.8, "Secondary custom valuetype," on page 1-19 for details. In this case and for Java classes that implement **`java.io.Externalizable`**, all the semantics of **`java.io.ObjectOutputStream`** and **`java.io.ObjectInputStream`** supported by RMI over JRMP are supported over IIOP.

If the class does not implement **`java.io.Externalizable`** and has a declared **`private static final`** field named **`serialPersistentFields`** of type **`java.io.ObjectStreamField[]`**, then the mapping of data fields to OMG IDL is governed by the value of that field. If the Java class has no **`writeObject`** method, then each **`ObjectStreamField`** instance in the array must correspond to a declared field in the class with the same name and the same declared type. For each **`ObjectStreamField`** instance **`osf`** in the array, there is an OMG IDL data member with name equal to **`osf.getName()`** and type equal to the standard mapping of the

Java type **`osf.getType().getName()`** to OMG IDL. If the corresponding field exists in the Java class and is declared **`public`**, then the OMG IDL field is also declared **public**; otherwise, the OMG IDL field is declared **private**.

If the class does not implement **`java.io.Externalizable`** and does not have a declared **`private static final`** field named **`serialPersistentFields`** of type **`java.io.ObjectStreamField[]`**, then each non-static non-transient field of the Java class is mapped to a corresponding OMG IDL data member with the same name, with the corresponding OMG IDL type. Java **`public`** fields are mapped to OMG IDL **public** data members. Non-public Java fields are mapped to OMG IDL **private** data members.

The following rules apply to the ordering of fields in an OMG IDL value type mapped from Java.

- All non-constant fields whose Java type is a primitive precede all other non-constant fields.
- The non-constant primitive fields are ordered by sorting their Java field names in increasing order. The sort compares the field name strings lexicographically. The comparison is based on the Unicode value of each character in the strings.
- The non-constant non-primitive fields are ordered by sorting their Java field names in the same way as non-constant primitive fields.

1.3.5.7 Repository ID

To allow reliable detection of version mismatches, a **#pragma ID** is generated to assign each value type a specific repository ID string with a specific version string.

The syntax of the repository ID is the standard OMG RMI Hashed format, with an initial “RMI:” followed by the Java class name, followed by a hash code string, followed optionally by a serialization version UID string.

For Java identifiers that contain illegal OMG IDL identifier characters such as ‘$’ or Unicode characters outside of ISO Latin 1, any such illegal characters are replaced by “\U” followed by the 4 hexadecimal characters (in upper case) representing the Unicode value. The use of a “\” is legal within a repository ID and it allows a reliable demangling from a repository ID back to the Java class name.

For example, the Java type **`java.util.Hashtable`** would be mapped to the OMG IDL type **::java::util::Hashtable** with a repository ID of **“RMI:java.util.Hashtable:C03324C0EA357270:13BB0F25214AE4B8”**.

Similarly, a Java class **`a.x\u03bCy`** might be mapped to the OMG IDL type **::a::xU03BCy** with repository ID **“RMI:a.x\U03BCy:0123456789ABCDEF:123456789ABCDEF0”**.

1.3.5.8 Secondary custom valuetype

In addition to the primary mapping described above, an RMI/IDL value type containing a **`writeObject`** method is mapped to a secondary IDL custom valuetype. The module name for this valuetype is formed by taking the **::org::omg::customRMI** prefix and then adding the primary mapped type's module name. The name of the secondary valuetype is the same as the name of the primary IDL custom value type to which the RMI/IDL value type was mapped. The secondary valuetype has no inheritance, data members, methods, or initializers. It has a **#pragma ID** specifying a repository ID formed by taking the repository ID of the primary custom valuetype and prefixing the Java package name with **`"org.omg.customRMI."`**. The secondary custom valuetype represents the enclosure of **`writeObject`** data that is wriitten to the serialization stream when the primary custom valuetype or any of its subclasses is serialized using format version 2, as described in item 1d of Section 1.4.10, "Custom Marshaling Format," on page 1-40.

For IDL custom marshaling and unmarshaling of the primary mapped IDL valuetype, the **marshal** and **unmarshal** methods can call **write_Value()** and **read_Value()** to write and read the nested valuetype enclosure. This will cause the **marshal** and **unmarshal** methods of the secondary mapped IDL valuetype to be called to write and read the custom serialized data.

1.3.5.9 Example without writeObject

The RMI/IDL value type:

```
// Java
package alpha.bravo;
public class Hedgehog extends Warthog
                implements java.io.Serializable {
   public final static short MAX_WARTS = 12;
   private int length;
   protected boolean foobah;
   int height;
   public int size;
   public void snuffle() { ... }
   public int getLength() { ... }
}
```

gets mapped to the IDL value type:

```
// IDL
module alpha {
module bravo {
    valuetype Hedgehog: ::alpha::bravo::Warthog {
        const short MAX_WARTS = 12;
        private boolean foobah;
        private long height;
        private long length_;
        public long size;
```

```
        // mapping of methods, attributes, and initializers is optional
        void snuffle();
        readonly attribute long length();
        factory create();
    };
#pragma ID Hedgehog
    "RMI:alpha.bravo.Hedgehog:12345678ABCDEF00:0123456789ABCDEF"
};
};
```

1.3.5.10 Example with writeObject

The RMI/IDL value type:

```
// Java
package alpha.bravo;
public class Kangaroo extends Wallaby
                implements java.io.Serializable {
   private int length;
   private Kangaroo(int length) { ... }
   private void writeObject(java.io.ObjectOutputStream s)
      { ... }
   public int hop() { ... }
}
```

gets mapped to the IDL value types:

```
// IDL
module alpha {
module bravo {
    custom valuetype Kangaroo: ::alpha::bravo::Wallaby {
        private long length;
        // mapping of methods shown below is optional
        long hop();
    };
#pragma ID Kangaroo
    "RMI:alpha.bravo.Kangaroo:87654321ABCDEF01:9876543210FEDCBA"
};
};

module org {
module omg {
module customRMI {
module alpha {
module bravo {
    custom valuetype Kangaroo {};
#pragma ID Kangaroo
    "RMI:org.omg.customRMI.alpha.bravo.Kangaroo:87654321ABCDEF01:
9876543210FEDCBA"
};
```

```
};
};
};
};
```

1.3.5.11 Mapping for java.lang.String

When used as a parameter type, return type, or data member, the Java **`String`** type is mapped to the type **::CORBA::WStringValue**. However when mapping Java **`String`** constant definitions, a Java **`String`** is simply mapped to a **wstring**.

::CORBA::WStringValue is a standard type that is part of the **CORBA** module. It is defined as

```
valuetype WStringValue wstring;
```

which is semantically equivalent to:

```
valuetype WStringValue {
    public wstring data;
};
```

1.3.5.12 Mapping for java.lang.Class

When used as a parameter type, return type, or data member, the Java **`Class`** type is mapped to the OMG IDL type **::javax::rmi::CORBA::ClassDesc**. This OMG IDL type is the result of mapping the following Java class to OMG IDL:

```
// Java
package javax.rmi.CORBA;
public class ClassDesc implements java.io.Serializable {
   public String repid;
   public String codebase; // space-separated list of URLs
   static final long serialVersionUID
                     = -3477057297839810709L;
}
```

1.3.6 Mapping for RMI/IDL Arrays

An RMI/IDL array is mapped to a "boxed" value type containing an IDL sequence. We use the syntax "**valuetype xyz foo**" as a shorthand for defining a value type named "**xyz**" that contains a single field of type "**foo**."

The module for each such value type is determined by the IDL type of the array element. For multi-dimensional arrays, this is the type of the innermost array element, after all the dimensions are resolved.

Primitive OMG IDL types such as **long**, **boolean**, etc. are mapped directly into the **::org::omg::boxedRMI** module. For other types, a module name is formed by taking the **::org::omg::boxedRMI** prefix and then adding the type's existing module name to identify a sub-module. So the type **::a::b::c** is mapped into the module **::org::omg::boxedRMI::a::b**.

For each "boxed" value type generated for a Java array, a **#pragma ID** is generated to specify an RMI Hashed format repository ID for the IDL type.

The OMG IDL value type name within the module is formed by prefixing the OMG IDL element type name with "**seq<n>**_" where **<n>** is the number of dimensions of the array. Any spaces (such as in the OMG IDL type **long long**) are replaced with underscores.

Some example value definitions resulting from Java arrays:

`boolean[]` => in the module **::org::omg::boxedRMI** the definition:

valuetype seq1_boolean sequence<boolean>;

`long[]` => in the module **::org::omg::boxedRMI** the definition:

valuetype seq1_long_long sequence<long long>;

`a.b.C[]` => in the module **::org::omg::boxedRMI::a::b** the definition:

valuetype seq1_C sequence<::a::b::C>;

`x.Y[][]` => in the module **::org::omg::boxedRMI::x** the definitions:

valuetype seq1_Y sequence<::x::Y>;
valuetype seq2_Y sequence<seq1_Y>;

1.3.6.1 Preventing redefinitions of boxed sequence types

Each generated boxed sequence type must be protected against multiple definitions and there are various ways in which this could be accomplished. For example, each generated boxed sequence type could be wrapped in an **#ifndef** and **#endif** pair where the tag of the **#ifndef** is the fully scoped name of the sequence value type, replacing the leading '**::**' with two underbars, replacing each inner '**::**' with one underbar, and adding two underbar characters at the end. The **#ifndef** would be followed by a **#define** of the tag, followed by the sequence definition, followed by an **#endif**.

A definition for a sequence of **boolean** that uses this approach would be wrapped in a preamble of

#ifndef __org_omg_boxedRMI_seq1_boolean__
#define __org_omg_boxedRMI_seq1_boolean__

and would be followed by an

#endif

1.3.6.2 Array example

Here's a more complete example. The Java definition:

```
// Java
package alpha.bravo;
public class Charlie implements java.io.Serializable {
   public omega.Dolphin fins[];
}
```

would result in the following OMG IDL definition:

```
// IDL
#ifndef __org_omg_boxedRMI_omega_seq1_Dolphin__
#define __org_omg_boxedRMI_omega_seq1_Dolphin__
module org {
module omg {
module boxedRMI {
module omega {
    valuetype seq1_Dolphin sequence<::omega::Dolphin>;
#pragma ID seq1_Dolphin
    "RMI:[Lomega.Dolphin;:ABCDEF0123456789:01ABCDEF23456789"
};
};
};
};
#endif

module alpha {
module bravo {
    valuetype Charlie {
        public ::org::omg::boxedRMI::omega::seq1_Dolphin fins;
    };
#pragma ID Charlie
    "RMI:alpha.bravo.Charlie:0123456789ABCDEF:ABCDEF9876543210"
};
};
```

1.3.7 Mapping RMI/IDL Exceptions

OMG IDL does not allow subclassing of exception types. By contrast Java programmers tend to make heavy use of exception subclassing, and the Java type system is used to distinguish different flavors of exceptions at run time. It is very common for a Java interface to say it raises a fairly generic exception (such as **`java.io.IOException`**) but for implementations to throw more specific sub-types (such as **`java.io.InterruptedIOException`**) and for clients to use the Java **`instanceof`** operator to check for specific subtypes. In addition, RMI/IDL exceptions can be passed as normal value types, whereas OMG IDL exceptions can only be used in **`raises`** clauses.

This mismatch of exception styles makes the mapping of RMI/IDL exception types to OMG IDL problematic.

To allow full support for subclassing when communicating Java to Java we use a mapping where an RMI/IDL exception type is mapped to both a specific OMG IDL exception and to an OMG IDL value type that allows subclassing. The OMG IDL exception has a single field that holds the corresponding value object.

This solution allows RMI/IDL to support the normal idiomatic use of Java exceptions, while still being correctly mappable into OMG IDL.

1.3.7.1 The IDL value type

Each RMI/IDL exception type is mapped to an OMG IDL value type in the OMG IDL module corresponding to the Java exception's package name (see Section 1.3.2.1, "Mapping packages to modules," on page 1-7). The value type's name is formed by taking the RMI/IDL exception name and applying the normal corrections for illegal IDL names (see Section 1.3.2, "Mapping Java Names to IDL Names," on page 1-7).

The OMG IDL value type inherits from an OMG IDL parent value type that corresponds to the base class of the RMI/IDL exception class. If an RMI/IDL exception type **`Fred`** extends **`Bert`**, then its OMG IDL value type **Fred** will inherit **Bert**.

The mapping of the fields, methods, constants, and inherited interfaces to the OMG IDL value type follow the same rules defined for other RMI/IDL value types in Section 1.3.5.2, "Inherited interfaces," on page 1-15 through Section 1.3.5.7, "Repository ID," on page 1-18.

1.3.7.2 The IDL exception

Each RMI/IDL exception type is also mapped to an OMG IDL exception in the OMG IDL module corresponding to the Java exception's package name (see Section 1.3.2.1, "Mapping packages to modules," on page 1-7). The OMG IDL exception name is formed from the Java exception name by

- removing any trailing "**`Exception`**" suffix.
- adding an "**Ex**" at the end of the name.
- applying the normal corrections for illegal OMG IDL names (see Section 1.3.2, "Mapping Java Names to IDL Names," on page 1-7).

If applying the above rules yields the same OMG IDL name for more than one Java exception name (e.g., there are Java exception names **`foo`** and **`fooException`**, which both map to the OMG IDL name **fooEx**), then this is treated as an error.

For example:

`java.lang.IllegalAccessException` is mapped to **::java::lang::IllegalAccessEx**

`alpha.bravo.Foo` is mapped to **::alpha::bravo::FooEx**

This OMG IDL exception name can then be used in the **raises** clause of OMG IDL method definitions.

The OMG IDL exception type is defined with a single data member named **value** that has the type of the associated value object.

1.3.7.3 Mapping References to RMI/IDL Exceptions

Whenever an RMI/IDL exception is used in a Java **throws** clause, it is mapped to a use of the corresponding OMG IDL exception type in the OMG IDL **raises** clause.

Whenever an RMI/IDL exception is used as a data field or as a method argument, it is mapped to the corresponding OMG IDL value type.

1.3.7.4 Example

The Java RMI/IDL definitions:

```
// Java
package omega;
public class FruitbatException extends MammalException {
   public FruitbatException(String message, int count) {
      ...
   }
   public int getCount() { ... }
   private int count;
}

public interface Thrower extends java.rmi.Remote {
   void doThrowFruitbat() throws FruitbatException,
            RemoteException;
   FruitbatException getLastException()
                              throws RemoteException;
}
```

are mapped to OMG IDL as:

```
// IDL
module omega {
    valuetype FruitbatException: ::omega::MammalException {
        private long count_;
        // mapping of attributes shown below is optional
        readonly attribute long count();
    };
#pragma ID FruitbatException
    "RMI:omega/FruitbatException:1234567899775511:3344556645678901"

    exception FruitbatEx {
        FruitbatException value;
    };
```

```
    interface Thrower {
        void doThrowFruitbat() raises (FruitbatEx);
        readonly attribute FruitbatException lastException;
    };
#pragma ID Thrower "RMI:omega.Thrower:0000000000000000"
};
```

1.3.8 Mapping CORBA Object Reference Types

A CORBA object reference type is mapped directly to its corresponding OMG IDL interface or to **Object** if it is **`org.omg.CORBA.Object`**.

1.3.9 Mapping IDL Entity Types

An IDL entity type that is not a CORBA object reference type is mapped to a "boxed" value type containing the IDL entity type, except as specified in Section 1.3.5, "Mapping for RMI/IDL Value Types," on page 1-15 and Section 1.3.10, "Mapping for Non-conforming Classes and Interfaces," on page 1-27.

The containing module for the boxed type is determined by the IDL entity type's containing module. A module name is formed by taking the **::org::omg::boxedIDL** prefix and appending the IDL entity type's fully scoped IDL module name. A boxed value type corresponding to the IDL entity type is defined within this module. The name of the value type is the same as the name of the IDL definition it is boxing.

For example, assume we have the following IDL and the Java class that results from applying the forward mapping:

```
// IDL
module hello {
    struct world {
        short x;
    };
};

// Java
package hello;
public final class world implements
                org.omg.CORBA.portable.IDLEntity {
...
}
```

Now assume that **`hello.world`** is used as an argument to a method or as a member of an RMI/IDL value type. The Java class **`hello.world`** is mapped as follows:

```
module org {
module omg {
module boxedIDL {
module hello {
    valuetype world ::hello::world;
#pragma ID world “RMI:hello.world:1234567890ABCDEF”
};
};
};
};
```

The exact mechanism by which the IDL for **::hello::world** is created is a tools issue and is not specified.

These generated types must be protected against multiple definitions. See Section 1.3.6.1, “Preventing redefinitions of boxed sequence types,” on page 1-22 for an example of an approach that could be used.

The IDL entity types **`org.omg.CORBA.Any`** and **`org.omg.CORBA.TypeCode`** are mapped as follows:

```
module org {
module omg {
module boxedIDL {
module CORBA {
    valuetype _Any any;
#pragma ID _Any “RMI:org.omg.CORBA.Any:0000000000000000”
};
};
};
};
```

```
module org {
module omg {
module boxedIDL {
module CORBA {
    valuetype _TypeCode ::CORBA::TypeCode;
#pragma ID _TypeCode
    “RMI:org.omg.CORBA.TypeCode:0000000000000000”
};
};
};
};
```

1.3.10 Mapping for Non-conforming Classes and Interfaces

In addition to generating OMG IDL for each conforming RMI/IDL type, OMG IDL definitions are also required for each Java class or interface that

- is inherited (either directly or indirectly) by another Java type that has been mapped to OMG IDL.

- is specified as an argument type or as a result type to an RMI/IDL remote interface method.
- has been mapped to a data member of an OMG IDL value type.

Each such Java class or interface (except for interfaces that extend **`org.omg.CORBA.portable.IDLEntity`** directly or indirectly) is mapped to an OMG IDL type with the corresponding name (see Section 1.3.2, "Mapping Java Names to IDL Names," on page 1-7) in the OMG IDL module corresponding to the Java type's package name (see Section 1.3.2.1, "Mapping packages to modules," on page 1-7).

Java interfaces that extend **`org.omg.CORBA.portable.IDLEntity`** directly or indirectly are not mapped to OMG IDL, because these Java interfaces correspond to existing OMG IDL interfaces that were mapped to Java using the OMG IDL to Java mapping.

Non-conforming Java classes are mapped to OMG IDL abstract value types with no data members. Non-conforming Java interfaces are mapped as follows:

- Java interfaces whose method definitions (including inherited method definitions) all throw **`java.rmi.RemoteException`** or a superclass of **`java.rmi.RemoteException`** are RMI/IDL abstract interfaces. They are mapped to OMG IDL abstract interfaces as described in Section 1.3.11, "Mapping Abstract Interfaces," on page 1-30.
- All other Java interfaces are mapped to OMG IDL abstract value types with no data members.

1.3.10.1 java.io.Serializable and java.io.Externalizable

As a special case, any uses of **`java.io.Serializable`** or **`java.io.Externalizable`** as a parameter, result, or field are mapped to the OMG IDL types **::java::io::Serializable** and **::java::io::Externalizable** respectively.

These OMG IDL types are defined as follows:

```
// IDL
module java {
module io {
    typedef any Serializable;
    typedef any Externalizable;
};
};
```

1.3.10.2 Mapping for java.lang.Object

The Java type **`java.lang.Object`** is mapped to the OMG IDL type **::java::lang::_Object**, which is defined as follows:

```
// IDL
module java {
module lang {
    typedef any _Object;
};
};
```

This is used when **`java.lang.Object`** is specified as the type of a parameter, result, or field. All Java classes implicitly inherit from **`java.lang.Object`,** but this implicit inheritance is not exposed as part of the RMI to OMG IDL mapping.

1.3.10.3 Inherited interfaces

Each inherited Java class or interface (other than **`java.io.Serializable`** and **`java.io.Externalizable`**) in the Java type is represented by an equivalent inherited value type or abstract interface type in OMG IDL.

1.3.10.4 Methods and constants

The methods and constants in these classes and interfaces are mapped as specified for value classes in Section 1.3.4.4, “Methods,” on page 1-13 and Section 1.3.4.5, “Constants,” on page 1-13.

1.3.10.5 Examples

The following non-conforming Java types:

```
// Java
package alpha.bravo;
public interface Mammal {
   public int getSize();
}

public class PolarBear {
   private int length;
   public int weight;
   public PolarBear(int length, int weight) { ... }
   public int getSize() { ... }
   public int getWeight() { ... }
}
```

get mapped to the OMG IDL value types:

```
// IDL
module alpha {
module bravo {
    abstract valuetype Mammal {
    };

    abstract valuetype PolarBear {
    };
};
};
```

1.3.11 Mapping Abstract Interfaces

Java interfaces that do not extend **`java.rmi.Remote`** directly or indirectly and whose method definitions (including inherited method definitions) all throw **`java.rmi.RemoteException`** or a superclass of **`java.rmi.RemoteException`** are mapped to OMG IDL abstract interfaces. Java interfaces that do not extend **`java.rmi.Remote`** directly or indirectly and have no methods are also mapped to OMG IDL abstract interfaces.

1.3.11.1 Inherited interfaces

Each inherited Java interface in the Java type is represented by an equivalent inherited abstract interface in the OMG IDL type.

1.3.11.2 Methods and constants

Methods and constants are mapped according to the rules specified in Section 1.3.4.3, “Property accessor methods,” on page 1-11, Section 1.3.4.4, “Methods,” on page 1-13, and Section 1.3.4.5, “Constants,” on page 1-13.

1.3.11.3 Examples

The following Java type:

```
// Java
package alpha.bravo;
public interface Bear {
   public int getSize() throws
                        java.rmi.RemoteException;
}
```

gets mapped to the OMG IDL type:

```
// IDL
module alpha {
module bravo {
```

```
    abstract interface Bear {
        readonly attribute long size();
    };
#pragma ID Bear "RMI:alpha.bravo.Bear:0000000000000000"
};
};
```

1.3.12 Mapping Implementation Classes

In general, mapping RMI implementation classes to OMG IDL is not needed. However, if a given RMI implementation class implements multiple distinct RMI/IDL remote interfaces, then it is necessary to generate an OMG IDL type that represents the unification of the distinct RMI/IDL types.

Any such composite RMI/IDL implementation class is mapped into an OMG IDL interface with the corresponding name (see Section 1.3.2, "Mapping Java Names to IDL Names," on page 1-7) in the OMG IDL module corresponding to the Java class's package name (see Section 1.3.2.1, "Mapping packages to modules," on page 1-7).

Each inherited RMI/IDL remote interface (other than **`java.rmi.Remote`**) inherited by the Java implementation class is represented by an equivalent inherited interface in the OMG IDL interface. Inherited classes and inherited interfaces that are not RMI/IDL remote interfaces are ignored.

At run time, any instances of the composite implementation class must, from a CORBA perspective, implement the corresponding composite OMG IDL interface. This implies, for example, they must return true to any calls of "**`is_a`**" on any of the OMG IDL interfaces associated with the distinct RMI/IDL interfaces.

1.3.12.1 Example

The RMI/IDL implementation class **`alpha.bravo.AB`** that implements the RMI/IDL remote interfaces **`alpha.bravo.A`** and **`alpha.bravo.B`**:

```
// Java
package alpha.bravo;
public class AB extends javax.rmi.PortableRemoteObject
             implements alpha.bravo.A, alpha.bravo.B {
    ...
}
```

is mapped to the OMG IDL:

```
// IDL
module alpha {
module bravo {
    interface AB: ::alpha::bravo::A, ::alpha::bravo::B {
    };
#pragma ID AB "RMI:alpha.bravo.AB:0000000000000000"
```

```
};
};
```

1.4 Run-Time Issues

In addition to the RMI/IDL mapping there are also run-time issues about how to implement Java RMI/IDL calls over GIOP.

1.4.1 Subclasses of Value Objects

It should be possible to send a subclass of an RMI/IDL value type where a base value type was specified in the OMG IDL.

If this occurs, the recipient is responsible for locating a suitable implementation subclass to represent the value object subtype. In cases where a Java virtual machine is available, this might include attempting to load Java bytecodes for the subclass. In the Java to C++ case this might involve attempting to locate a suitable C++ subclass.

The name of the subclass can be obtained by parsing the value object's repository ID, which must be in the standard OMG RMI Hashed format (see Section 1.3.5.7, "Repository ID," on page 1-18).

If a suitable subclass is not available, then the recipient must raise an exception. It is not acceptable for an implementation to attempt to substitute a base class of the subclass value that was transmitted.

1.4.2 Locating Stubs for Remote References

When receiving an IOR from another system, it is the responsibility of the receiving system to know which RMI/IDL type is expected. The receiving system should be prepared to use stubs associated with this RMI/IDL type to manage the received object reference. However, the receiving system may also optionally use the Repository ID of the incoming IOR to locate and use stubs that more accurately reflect the true run-time type of the object reference.

1.4.3 Narrowing

To narrow an RMI/IDL object reference to a different type, application programmers must use the static **`narrow`** method provided by the **`javax.rmi.PortableRemoteObject`** class (see Section 1.6.1, "PortableRemoteObject," on page 1-62).

Thus for example they might do:

```
// Java
alpha.bravo.Mammal m = getMammal();
try {
```

```
   b = (alpha.bravo.Bandicoot)
         javax.rmi.PortableRemoteObject.narrow(
                  m, alpha.bravo.Bandicoot.class);
} catch (ClassCastException ex) {
   ...
}
```

1.4.4 Allocating Ties for Remote Values

Following normal RMI semantics, an RMI server-side implementation object may be passed across an RMI remote interface as though it were a remote reference.

The **`javax.rmi.CORBA.Util.writeRemoteObject`** method checks whether a transmitted object is an implementation object and if so, allocates or reuses a suitable tie object. The type of the tie object should correspond to the OMG IDL type that the implementation object implements.

This tie class is located at run time by finding the class of the implementation object and checking for a corresponding tie class (see Section 1.4.6, “Locating Stubs and Ties,” on page 1-33). If no suitable tie class is found, the check is repeated on the implementation class’s base class and so on up the inheritance chain, excluding **`java.lang.Object`**. If no suitable tie class is found, a marshaling error occurs.

1.4.5 Wide Character Support

Since Java supports Unicode characters and strings, ORBs supporting RMI/IDL must provide some form of wide character support.

Note that as part of IIOP code set negotiation, ORBs are required to accept Unicode UTF16 for use as a fallback transmission format for wide characters, though they may negotiate to use other formats.

1.4.6 Locating Stubs and Ties

At various times it may be necessary for the ORB to locate either a stub class for a given RMI/IDL remote interface or abstract interface, or a tie class for a given RMI/IDL implementation class. The name of the stub class is formed by taking the name of the RMI/IDL interface, prepending “_” and appending "_Stub." The name of the tie class is formed by taking the name of the RMI/IDL implementation class, prepending “_” and appending "_Tie." For RMI/IDL implementation classes that are mapped to IDL (see Section 1.3.12, “Mapping Implementation Classes,” on page 1-31), the name of the stub class for the composite interface is formed by taking the name of the RMI/IDL implementation class, prepending “_” and appending “_Stub.”

The stub class corresponding to an RMI/IDL interface or implementation class may either be in the same package as its associated interface or class, or may be further qualified by the **`org.omg.stub`** package prefix. For example, the stub class for an

RMI/IDL interface class **a.b.Fred** would be named either **a.b._Fred_Stub** or **org.omg.stub.a.b._Fred_Stub**. For an RMI/IDL implementation class **x.y.Z**, the tie class would be named **x.y._Z_Tie**.

When loading a stub class corresponding to an interface or class <packagename>.<typename>, the class <packagename>._<typename>_Stub shall be used if it exists; otherwise, the class org.omg.stub.<packagename>._<typename>_Stub shall be used.

A given Java virtual machine may have several different "class loaders" active simultaneously. Each of these class loaders provides a separate naming context for Java classes. For example, a browser might be running applets from several different hosts. To avoid class name conflicts it will run the applets in different class loaders. Thus, two different applets might both reference a class called **Foo**, but each of them will get its own version of the **Foo** class from its own class loader.

The **java.lang.Class.getClassloader** method returns the class loader for a given **Class**. So given one **Class** it is possible to generate new class names and then attempt to load those additional classes from the original class's class loader.

It is important in Java APIs to use an appropriate class loader when trying to locate a named class. To ease this problem in the ORB Portability APIs we normally pass around **java.lang.Class** objects rather than simply class names. When it is necessary to load named classes, runtime code should take care to use an appropriate class loader (e.g., by using one from an existing **Class** object).

1.4.7 Mapping RMI Exceptions to CORBA Exceptions

To ensure correct RMI exception passing semantics when running over IIOP, all Java exceptions thrown by the server implementation must be passed back to the client. Any exception that is an instance of an RMI/IDL exception type declared by the method or any subclass of such a type (other than **java.rmi.RemoteException** and its subclasses) is marshaled as the mapped IDL exception corresponding to the declared RMI/IDL exception (see Section 1.3.7.2, "The IDL exception," on page 1-24) containing a mapped IDL valuetype corresponding to the actual runtime RMI/IDL exception type (see Section 1.3.7.1, "The IDL value type," on page 1-24). On the client side, the mapped IDL valuetype is unmarshaled and thrown back to the application.

For example, if a method in an RMI/IDL remote interface declares an exception type **MammalException** and its implementation throws an instance of **WombatException** (a subclass of **MammalException**), then this exception is marshaled as an IDL exception **MammalEx** containing an IDL valuetype **WombatException**, and a **WombatException** is thrown to the client application.

All other Java exceptions are marshaled as CORBA UNKNOWN system exceptions whose GIOP Reply message includes an **UnknownExceptionInfo** service context containing the marshaled Java exception thrown by the server implementation. The Java exception is marshaled using the rules for CDR marshaling of value types as defined by the GIOP specification, applied in conjunction with the rules for mapping RMI/IDL value types to IDL as defined in Section 1.3.5, "Mapping for RMI/IDL Value Types," on page 1-15 of this specification.

In order to support versioning of the Java exception marshaled within an **UnknownExceptionInfo** service context, a **SendingContextRunTime** service context must previously have been processed for the connection. If a GIOP message carrying both an **UnknownExceptionInfo** service context and a **SendingContextRunTime** service context is received, and no **SendingContextRunTime** service context has previously been processed for this connection, then the **SendingContextRunTime** service context must be processed before the data within the **UnknownExceptionInfo** service context is unmarshaled

1.4.8 Mapping CORBA System Exceptions to RMI Exceptions

In general CORBA system exceptions are simply mapped to instances of **`java.rmi.RemoteException`**; however, some CORBA system exceptions are mapped to more specific subclasses of **`RemoteException`**. These are listed in Table 1-2.

Table 1-2 CORBA and RMI Exceptions

CORBA Exception	RMI Exception
COMM_FAILURE	**`java.rmi.MarshalException`**
INV_OBJREF	**`java.rmi.NoSuchObjectException`**
NO_PERMISSION	**`java.rmi.AccessException`**
MARSHAL	**`java.rmi.MarshalException`**
BAD_PARAM	**`java.rmi.MarshalException`**
OBJECT_NOT_EXIST	**`java.rmi.NoSuchObjectException`**
TRANSACTION_REQUIRED	**`javax.transaction.TransactionRequiredException`**
TRANSACTION_ROLLEDBACK	**`javax.transaction.TransactionRolledbackException`**
INVALID_TRANSACTION	**`javax.transaction.InvalidTransactionException`**

In all cases, the RMI exception is created with a detail string that consists of:

- the string “CORBA”
- followed by the CORBA name of the system exception
- followed by a space
- followed by the hexadecimal value of the system exception’s minor code
- followed by a space
- followed by the completion status of “Yes,” “No,” or “Maybe.”

Thus a CORBA UNKNOWN system exception with a minor code of 0x31 and a completion status of Maybe would be mapped to a **`RemoteException`** with the following detail string:

`“CORBA UNKNOWN 0x31 Maybe”`

The **RemoteException** returned by **mapSystemException** must preserve the original CORBA system exception as the detail field, except when the original CORBA system exception is BAD_PARAM with a minor code of 6, which is mapped to **java.io.NotSerializableException**.

1.4.8.1 Mapping of UnknownExceptionInfo Service Context

CORBA UNKNOWN exceptions whose GIOP Reply message includes an UnknownExceptionInfo service context containing a marshaled instance of **java.lang.Throwable** or one of its subclasses are mapped to RMI exceptions according to the type of the object contained in the service context, as shown in Table 1-3.

Table 1-3 UnknownExceptionInfo and RMI Exceptions

UnknownExceptionInfo	RMI Exception
java.lang.Error (or subclass)	**java.rmi.ServerError**
java.rmi.RemoteException (or subclass)	**java.rmi.ServerException**
java.lang.RuntimeException (or subclass)	**java.rmi.ServerRuntimeException** (JDK 1.1) **java.lang.RuntimeException** (Java 2)

1.4.9 Code Downloading

Class downloading is supported for stubs, ties, values, and value helpers. The specification has been designed to be implementable using either JDK 1.1.6 or Java 2 APIs, allows transmission of codebase information on the wire for stubs and ties, and enables usage of pre-existing ClassLoaders when relevant.

1.4.9.1 Definitions

"codebase" - A **java.lang.String** containing a space-separated array of URLs (e.g., "http://acme.com/classes" or "http://abc.net/classes http://abc.net/ext/classes").

"localCodebase" - The System Property "java.rmi.server.codebase" whose value is a codebase or null. Defaults to null.

"remoteCodebase" - The codebase transmitted from a remote system. May be null.

"useCodebaseOnly" - The System Property "java.rmi.server.useCodebaseOnly" whose value is either "true" or "false." Defaults to "false." If "true" (ignoring case), any remote codebase is ignored and only the local codebase used.

"loader" - A class loader that specifies a context within which class loading is initiated. May be null.

1.4.9.2 Codebase Selection

The **`Util.getCodeBase(Class clz)`** method (see Section 1.5.1.6, “Util,” on page 1-51) performs codebase selection.

On Java 2, this method returns the same string as **`java.rmi.server.RMIClassLoader.getClassAnnotation(clz)`**

On JDK 1.1, this method works as follows:

1. If the name of **`clz`** has a top-level package qualifier of **`java`**, then return null, else...
2. If **`clz`** has a ClassLoader with a URL security context, then return this URL, else...
3. If there is a security manager with a URL security context, then return this URL, else...
4. Return localCodebase.

When sending RMI/IDL values from Java, the codebase transmitted over GIOP must be the codebase that this method would return for the value's class.

When sending RMI/IDL object references from Java, the codebase transmitted over GIOP is selected by calling the method **`org.omg.CORBA_2_3.portable.ObjectImpl._get_codebase()`** on the stub object.

1.4.9.3 Codebase Transmission

For values and value helpers, the codebase is transmitted after the value tag.

For stubs and ties, the codebase is transmitted as the **TaggedComponent TAG_JAVA_CODEBASE** in the IOR profile, where the **component_data** is a CDR encapsulation of the codebase written as an IDL string. The codebase is a space-separated list of one or more URLs.

In all cases, the **SendingContextRunTime** service context may provide a default codebase that is used if not overridden by a more specific codebase encoded in a valuetype or IOR.

For object references created using **`InputStream.read_Object`** or **`InputStream.read_abstract_interface`**, the transmitted codebase is stored in the object reference (stub) and can be retrieved subsequently using the **`org.omg.CORBA_2_3.portable.ObjectImpl._get_codebase()`** method, described below.

If no codebase was transmitted, localCodebase is stored in the object reference (stub).

1.4.9.4 Codebase Access

In the event that **PortableRemoteObject.narrow()** must load a stub, it needs to call a portable API to extract codebase information from the original stub. This API is also used by the **OutputStream** methods **write_Object** and **write_abstract_interface** to obtain the codebase to be transmitted in the **TAG_JAVA_CODEBASE TaggedComponent**. The API that is provided for these purposes is the **_get_codebase()** method of the **org.omg.CORBA_2_3.portable.ObjectImpl** class. See the *IDL/Java Language Mapping* document.

1.4.9.5 Codebase Usage

The following method (see Section 1.5.1.6, "Util," on page 1-51) is used to load classes.

```
Util.loadClass(String className,
                  String remoteCodebase,
                  ClassLoader loader)
            throws ClassNotFoundException { ... }
```

On Java 2, this method works as follows:

1. Find the first non-null ClassLoader on the call stack and attempt to load the class using this ClassLoader. If this fails...
2. If **remoteCodebase** is non-null and useCodebaseOnly is false, then call **java.rmi.server.RMIClassLoader.loadClass (remoteCodebase, className)**
3. If **remoteCodebase** is null or useCodebaseOnly is true, then call **java.rmi.server.RMIClassLoader.loadClass(className)**
4. If a class was not successfully loaded by step 1, 2, or 3, and **loader** is non-null, then call **Class.forName(className, false, loader)**
5. If a class was successfully loaded by step 1, 2, 3, or 4, then return the loaded class.

On JDK 1.1, this method works as follows:

1. If **className** is an array type, extract the array element type. If this is a primitive type, then call **Class.forName(className)**, else proceed using the array element class name as **className**.
2. Search the call stack for the first non-null ClassLoader. If a ClassLoader is found, then attempt to load the class using this ClassLoader, else attempt to load the class using **Class.ForName(className)**. If this fails...
3. If **remoteCodebase** is non-null and useCodebaseOnly is false, then call **java.rmi.server.RMIClassLoader.loadClass(codebaseURL, className)** for each remote codebase URL in the **remoteCodebase** string until the class is found.

4. If **`remoteCodebase`** is null or useCodebaseOnly is true, then call **`java.rmi.server.RMIClassLoader.loadClass(className)`**
5. If a class was not successfully loaded by step 1, 2, 3, or 4, and **`loader`** is non-null, then call **`loader.loadClass(className)`**
6. If a class was successfully loaded by step 1, 2, 3, 4, or 5, then return the loaded class, unless the **`className`** parameter was a non-primitive array type, in which case return a suitably dimensioned array class for the element class that was loaded.

1.4.10 Custom Marshaling Format

When an RMI/IDL value type is custom marshaled over GIOP, the following data is transmitted:

a. **octet** - Format version. 1 or 2.

For serializable objects with a **writeObject** method:

b. **boolean** - True if **defaultWriteObject** was called, false otherwise.

c. (optional) Data written by **defaultWriteObject**. The ordering of the fields is the same as the order in which they appear in the mapped IDL valuetype, and these fields are encoded exactly as they would be if the class did not have a **writeObject** method.

d. Additional data written by **writeObject**, encoded as specified below. For format version 1, this data is optional and if present must be written "as is". For format version 2, if optional data is present then it must be enclosed within a CDR custom valuetype with no codebase and repid **"RMI:org.omg.custom.<class>"** where **<class>** is the fully-qualified name of the class whose **writeObject** method is being invoked. For format version 2, if optional data is not present then a null valuetype (0x00000000) must be written to indicate the absence of optional data.

For externalizable objects:

b. (optional) Data written by **writeExternal**, encoded as specified below.

Primitive Java types are marshaled as their corresponding IDL primitives (see Section 1.3.3, “Mappings for Primitive Types,” on page 1-10). Java strings written by the **java.io.ObjectOutputStream.writeUTF()** method and read by the **java.io.ObjectInputStream.readUTF()** method are marshaled as IDL **wstring**s. Java **int**s and **String**s written by the **writeByte**, **writeChar**, **writeBytes**, and **writeChars** methods of **java.io.ObjectOutputStream** are marshaled as specified by the definitions of these methods in the **java.io.DataOutput** interface. Other Java objects are marshaled in the form of an IDL abstract interface (i.e., a union with a boolean discriminator containing either an object reference if the discriminator is true or a value type if the discriminator is false).

RMI/IDL stubs, RMI/IDL remote implementations, and IDL stubs are marshaled as object references (IORs). All other Java objects are marshaled as value types. The value type encoding is determined from the object's runtime type by applying the mappings specified in Section 1.3.5, “Mapping for RMI/IDL Value Types,” on page 1-15 and Section 1.3.6, “Mapping for RMI/IDL Arrays,” on page 1-21.

The default custom stream format is 1 for GIOP 1.2 and 2 for GIOP 1.3. For RMI/IDL custom value types marshaled within GIOP requests, a format version not greater than the default for the GIOP message level must be sent, except where the **TAG_RMI_CUSTOM_MAX_STREAM_FORMAT TaggedComponent** (see Section 1.4.11, “TAG_RMI_CUSTOM_MAX_STREAM_FORMAT Component,” on page 1-41) is part of the IOR profile. For RMI/IDL custom value types marshaled within GIOP replies (including the **UnknownExceptionInfo** service context), a format version not greater than the default for the GIOP message level must be sent, except

where the **RMICustomMaxStreamFormat** service context (see Section 1.4.12, “RMICustomMaxStreamFormat Service Context,” on page 1-41) was sent on the associated GIOP request

1.4.11 TAG_RMI_CUSTOM_MAX_STREAM_FORMAT Component

Although the IIOP level of an IOR specifies a default maximum stream format version for RMI/IDL custom value types marshaled as part of GIOP requests to this IOR, there are cases when it may be necessary to override this default.

The **TAG_RMI_CUSTOM_MAX_STREAM_FORMAT** component has an associated value of type octet, encoded as a CDR encapsulation, designating the maximum stream format version for RMI/IDL custom value types that can be used in GIOP messages sent to this IOR.

The **TAG_RMI_CUSTOM_MAX_STREAM_FORMAT** component can appear at most once in any IOR profile. For profiles supporting IIOP 1.2 or greater, it is optionally present. If this component is omitted, then the default maximum stream format version for RMI/IDL custom value types sent to this IOR is 1 for IIOP 1.2 and 2 for IIOP 1.3.

1.4.12 RMICustomMaxStreamFormat Service Context

Although the GIOP level of a request specifies a default maximum stream format version for RMI/IDL custom value types marshaled as part of the associated reply, there are cases when it may be necessary to override this default.

RMICustomMaxStreamFormat identifies a CDR encapsulation of a single octet that specifies the highest RMI/IDL custom stream format version that can be used for RMI/IDL custom valuetypes marshaled within a GIOP reply associated with the GIOP request that carries this service context. If this service context is omitted from a GIOP request, then the default maximum stream format version for RMI/IDL custom value types marshaled within a GIOP reply associated with this request is 1 for GIOP 1.2 and 2 for GIOP 1.3.

1.4.13 Marshaling RMI/IDL Arrays

RMI/IDL arrays must be marshaled with a repository ID indicating their runtime type. Also, RMI/IDL arrays must be unmarshaled according to the type specified in the repository ID.

1.4.14 Creating ORB Instances

The Portability APIs (see Section 1.5, “Portability Interfaces,” on page 1-42) and Application Programming Interfaces (see Section 1.6, “Application Programming Interfaces,” on page 1-62) in the java.rmi.CORBA package define functionality that is not part of the ORB and requires the use of an existing ORB instance for certain operations. Nothing in this specification requires an implementation of these javax.rmi.CORBA APIs to create a new ORB instance.

1.4.15 Runtime Limitations

Our mapping implies three runtime limitations relative to current Java RMI.

Shared reference objects

In Java, remote object references are represented as Java objects. This means that there can be several Java pointers to one object reference. This pointer sharing may be lost when transmitting graphs of Java objects across RMI/IDL.

In practice this is likely to have only very minor impact on Java programmers.

Distributed garbage collection

Java provides automatic garbage collection and RMI using its native protocol extends this to the net with distributed garbage collection.

CORBA does not currently provide support for distributed garbage collection; therefore, distributed garbage collection is not supported as part of RMI/IDL. It is instead each server's responsibility to maintain references to any server objects it wishes to keep active, and to free these references when it wishes the server object to be garbage collected. This is done using the **`exportObject`** and **`unexportObject`** methods of **`javax.rmi.PortableRemoteObject`** (see Section 1.6.1, "PortableRemoteObject," on page 1-62).

Narrowing

Java provides type-checked casts as part of the language. RMI using its native protocol dynamically downloads stubs that accurately reflect the RMI interface types of each remote object reference, thereby allowing Java language casts to be used to narrow remote object references.

Downloadable stubs are not required by the CORBA object model. Since we cannot rely on downloadable stubs, we cannot rely on simple Java casts to implement narrowing of object references. We have therefore defined an explicit **`narrow`** method (see Section 1.4.3, "Narrowing," on page 1-32) that programmers must use when narrowing portable RMI object references.

1.5 Portability Interfaces

This section describes extensions to the portable stubs and skeletons architecture defined in the IDL/Java language mapping. These extensions allow stubs and skeletons to be created for this Java to IDL mapping that can rely on a standard set of Java ORB Portability APIs, including APIs for serializing Java objects to GIOP format.

These ORB portability APIs also allow alternative implementations of the RMI/IDL APIs.

See Section 1.5.2.1, "Stub classes," on page 1-54 and Section 1.5.2.3, "Tie classes," on page 1-57 for simple example stubs and ties.

1.5.1 Portability APIs

1.5.1.1 Tie

The interface **javax.rmi.CORBA.Tie** defines methods that all RMI/IDL server side ties must implement.

The **javax** prefix indicates these classes are part of a standard extension. The use of this prefix allows these interfaces and classes to be delivered as an add-on to existing JDKs. Security checks in the browsers prevent downloading of classes whose top-level package qualifier is **java**, so Sun has defined the convention of using a top-level qualifier of **javax** for extensions.

```
// Java
public interface Tie extends
   org.omg.CORBA.portable.InvokeHandler {

   org.omg.CORBA.Object thisObject();

   void deactivate() throws java.rmi.NoSuchObjectException;

   org.omg.CORBA.ORB orb();

   void orb(org.omg.CORBA.ORB orb);

   void setTarget(java.rmi.Remote target);

   java.rmi.Remote getTarget();
}
```

The **thisObject** method returns an object reference for the target object represented by the **Tie**. It is semantically equivalent to the **_this_object()** method of the **org.omg.PortableServer.Servant** class.

The **deactivate** method deactivates the target object represented by the **Tie**. It is semantically equivalent to the **deactivate_object** method of the **org.omg.PortableServer.POA** class. If the target object could not be deactivated (e.g., because it is not currently active), a **NoSuchObjectException** is thrown.

The **orb()** method returns the ORB for the **Tie**. It is semantically equivalent to the **_orb()** method of the **org.omg.PortableServer.Servant** class.

The **orb(ORB orb)** method sets the ORB for the **Tie**. It is semantically equivalent to calling **ORB.set_delegate()** with an actual parameter of type **org.omg.PortableServer.Servant**.

The **setTarget** method must be implemented by tie classes. It will be called by **Util.registerTarget** to notify the tie of its registered target implementation object.

The **getTarget** method must be implemented by tie classes. It returns the registered target implementation object for the tie.

1.5.1.2 Stub

The class **javax.rmi.CORBA.Stub** is the standard base class from which all RMI/IDL stubs must inherit. Its main reason for existence is to act as a convenience base class to handle stub serialization.

```
// Java
public abstract class Stub
        extends org.omg.CORBA_2_3.portable.ObjectImpl
        implements java.io.Serializable {

   private static final long serialVersionUID =
        1087775603798577179L;

   public int hashCode() { ... }
   public boolean equals(java.lang.Object obj) { ... }
   public String toString() { ... }

   public void connect(org.omg.CORBA.ORB orb)
        throws java.rmi.RemoteException { ... }

   private void writeObject(java.io.ObjectOutputStream s)
        throws java.io.IOException { ... }

   private void readObject(java.io.ObjectInputStream s)
        throws java.io.IOException,
        ClassNotFoundException { ... }
}
```

The **hashCode** method shall return the same hash code for all stubs that represent the same remote object. The **equals** method shall return **true** when used to compare stubs that represent the same remote object, and **false** otherwise. The **toString** method shall return the same string for all stubs that represent the same remote object.

The **connect** method makes the stub ready for remote communication using the specified ORB object **orb**. Connection normally happens implicitly when the stub is received or sent as an argument on a remote method call, but it is sometimes useful to do this by making an explicit call (e.g., following deserialization). If the stub is already connected to **orb** (i.e., has a delegate set for **orb**), then **connect** takes no action. If the stub is connected to some other ORB, then a **RemoteException** is thrown. Otherwise, a delegate is created for this stub and the ORB object **orb**.

The **Stub.connect** method is not intended to be called directly by application code. Instead, application code should call the **PortableRemoteObject.connect** method (see Section 1.6.1, “PortableRemoteObject,” on page 1-62), which will in turn

call the **Stub.connect** method. This allows the application code to remain portable between IIOP and JRMP. RMI/IDL stubs may also be connected to an ORB implicitly by being passed to **OutputStream.write_Object**.

The **writeObject** and **readObject** methods support stub serialization and deserialization by saving and restoring the IOR associated with the stub. The **writeObject** method writes the following data to the serialization stream:

1. int - length of IOR type id
2. byte[] - IOR type ID encoded using ISO 8859-1 (written using a **write** call, not a **writeObject** call)
3. int - number of IOR profiles
4. For each IOR profile:

   a. int - profile tag
   b. int - length of profile data
   c. byte[] - profile data (written using a **write** call, not a **writeObject** call)

1.5.1.3 ValueOutputStream

The interface **org.omg.CORBA.portable.ValueOutputStream** defines methods that allow serialization of custom-marshaled RMI/IDL objects to GIOP streams.

```
// Java
public interface ValueOutputStream {

   void start_value(java.lang.String rep_id);

   void end_value();
}
```

The **start_value** method ends any currently open chunk, writes a valuetype header for a nested custom valuetype (with a null codebase and the specified repository ID), and increments the valuetype nesting depth.

The **end_value** method ends any currently open chunk, writes the end tag for the nested custom valuetype, and decrements the valuetype nesting depth.

1.5.1.4 ValueInputStream

The interface **org.omg.CORBA.portable.ValueInputStream** defines methods that allow deserialization of custom-marshaled RMI/IDL objects froim GIOP streams.

```
// Java
public interface ValueInputStream {

   void start_value();

   void end_value();
```

```
}
```

The **start_value** method reads a valuetype header for a nested custom valuetype and increments the valuetype nesting depth.

The **end_value** method reads the end tag for the nested custom valuetype (after skipping any data that precedes the end tag) and decrements the valuetype nesting depth.

1.5.1.5 ValueHandler and Related Interfaces

The interfaces **javax.rmi.CORBA.ValueHandler**, **javax.rmi.CORBA.ValueHandlerMultiFormat**, and **javax.rmi.CORBA.ValueHandlerCodeBaseDelegate** define methods that allow serialization of Java objects to and from GIOP streams.

```
// Java
public interface ValueHandler {

   void writeValue(org.omg.CORBA.portable.OutputStream out,
                   java.io.Serializable value);

   java.io.Serializable readValue(
         org.omg.CORBA.portable.InputStream in,
         int offset,
         Class clz,
         String repositoryID,
         org.omg.SendingContext.RunTime sender);

   String getRMIRepositoryID(Class clz);

   boolean isCustomMarshaled(Class clz);

   /**
   *@deprecated
   */
   org.omg.SendingContext.RunTime getRunTimeCodeBase();

   java.io.Serializable writeReplace(
            java.io.Serializable value);
}

public interface ValueHandlerMultiFormat
                     extends ValueHandler {

   byte getMaximumStreamFormatVersion();

   void writeValue(org.omg.CORBA.portable.OutputStream out,
                     java.io.Serializable value,
                     byte streamFormatVersion);
}
```

```
public interface ValueHandlerCodeBaseDelegate {
   org.omg.SendingContext.CodeBaseOperations
            getRunTimeCodeBaseDelegate();
}
```

The **writeValue** method can be used to write GIOP data, including RMI remote objects and serialized data objects, to an underlying portable **OutputStream**.

The implementation of the **writeValue** method interacts with the core Java serialization machinery. The data generated during serialization is written using the underlying **OutputStream** object.

The **readValue** method can be used to read GIOP data, including RMI remote objects and serialized data objects, from an underlying portable **InputStream**. The **offset** parameter is the offset in the stream of the value being unmarshaled. The **clz** parameter is the Java class of the value to be unmarshaled. The **repositoryID** parameter is the repository ID unmarshaled from the value header by the caller of **readValue**. The **sender** parameter is the sending context object passed in the optional service context tagged **SendingContextRunTime** in the GIOP header, if any, or null if no sending context was passed.

The implementation of the **readValue** method interacts with the core Java serialization machinery. The data required during deserialization is read using the underlying **InputStream** object.

The **getRMIRepositoryID** method returns the RMI-style repository ID string for **clz**.

The **isCustomMarshaled** method returns **true** if the value is custom marshaled and therefore requires a chunked encoding, and **false** otherwise.

The **getRunTimeCodeBase** method returns the **ValueHandler** object's **SendingContext::RunTime** object reference, which is used to construct the **SendingContextRunTime** service context.

The **writeReplace** method returns the serialization replacement for the **value** object. This is the object returned by calling **value.writeReplace()**, if **value** has a **writeReplace** method.

The **ValueHandlerMultiFormat** interface introduces a method **getMaximumStreamFormatVersion** that returns the maximum stream format version for RMI/IDL custom value types that is supported by this **ValueHandler** object. The **ValueHandler** object must support the returned stream format version and all lower versions. The format versions currently defined are 1 and 2. See Section 1.4.10, “Custom Marshaling Format,” on page 1-40 for more details.

The **ValueHandlerMultiFormat** interface introduces an overloaded **writeValue** method that allows the ORB to pass the required stream format version for RMI/IDL custom value types. If the ORB calls this method, it must pass a stream format version between 1 and the value returned by the **getMaximumStreamFormatVersion**

method inclusive, or else a BAD_PARAM exception with standard minor code 39 must be thrown. If the ORB calls the **ValueHandler.writeValue** method, stream format version 1 is implied.

The **valueHandlerCodeBaseDelegate** interface introduces a method **getRunTimeCodeBaseDelegate**. This method returns an implementation delegate that an ORB can use to create a **SendingContext::RunTime** object reference and a **SendingContextRunTime** service context. This method replaces the **ValueHandler.getRunTimeCodeBase** method, which is deprecated. The **ValueHandler** object returned by the **Util.createValueHandler** method must also implement the **ValueHandlerCodeBaseDelegate** interface.

Execution model for Serialization

Sun will provide an implementation of the **ValueHandler** interface that handles writing and reading RMI/IDL objects by making calls to lower-level CORBA **OutputStream** and **InputStream** objects, which can be provided by an independent ORB vendor. The Sun-provided implementation will handle the interactions with the Java serialization machinery and will write any serialized data through to the lower level stream.

Typically the ORB vendors will implement their own GIOP input and output streams. Before transmitting RMI/IDL data they will create an object that supports the **ValueHandler** interface by calling the **createValueHandler** method of the **javax.rmi.CORBA.Util** class (see Section 1.5.1.6, “Util,” on page 1-51). When they need to marshal a non-IDL value, they will call **ValueHandler.writeValue**, and when they need to unmarshal a non-IDL value, they will call **ValueHandler.readValue**.

The ORB output stream passed to the **ValueHandlerMultiFormat.writeValue** method must implement the **ValueOutputStream** interface (see Section 1.5.1.3, “ValueOutputStream,” on page 1-45), and the ORB input stream passed to the **ValueHandler.readValue** method must implement the **ValueInputStream** interface (see Section 1.5.1.4, “ValueInputStream,” on page 1-45).

Value Marshaling

When marshaling an RMI value, the ORB stream must call **Util.getCodeBase** to get the codebase string, **ValueHandler.getRMIRepositoryID** to get the repository ID string, and **ValueHandler.isCustomMarshaled** to see if the value is custom marshaled and therefore requires a chunked encoding.

The ORB stream writes the value tag, codebase (if any), and repository ID. It calls **ValueHandler.writeValue** to write the state of the value. The ORB stream deals with nulls, indirections, chunking, and end tags.

The ORB casts the **ValueHandler** object to type **ValueHandlerCodeBaseDelegate** and calls its **getRunTimeCodeBaseDelegate** method to obtain an implementation delegate of type **CodeBaseOperations**. The ORB creates a **SendingContextRunTime** service context containing an object reference for a tied implementation whose delegate

is this **CodeBaseOperations** object. Clients must send this service context on the first GIOP request that flows over a connection that may be used to send RMI values to the server. Servers must send this service context on the first GIOP reply that flows over a connection that may be used to send RMI values to the client.

The ORB calls the **writeReplace** method before calling **writeValue**. The result from calling this method is passed to **ValueHandler.writeValue** unless either

- it is a previously marshaled value, in which case it is marshaled as an indirection, or
- its class implements **org.omg.CORBA.Object**, in which case it is marshaled as an object reference.

An ORB stream instance must only call **writeReplace** once for each value that it marshals.

Before calling the **writeValue** method of the **ValueHandler** object, the ORB must determine the stream format version to be used. This is the maximum format version that is supported by both the local **ValueHandler** object and the remote connection endpoint. The maximum local format version is the value returned by the **getMaximumStreamFormatVersion** method of the **ValueHandler** object, or 1 if the **ValueHandler** object doesn't support the **ValueHandlerMultiFormat** interface. The maximum remote format version is 1 for GIOP 1.2 messages and 2 for GIOP 1.3 messages, except where these default values are overridden by either the **TAG_RMI_CUSTOM_MAX_STREAM_FORMAT TaggedComponent** (see Section 1.4.11, “TAG_RMI_CUSTOM_MAX_STREAM_FORMAT Component,” on page 1-41) or the **RMICustomMaxStreamFormat** service context (see Section 1.4.12, “RMICustomMaxStreamFormat Service Context,” on page 1-41). For GIOP 1.2 messages, recognition of these overrides is optional.

If the stream format version computed in this way is 2 or greater, the ORB must call the **ValueHandlerMultiFormat.writeValue** method, passing this value. If the stream format version computed in this way is 1, the ORB may call either the **ValueHandlerMultiFormat.writeValue** method (with stream format 1) or the **ValueHandler.writeValue** method.

If the ORB's call to the **ValueHandler** object's **writeValue** method specified RMI/IDL custom value type stream format version 2, then the **ValueHandler** object must call the **ValueOutputStream.start_value** and **ValueOutputStream.end_value** methods of the ORB stream before and after writing the data specified by item 1d of Section 1.4.10, “Custom Marshaling Format,” on page 1-40. The **rep_id** string passed to the **start_value** method must be **"RMI:org.omg.custom.<class>:<hashcode>:<suid>"** where **<class>** is the fully-qualified name of the class whose **writeObject** method is being invoked and **<hashcode>** and **<suid>** are the class's hashcode and SUID. For format version 2, if the ORB stream passed to the **ValueHandler** object doesn't support the **ValueOutputStream** interface, then a BAD_PARAM exception with standard minor code 40 must be thrown.

Value Unmarshaling

When unmarshaling an RMI value, the ORB stream must read the value tag, codebase (if any), and repository ID. The ORB stream calls **`Util.loadClass`** to load the value's class, passing the Java class name contained in the RMI-style repository ID and the codebase string from the value's GIOP encoding (if present) or the **SendingContextRunTime** service context.

The ORB stream calls **`ValueHandler.readValue`** to read the state of the value, passing the current stream offset, the class returned by **`Util.loadClass`**, the repository ID, and the sender's **SendingContext::RunTime** object reference. The repository ID is needed so that the **`ValueHandler`** object can determine if the class passed in is structurally identical to the class used by the sender to marshal the value. The ORB stream deals with nulls, indirections, chunking, and end tags.

The **`ValueHandler`** object may receive an **`org.omg.CORBA.portable.IndirectionException`** from the ORB stream. The ORB input stream throws this exception when it is called to unmarshal a value encoded as an indirection that is in the process of being unmarshaled. This can occur when the ORB stream calls the **`ValueHandler`** object to unmarshal an RMI value whose state contains a recursive reference to itself. Because the top-level **`ValueHandler.readValue`** call has not yet returned a value, the ORB stream's indirection table contains no entry for an object with the stream offset specified by the indirection tag. This stream offset is returned in the exception's **`offset`** field.

If the **`ValueHandler`** object receives an **`IndirectionException`**, it is responsible for ensuring that the correct Java object reference is assigned to the value field that would have held the result returned by the ORB stream if an **`IndirectionException`** had not occurred. The manner in which this is done (e.g., eager or lazy) is not specified. If the offset in an **`IndirectionException`** does not correspond to any offset previously passed to the **`ValueHandler`** object in a **`ValueHandler.readValue`** method call, the **`ValueHandler.readValue`** method shall throw a MARSHAL exception.

If the RMI/IDL custom data unmarshaled from the input stream was encoded using stream format 2, then the **`ValueHandler`** object must call the **`ValueInputStream.start_value`** and **`ValueInputStream.end_value`** methods of the ORB stream before and after reading the data specified by item 1d of Section 1.4.10, “Custom Marshaling Format,” on page 1-40. For format version 2, if the ORB stream passed to the **`ValueHandler`** object doesn't support the **`ValueInputStream`** interface, then a BAD_PARAM exception with standard minor code 41 must be thrown. If the format version unmarshaled by the **`ValueHandler`** object is greater than the maximum version that it supports, then a MARSHAL exception with standard minor code 7 must be thrown.

When using stream version 2, the ORB input stream must throw a MARSHAL exception with standard minor code *[note to editor: number to be assigned by OMG]* to signal an incompatibility between the custom data on the wire and read operations from the **`ValueHandler`** object until **`end_value`** is called. This can occur when a sender's version of a class does not write custom data, but the receiver's version attempts to perform a read operation.

1.5.1.6 Util

A utility class **javax.rmi.CORBA.Util** provides methods that can be used by stubs to perform common operations.

```
// Java
public class Util {

   public static java.rmi.RemoteException
      mapSystemException(org.omg.CORBA.SystemException ex)
            { ... }

   public static void writeAny(
            org.omg.CORBA.portable.OutputStream out,
            java.lang.Object obj){ ... }

   public static java.lang.Object readAny(
            org.omg.CORBA.portable.InputStream in) { ... }

   public static void writeRemoteObject(
            org.omg.CORBA.portable.OutputStream out,
            java.lang.Object obj) { ... }

   public static void writeAbstractObject(
            org.omg.CORBA.portable.OutputStream out,
            java.lang.Object obj) { ... }

   public static void registerTarget(Tie tie,
            java.rmi.Remote target) { ... }

   public static void unexportObject(java.rmi.Remote target)
            throws java.rmi.NoSuchObjectException
            { ... }

   public static Tie getTie(java.rmi.Remote target) { ... }

   public static ValueHandler createValueHandler() { ... }

   public static java.rmi.RemoteException wrapException(
            Throwable obj) { ... }

   public static java.lang.Object copyObject(
            java.lang.Object obj, org.omg.CORBA.ORB orb)
      throws java.rmi.RemoteException { ... }

   public static java.lang.Object[] copyObjects(
            java.lang.Object[] obj, org.omg.CORBA.ORB orb)
      throws java.rmi.RemoteException { ... }

   public static boolean isLocal(Stub s)
      throws java.rmi.RemoteException { ... }
```

```
    public static String getCodebase(Class clz) {... }

    public static Class loadClass(String className,
                                  String remoteCodebase,
                                  ClassLoader loader)
        throws ClassNotFoundException { ... }
}
```

The **mapSystemException** method maps a CORBA system exception to a **java.rmi.RemoteException** or a **java.lang.RuntimeException**. The mapping is described in Section 1.4.8, “Mapping CORBA System Exceptions to RMI Exceptions,” on page 1-35. If the mapped exception is an instance of **java.rmi.RemoteException** or a subclass, the mapped exception is returned; otherwise, it is thrown.

The **writeAny** method writes the Java object **obj** to the output stream **out** in the form of a GIOP **any**. The contents of the GIOP **any** are determined by applying the Java to IDL mapping rules to the actual runtime type of **obj**. If **obj** is null, then it is written as follows: the **TypeCode** is **tk_abstract_interface**, the repository ID is **“IDL:omg.org/CORBA/AbstractBase:1.0”**, the name string is **“”**, and the **any**'s value is a null abstract interface type (encoded as a **boolean** discriminant of **false** followed by a **long** value of **0x00000000**).

The **readAny** method reads a GIOP **any** from the input stream **in** and unmarshals it as a Java object, which is returned. The following **TypeCode**s are valid for the GIOP **any**: **tk_value**, **tk_value_box**, **tk_objref**, and **tk_abstract_interface**. For each of these types, both null and non-null values are valid. If the **TypeCode** is anything other than these, a MARSHAL exception is thrown.

The **writeRemoteObject** method is a utility method for use by stubs when writing an RMI/IDL object reference to an output stream. If **obj** is a stub object, **writeRemoteObject** simply writes **obj** to **out.write_Object**. However, if **obj** is an exported RMI/IDL implementation object, then **writeRemoteObject** allocates (or reuses) a suitable **Tie** (see Section 1.4.4, “Allocating Ties for Remote Values,” on page 1-33), plugs together the tie with **obj**, and writes the object reference for the tie to **out.write_Object**. This method cannot be used to write a JRMP object reference to an output stream.

The **writeAbstractObject** method is another similar utility method for use by stubs. If **obj** is a value object, or a stub object, **writeAbstractObject** simply writes **obj** to **out.write_abstract_interface**. However, if **obj** is an exported RMI/IDL implementation object, then **writeAbstractObject** allocates (or reuses) a suitable **Tie** (see Section 1.4.4, “Allocating Ties for Remote Values,” on page 1-33), plugs together the tie with **obj**, and writes the object reference for the tie to the **out.write_abstract_interface**. This method cannot be used to write a JRMP object reference to an output stream.

The **registerTarget** method is needed to support **unexportObject**. Because **unexportObject** takes a target implementation object as its parameter, it is necessary for the **Util** class to maintain a table mapping target objects back to their

associated **Tie**s. It is the responsibility of the code that allocates a **Tie** to also call the **registerTarget** method to notify the **Util** class of the target object for a given tie. The **registerTarget** method will call the **Tie.setTarget** method to notify the tie object of its target object.

The **unexportObject** method deactivates an implementation object and removes its associated **Tie** from the table maintained by the **Util** class. If the object is not currently exported or could not be deactivated, a **NoSuchObjectException** is thrown.

The **getTie** method returns the tie object for an implementation object **target**, or null if no tie is registered for the **target** object.

The **createValueHandler** method returns a singleton instance of a class that implements the **ValueHandler** and **ValueHandlerCodeBaseDelegate** interfaces.

The **wrapException** method wraps an exception thrown by an implementation method. It returns the corresponding client-side exception. See Section 1.4.8.1, “Mapping of UnknownExceptionInfo Service Context,” on page 1-36 for details.

The **copyObject** method is used by local stubs to copy an actual parameter, result object, or exception. The **copyObjects** method is used by local stubs to copy any number of actual parameters, preserving sharing across parameters as necessary to support RMI/IDL semantics. The actual parameter **Object[]** array holds the method parameter objects that need to be copied, and the result **Object[]** array holds the copied results.

The **copyObject** and **copyObjects** methods ensure that remote call semantics are observed for local calls. They observe copy semantics for value objects that are equivalent to marshaling, and they handle remote objects correctly. Stubs must either call these methods or generate inline code to provide equivalent semantics.

The **isLocal** method has the same semantics as the **ObjectImpl._is_local** method, except that instead of throwing an **org.omg.CORBA.SystemException**, it throws a **java.rmi.RemoteException** that is the result of passing the **SystemException** to the **mapSystemException** method.

The **getCodebase** method returns the Java codebase for the Class object **clz** as a space-separated list of URLs. See Section 1.4.9.2, “Codebase Selection,” on page 1-37 for details.

The **loadClass** method loads a Java class object for the Java class name **className**, using additional information passed in the **remoteCodebase** and **loader** parameters. See Section 1.4.9.5, “Codebase Usage,” on page 1-38 for details.

1.5.1.7 Additional Portability APIs

The Java Language to IDL Mapping uses the following portability APIs which are also used by the OMG IDL to Java Mapping.

```
org.omg.CORBA.portable.InputStream
org.omg.CORBA.portable.OutputStream
org.omg.CORBA_2_3.portable.InputStream
org.omg.CORBA_2_3.portable.OutputStream
org.omg.CORBA.portable.ObjectImpl
org.omg.CORBA.portable.Delegate
org.omg.CORBA_2_3.portable.ObjectImpl
org.omg.CORBA_2_3.portable.Delegate
org.omg.CORBA.portable.InvokeHandler
org.omg.CORBA.portable.ResponseHandler
org.omg.CORBA.portable.ApplicationException
org.omg.CORBA.portable.RemarshalException
org.omg.CORBA.portable.UnknownException
org.omg.CORBA.portable.IndirectionException
org.omg.CORBA.portable.ServantObject
org.omg.CORBA.portable.ServantObjectExt
```

These APIs are described in the *IDL to Java Language Mapping* document.

1.5.2 Generated classes

There are two kinds of classes generated as part of this specification.

1. Stub classes. These are used by RMI/IDL clients to send calls to a server. A stub class is required for each RMI/IDL remote interface.
2. Tie classes. These are used to process incoming calls and dispatch the calls to a server implementation class. A tie class is required for each RMI/IDL implementation class.

No generated classes are required for RMI/IDL value types, exceptions, etc.

1.5.2.1 Stub classes

For each RMI/IDL remote interface **`Foo`** there will be a stub class **`_Foo_Stub`** that extends **`javax.rmi.CORBA.Stub`** and implements **`Foo`**.

The stub class supports stub methods for all the RMI/IDL remote methods in the RMI/IDL remote interfaces that it implements, and must have a public no-argument constructor.

Here is a simple RMI/IDL interface and an example stub class:

```
// Java
public interface Aardvark extends java.rmi.Remote {
   public int echo(int x) throws java.rmi.RemoteException,
                                  Boomerang;
}

public class _Aardvark_Stub extends javax.rmi.CORBA.Stub
            implements Aardvark {
```

```
   public _Aardvark_Stub() {}  // implicit or explicit

   public int echo(int x) throws java.rmi.RemoteException,
                                 Boomerang {
      org.omg.CORBA_2_3.portable.InputStream in = null;
      try {
         try {
            org.omg.CORBA.OutputStream out =
               _request("echo", true);
            out.write_long(x);
            in = (org.omg.CORBA_2_3.portable.InputStream)
               _invoke(out);
            return in.read_long();
         } catch (org.omg.CORBA.portable.
                     ApplicationException ex) {
            in = (org.omg.CORBA_2_3.portable.InputStream)
               ex.getInputStream();
            String id = in.read_string();
            if (id.equals("IDL:BoomerangEx/1.0")) {
               throw (Boomerang)in.read_value();
            } else {
               throw new java.rmi.UnexpectedException(id);
            }
         } catch (org.omg.CORBA.portable.RemarshalException
                  ex) {
            return echo(x);
         }
      } catch (org.omg.CORBA.SystemException ex) {
         throw javax.rmi.CORBA.Util.mapSystemException(ex);
      } finally {
         _releaseReply(in);
      }
   }
}
```

1.5.2.2 Local Stubs

The stub class may provide an optimized call path for local server implementation objects. For a method **`echo(int x)`** of a remote interface **`Aardvark`**, the optimized path does the following:

1. Find out if the servant is local by calling **`Util.isLocal()`**
2. If the servant is local, call **`this._servant_preinvoke("echo", Aardvark.class)`**
3. If **`_servant_preinvoke`** returned a non-null ServantObject **`so`**, do the following:
   a. Call **`((Aardvark)so.servant).echo(x)`**

b. If the invocation on the servant completed without throwing an exception, and **so** is an instance of **ServantObjectExt**, then call **so.normalCompletion()**
c. If the invocation on the servant threw exception **exc**, and **so** is an instance of **ServantObjectExt**, then call **so.exceptionalCompletion(exc)**
d. Call **this._servant_postinvoke(so)**

4. If **_servant_preinvoke** returned null, repeat step 1. The call to **Util.isLocal()** will return false, causing the non-optimized path to be followed.

The **_servant_preinvoke** method returns non-null if, and only if, an optimized local call may be used. It performs any security checking that may be necessary. If the **_servant_preinvoke** method returns non-null, then the **servant** field of the returned **ServantObject** must contain an object that implements the RMI/IDL remote interface and can be used to call the servant implementation

Local stubs are responsible for performing copying of method parameters, results and exceptions, and handling remote objects correctly in order to provide remote/local-transparent RMI/IDL semantics.

The following is an example of a stub class that provides this optimized call path.

```
// Java
import org.omg.CORBA.portable.ServantObjectExt;

public class _Aardvark_Stub extends javax.rmi.CORBA.Stub
            implements Aardvark {

public int echo(int x) throws java.rmi.RemoteException,
            Boomerang {
   if (!javax.rmi.CORBA.Util.isLocal(this)) {
      // remote call path
      org.omg.CORBA_2_3.portable.InputStream in = null;
      try {
         try {
            org.omg.CORBA.portable.OutputStream out =
               _request("echo", true);
            out.write_long(x);
            in = (org.omg.CORBA_2_3.portable.InputStream)
               _invoke(out);
            return in.read_long();
         } catch (org.omg.CORBA.portable.
                     ApplicationException ex) {
            in = (org.omg.CORBA_2_3.portable.InputStream)
               ex.getInputStream();
            String id = in.read_string();
            if (id.equals("IDL:BoomerangEx/1.0")) {
               throw (Boomerang)in.read_value();
            } else {
               throw new java.rmi.UnexpectedException(id);
            }
         } catch (org.omg.CORBA.portable.RemarshalException
```

```
                    ex) {
                return echo(x);
            }
        } catch (org.omg.CORBA.SystemException ex) {
            throw javax.rmi.CORBA.Util.mapSystemException(ex);
        } finally {
            _releaseReply(in);
        }

    } else {
        // local call path
        org.omg.CORBA.portable.ServantObject so =
            _servant_preinvoke("echo", Aardvark.class);
        if (so == null)
            return echo(x);
        try {
            int result = ((Aardvark)so.servant).echo(x);
            if (so instanceof ServantObjectExt)
                ((ServantObjectExt)so).normalCompletion();
            return result;
        } catch (Throwable ex) {
            if (so instanceof ServantObjectExt)
                ((ServantObjectExt)so).
                    exceptionalCompletion(ex);
            Throwable ex2 = (Throwable)
                javax.rmi.CORBA.Util.copyObject(ex, _orb());
            if (ex2 instanceof Boomerang)
                throw (Boomerang)ex2;
            else
                throw javax.CORBA.Util.wrapException(ex2);
        } finally {
            _servant_postinvoke(so);
        }
    }
}

}
```

1.5.2.3 Tie classes

For each RMI/IDL implementation class there will be a corresponding tie class that implements **`javax.rmi.CORBA.Tie`**. The tie class is called by the ORB to process an incoming call and to pass the call through to an associated target implementation object.

After the **`Tie`** object has been constructed, the target implementation object must be set with a call on **`Util.registerTarget`**.

Here is a simple RMI/IDL interface and an example **`Tie`** class:

```
// Java
public interface Aardvark extends java.rmi.Remote {
   public int echo(int x) throws java.rmi.RemoteException,
                                  Boomerang;
}

public class _Aardvark_Tie
   extends org.omg.PortableServer.Servant
   implements javax.rmi.CORBA.Tie {
   private Aardvark target;

   public void setTarget(java.rmi.Remote targ) {
      target = (Aardvark) targ;
   }

   public java.rmi.Remote getTarget() {
      return target;
   }

   public org.omg.CORBA.OutputStream _invoke(String method,
            org.omg.CORBA.InputStream in,
            org.omg.CORBA.portable.ResponseHandler rh) {
      try {
         if (method.equals("echo")) {
            try {
               int x = in.read_long();
               int result = target.echo(x);
               org.omg.CORBA_2_3.portable.OutputStream out
                = (org.omg.CORBA_2_3.portable.OutputStream)
                  rh.createReply();
               out.write_long(result);
               return out;
            } catch (Boomerang ex) {
               String exid = "IDL:BoomerangEx/1.0";
               org.omg.CORBA_2_3.portable.OutputStream out
                = (org.omg.CORBA_2_3.portable.OutputStream)
                  rh.createExceptionReply();
               out.write_string(exid);
               out.write_value(ex);
               return out;
            }
         } else {
            throw new org.omg.CORBA.BAD_OPERATION();
         }
      } catch (org.omg.CORBA.SystemException ex) {
         throw ex;
      } catch (Throwable ex) {
         throw new
            org.omg.CORBA.portable.UnknownException(ex);
      }
   }
```

```
    public org.omg.CORBA.Object thisObject() { ... }

    public void deactivate() { ... }

    public org.omg.CORBA.ORB orb() { ... }

    public void orb(org.omg.CORBA.ORB orb) { ... }
}
```

1.5.3 Replaceability of API Implementations

A framework is provided to enable vendor-specific implementations of the Java Language to IDL Mapping Portability Interfaces and Application Programming Interfaces. The affected classes are:

```
javax.rmi.CORBA.Stub
javax.rmi.CORBA.Util
javax.rmi.PortableRemoteObject
```

These classes are able to optionally delegate their methods to separate implementation classes, which can be provided by ORB vendors.

1.5.3.1 StubDelegate

The implementation delegate class for **`javax.rmi.CORBA.Stub`** must implement the following interface for per-instance delegation:

```
package javax.rmi.CORBA;

public interface StubDelegate {

    int hashCode(Stub self);

    boolean equals(Stub self, java.lang.Object obj);

    String toString(Stub self);

    void connect(Stub self, org.omg.CORBA.ORB orb)
          throws java.rmi.RemoteException;

    void writeObject(Stub self, java.io.ObjectOutputStream s)
          throws java.io.IOException;

    void readObject(Stub self, java.io.ObjectInputStream s)
          throws java.io.IOException,
                ClassNotFoundException;
}
```

The above methods are called by the corresponding methods of **javax.rmi.CORBA.Stub** when delegation has been specified as described in Section 1.5.3.4, “Delegation Mechanism,” on page 1-61.

1.5.3.2 UtilDelegate

The implementation delegate class for **javax.rmi.CORBA.Util** must implement the following interface for per-class delegation:

```
package javax.rmi.CORBA;

public interface UtilDelegate {

   java.rmi.RemoteException mapSystemException(
               org.omg.CORBA.SystemException ex);

   void writeAny(org.omg.CORBA.portable.OutputStream out,
               java.lang.Object obj);

   java.lang.Object readAny(
               org.omg.CORBA.portable.InputStream in);

   void writeRemoteObject(
               org.omg.CORBA.portable.OutputStream out,
               java.lang.Object obj);

   void writeAbstractObject(
               org.omg.CORBA.portable.OutputStream out,
               java.lang.Object obj);

   void registerTarget(Tie tie, java.rmi.Remote target);

   void unexportObject(java.rmi.Remote target)
               throws java.rmi.NoSuchObjectException;

   Tie getTie(java.rmi.Remote target);

   ValueHandler createValueHandler();

   String getCodebase(Class clz);

   Class loadClass(String className, String remoteCodebase,
                  ClassLoader loader)
         throws ClassNotFoundException;

   boolean isLocal(Stub stub)
         throws java.rmi.RemoteException;

   java.rmi.RemoteException wrapException(Throwable obj);
```

```
    java.lang.Object copyObject(java.lang.Object obj,
                   org.omg.CORBA.ORB orb)
          throws java.rmi.RemoteException;

    java.lang.Object[] copyObjects(java.lang.Object[] obj,
                   org.omg.CORBA.ORB orb)
          throws java.rmi.RemoteException;
}
```

The above methods are called by the corresponding methods of **`javax.rmi.CORBA.Util`** when delegation has been specified as described in Section 1.5.3.4, “Delegation Mechanism,” on page 1-61.

1.5.3.3 PortableRemoteObjectDelegate

The implementation delegate class for **`javax.rmi.PortableRemoteObject`** must implement the following interface for per-class delegation:

```
package javax.rmi.CORBA;

public interface PortableRemoteObjectDelegate {

    void exportObject(java.rmi.Remote obj)
          throws java.rmi.RemoteException;

    java.rmi.Remote toStub (java.rmi.Remote obj)
          throws NoSuchObjectException;

    void unexportObject(java.rmi.Remote obj)
          throws NoSuchObjectException;

    java.lang.Object narrow (java.lang.Object narrowFrom,
                             Class narrowTo)
          throws ClassCastException;

    void connect (java.rmi.Remote target,
                  java.rmi.Remote source)
          throws java.rmi.RemoteException;
}
```

The above methods are called by the corresponding methods of **`javax.rmi.PortableRemoteObject`** when delegation has been specified as described in Section 1.5.3.4, “Delegation Mechanism,” on page 1-61.

1.5.3.4 Delegation Mechanism

Alternate implementations of the standard API classes are enabled by setting system properties or placing entries in the orb.properties file. The names of the new system properties are:

javax.rmi.CORBA.StubClass
javax.rmi.CORBA.UtilClass
javax.rmi.CORBA.PortableRemoteObjectClass

For security reasons, each replaceable API class reads its implementation delegate class system property at static initialization time and uses this information to set up implementation delegation if this has been specified. The delegation arrangement thus established cannot be changed subsequently. The search order for implementation delegate class names is:

1. The system properties
2. The orb.properties file

For each implementation delegate class, an instance is created using the **Class.newInstance()** method. For the **Util** and **PortableRemoteObject** delegate classes, this is a singleton instance. For the **Stub** delegate class, there is one delegate instance per stub object. The methods in the standard API classes test if a delegate instance exists and if so, forward the method call on to the delegate instance.

1.6 Application Programming Interfaces

One new API class is introduced to support RMI/IDL implementations.

1.6.1 PortableRemoteObject

The **javax.rmi.PortableRemoteObject** class is intended to act as a base class for RMI/IDL server implementation classes (see Section 1.2.3.1, “Stubs and remote implementation classes,” on page 1-4).

```
// Java
public class PortableRemoteObject {

   protected PortableRemoteObject()
      throws java.rmi.RemoteException { ... }

   public static void exportObject(java.rmi.Remote obj)
      throws java.rmi.RemoteException { ... }

   public static java.rmi.Remote toStub(java.rmi.Remote obj)
      throws java.rmi.NoSuchObjectException { ... }

   public static void unexportObject(java.rmi.Remote obj)
      throws java.rmi.NoSuchObjectException { ... }

   public static java.lang.Object narrow(
            java.lang.Object obj, Class newClass)
      throws ClassCastException { ... }
```

```
   public static void connect(
            java.rmi.Remote target, java.rmi.Remote source)
      throws java.rmi.RemoteException { ... }
}
```

The protected constructor is called by the derived implementation class to initialize the base class state.

Server side implementation objects may either inherit from **javax.rmi.PortableRemoteObject** or they may simply implement an RMI/IDL remote interface and then use the **exportObject** method to register themselves as a server object.

A call to **exportObject** with no objects exported creates a non-daemon thread that keeps the Java virtual machine alive until all exported objects have been unexported by calling **unexportObject**.

It is up to the implementation to decide when to actually export (i.e., connect) remote objects. It may be done in the **PortableRemoteObject** constructor (for objects that subclass **PortableRemoteObject**) or in the **exportObject** method, or it may be deferred until the remote object is actually written to an **OutputStream**.

It is an error to call **exportObject** on an object that is already exported.

The **toStub** method takes a server implementation object and returns a stub object that can be used to access that server object. The argument object must currently be exported, either because it is a subclass of **PortableRemoteObject** or by virtue of a previous call to **PortableRemoteObject.exportObject**. If the object is not currently exported, a NoSuchObjectException is thrown. The returned stub implements the same RMI/IDL remote interfaces as the implementation object. If an RMI/IDL Tie class is available for the given object, the **toStub** method will return an IIOP stub; otherwise, it will return a JRMP stub. The **toStub** method may be passed a stub, in which case it simply returns this stub.

The stub returned by **toStub** has the same connection status as the target implementation object passed to **toStub**. So if the target object is connected, the returned stub is connected to the same ORB. If the target object is unconnected, the returned stub is unconnected.

The **unexportObject** method is used to deregister a currently exported server object from the ORB runtimes, allowing the object to become available for garbage collection. If the object is not currently exported, a NoSuchObjectException is thrown. This is implemented by calling through to **Util.unexportObject**.

The **narrow** method takes an object reference or an object of an RMI/IDL abstract interface type and attempts to narrow it to conform to the given **newClass** RMI/IDL type. If the operation is successful, the result will be an object of type **newClass**; otherwise, an exception will be thrown. If **obj** is null, then **narrow** returns null.

The **connect** method makes the remote object **target** ready for remote communication using the same communications runtime[3] as **source**. Connection normally happens implicitly when the object is sent or received as an argument on a remote method call, but it is sometimes useful to do this by making an explicit call.

The **target** object may be either an RMI/IDL stub or an exported RMI/IDL implementation object, and the **source** object may also be either an RMI/IDL stub or an exported RMI/IDL implementation object.

If **target** is already connected to the same communications runtime as **source**, then **connect** takes no action. Otherwise, **target** must be an unconnected object (i.e., an RMI/IDL CORBA stub without a delegate or an implementation object whose RMI/IDL tie has not been associated with an ORB), and **source** must be a connected object (i.e., an RMI/IDL CORBA stub with a delegate or an implementation object with an RMI/IDL tie that has been associated with an ORB), or else a RemoteException is thrown. The **target** object is connected to the same ORB as **source** by calling the **Stub.connect** method if it is a stub (see Section 1.5.1.2, "Stub," on page 1-44) or by associating its tie with an ORB if it is an implementation object.

RMI/IDL implementation objects may be connected implicitly by being passed to **Util.writeRemoteObject** or **Util.writeAbstractObject**. RMI/IDL stubs may be connected implicitly by being passed to **OutputStream.write_Object**. Connecting an implementation object is not the same as exporting it, and RMI/IDL implementation objects may be unconnected when first exported. RMI/IDL implementation objects are implicitly connected when they are exported to JRMP, and RMI-JRMP stubs are implicitly connected when they are created.

1.7 Generated IDL File Structure

This section is not part of the formal specification of the Java Language to OMG IDL Mapping, but it contains some suggestions for generated file structure.

Tool vendors may choose to map each RMI/IDL interface, value type, or exception type to a separate .idl file. This follows the normal Java style and may be easier for Java RMI/IDL programmers to maintain than requiring that (say) all OMG IDL definitions be put into a single OMG IDL file.

This approach does raise some issues for the generated OMG IDL, which are briefly worth mentioning.

First, the use of separate .idl files requires the use of "reopenable" modules, so that separate files can have separate free-standing module definitions.

Second, although OMG IDL permits forward references to OMG IDL interfaces, it does not support forward references to structs or exceptions, and there are some limits on the use of interface references. Any forward references to interfaces must be satisfied by later definitions of those interfaces.

---

3. For IIOP, the communications runtime is an ORB; for JRMP, it is the JRMP transport subsystem.

One possible way of dealing with these difficulties is to use an OMG IDL file layout similar to the following:

1. The entire OMG IDL definition is bracketed in standard C pre-processor boilerplate used to guarantee it is only included once:

```
#ifndef __foo__
#define __foo__
    ...
#endif
```

2. An OMG IDL forward reference is generated for each OMG IDL interface that is referenced. (This may require entering and exiting the appropriate target module.)

3. An OMG IDL forward reference is generated for each OMG IDL value type that is referenced. (This may require entering and exiting the appropriate target module.)

4. Each exception referenced in the OMG IDL is **#include**d, in arbitrary order.

5. If the generated OMG IDL is an interface, then **#include** any inherited interfaces.

6. If the generated OMG IDL is a value type, then **#include** any inherited value types.

7. If there are any references to the OMG IDL types **::java::rmi::Remote**, **java::io::Serializable**, **::java::io::Externalizable**, or **::java::lang::_Object**, then generate the following bracketed definitions as required.

```
#ifndef __java_rmi_Remote__
#define __java_rmi_Remote__
module java {
module rmi {
    typedef Object Remote;
};
};
#endif

#ifndef __java_io_Serializable__
#define __java_io_Serializable__
module java {
module io {
    typedef any Serializable;
};
};
#endif

#ifndef __java_io_Externalizable__
#define __java_io_Externalizable__
module java {
module io {
    typedef any Externalizable;
};
```

```
};
#endif

#ifndef __java_lang_Object__
#define __java_lang_Object__
module java {
module lang {
    typedef any _Object;
};
};
#endif
```

This allows different OMG IDL files in the same module to independently define any necessary typedefs.

8. For each OMG IDL sequence type that is referenced, generate a bracketed value definition similar to the following.

```
#ifndef __org_omg_boxedRMI_fred_seq1_Stuff__
#define __org_omg_boxedRMI_fred_seq1_Stuff__
module org {
module omg {
module boxedRMI {
module fred {
    valuetype seq1_Stuff sequence<::fred::Stuff>;
#pragma ID seq1_Stuff
    "RMI:[Lfred.Stuff;:0123456789012345:9876543210987654"
};
};
};
};
#endif
```

This allows different OMG IDL files to independently define any necessary sequence valuetypes.

9. Generate the target OMG IDL in the appropriate module.
10. **#include** any interfaces to which forward references have been declared.
11. **#include** any value types to which forward references have been declared.

Below is an example of how a chunk of RMI/IDL code would be mapped to OMG IDL using this approach.

1.7.1 The Java Definition

Here's a sample RMI/IDL interface, where the referenced type **`fred.Stuff`** is an RMI/IDL value type, **`fred.Test1`** and **`fred.Test2`** are RMI/IDL remote interface types, and **`fred.OurException`** is an RMI/IDL exception type.

```
// Java
package fred;

import java.rmi.*;

public interface Test extends Test1 {
    void noop() throws RemoteException;

    String echo(String arg) throws RemoteException;

    Stuff echoStuff(Stuff p) throws RemoteException;

    Test echoTest(Test t) throws RemoteException;

    int[] echoInts(int args[]) throws RemoteException;

    Stuff[] echoStuffs(Stuff args[]) throws RemoteException;

    void manyArgs(char a, byte b, short c, int d,
          long e,float f, double g) throws RemoteException;

    Test2 fetchTest2(Object x) throws RemoteException;

    void throwAnException() throws RemoteException,
                    OurException;
}
```

1.7.2 The Generated OMG IDL Definition

```
// IDL
#ifndef __fred_Test__
#define __fred_Test__

#include “orb.idl”

module fred {
    interface Test2;
    valuetype Stuff;
};

#include “fred/OurEx.idl”
#include “fred/Test1.idl”

#ifndef __java_lang_Object__
#define __java_lang_Object__
module java {
module lang {
    typedef any _Object;
};
};
```

```
#endif

#ifndef __org_omg_boxedRMI_seq1_long__
#define __org_omg_boxedRMI_seq1_long__
module org {
module omg {
module boxedRMI {
    valuetype seq1_long sequence<long>;
#pragma ID seq1_long "RMI:[I:0000000000000000"
};
};
};
#endif

#ifndef __org_omg_boxedRMI_fred_seq1_Stuff__
#define __org_omg_boxedRMI_fred_seq1_Stuff__
module org {
module omg {
module boxedRMI {
module fred {
    valuetype seq1_Stuff sequence<::fred::Stuff>;
#pragma ID seq1_Stuff
    "RMI:[Lfred.Stuff;:0123456789012345:9876543210987654"
};
};
};
};
#endif

module fred {
    interface Test: Test1 {
        void noop();

        ::CORBA::WStringValue echo(in ::CORBA::WStringValue arg0);

        ::fred::Stuff echoStuff(in ::fred::Stuff arg0);

        ::fred::Test echoTest(in ::fred::Test arg0);

        ::org::omg::boxedRMI::seq1_long echoInts(
                in ::org::omg::boxedRMI::seq1_long arg0);

        ::org::omg::boxedRMI::fred::seq1_Stuff echoStuffs(
                in ::org::omg::boxedRMI::fred::seq1_Stuff arg0);

        void manyArgs(
                in wchar arg0,
                in octet arg1,
                in short arg2,
                in long arg3,
                in long long arg4,
```

```
                in float arg5,
                in double arg6);

        ::fred::Test2 fetchTest2(::java::lang::_Object);

        void throwAnException()  raises (::fred::OurEx);
    };
#pragma ID Test "RMI:fred.Test:0000000000000000"
};

#include "fred/Test2.idl"
#include "fred/Stuff.idl"

#endif
```