The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for
errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner. RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013 OMGÆ and Object Management are registered trademarks of the Object Management Group, Inc. Object Request Broker, OMG IDL, ORB, CORBA, CORBAfacilities, CORBAservices, and COSS are trademarks of the Object Management Group, Inc. X/Open is a trademark of X/Open Company Ltd.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at http://www.omg.org/library/issuerpt.htm.
## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>iii</td>
</tr>
<tr>
<td>0.1 About CORBA Language Mapping Specifications</td>
<td>iii</td>
</tr>
<tr>
<td>0.1.1 Alignment with CORBA</td>
<td>iii</td>
</tr>
<tr>
<td>0.2 Definition of CORBA Compliance</td>
<td>iv</td>
</tr>
<tr>
<td>0.3 Acknowledgements</td>
<td>iv</td>
</tr>
<tr>
<td>0.4 References</td>
<td>v</td>
</tr>
<tr>
<td>1. Smalltalk Language Mapping</td>
<td>1-1</td>
</tr>
<tr>
<td>1.1 Mapping Summary</td>
<td>1-3</td>
</tr>
<tr>
<td>1.2 Key Design Decisions</td>
<td>1-4</td>
</tr>
<tr>
<td>1.2.1 Consistency of Style, Flexibility and Portability of Implementation</td>
<td>1-5</td>
</tr>
<tr>
<td>1.3 Implementation Constraints</td>
<td>1-5</td>
</tr>
<tr>
<td>1.3.1 Avoiding Name Space Collisions</td>
<td>1-5</td>
</tr>
<tr>
<td>1.3.2 Limitations on OMG IDL Types</td>
<td>1-6</td>
</tr>
<tr>
<td>1.4 Smalltalk Implementation Requirements</td>
<td>1-6</td>
</tr>
<tr>
<td>1.5 Conversion of Names to Smalltalk Identifiers</td>
<td>1-7</td>
</tr>
<tr>
<td>1.6 Mapping for Interfaces</td>
<td>1-8</td>
</tr>
<tr>
<td>1.7 Memory Usage</td>
<td>1-8</td>
</tr>
<tr>
<td>1.8 Mapping for Objects</td>
<td>1-8</td>
</tr>
<tr>
<td>1.9 Invocation of Operations</td>
<td>1-8</td>
</tr>
<tr>
<td>1.10 Mapping for Attributes</td>
<td>1-9</td>
</tr>
<tr>
<td>1.10.1 Mapping for Constants</td>
<td>1-10</td>
</tr>
<tr>
<td>1.11 Mapping for Basic Data Types</td>
<td>1-10</td>
</tr>
<tr>
<td>1.12 Mapping for the Any Type</td>
<td>1-12</td>
</tr>
<tr>
<td>1.13 Mapping for Enums</td>
<td>1-12</td>
</tr>
<tr>
<td>1.14 Mapping for Struct Types</td>
<td>1-13</td>
</tr>
<tr>
<td>1.15 Mapping for Fixed Types</td>
<td>1-14</td>
</tr>
<tr>
<td>1.16 Mapping for Union Types</td>
<td>1-14</td>
</tr>
<tr>
<td>1.16.1 Implicit Binding</td>
<td>1-14</td>
</tr>
<tr>
<td>1.16.2 Explicit Binding</td>
<td>1-15</td>
</tr>
<tr>
<td>1.17 Mapping for Sequence Types</td>
<td>1-15</td>
</tr>
<tr>
<td>1.18 Mapping for String Types</td>
<td>1-15</td>
</tr>
<tr>
<td>1.19 Mapping for Wide String Types</td>
<td>1-15</td>
</tr>
<tr>
<td>1.20 Mapping for Array Types</td>
<td>1-15</td>
</tr>
<tr>
<td>1.21 Mapping for Exception Types</td>
<td>1-15</td>
</tr>
<tr>
<td>1.22 Mapping for Operations</td>
<td>1-16</td>
</tr>
<tr>
<td>1.23 Implicit Arguments to Operations</td>
<td>1-16</td>
</tr>
</tbody>
</table>
Contents

1.24 Argument Passing Considerations ................. 1-17
1.25 Handling Exceptions ............................. 1-17
1.26 Exception Values .................................. 1-18
   1.26.1 The CORBAExceptionValue Protocol ...... 1-19
   1.26.2 Pseudo-Objects Mapping Overview ....... 1-19
1.27 CORBA::Request .................................... 1-19
1.28 CORBA::Context .................................... 1-20
1.29 CORBA::Object ..................................... 1-21
1.30 CORBA::ORB ........................................ 1-22
1.31 CORBA::NamedValue ............................... 1-23
1.32 CORBA::NVList .................................... 1-24
Preface

0.1 About CORBA Language Mapping Specifications

The CORBA Language Mapping specifications contain language mapping information for the following languages:

- Ada
- C
- C++
- COBOL
- IDL to Java
- Java to IDL
- Smalltalk

Each language is described in a separate stand-alone volume.

0.1.1 Alignment with CORBA

The following table lists each language mapping and the version of CORBA that this language mapping is aligned with.

<table>
<thead>
<tr>
<th>Language Mapping</th>
<th>Aligned with CORBA version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>CORBA 2.0</td>
</tr>
<tr>
<td>C</td>
<td>CORBA 2.1</td>
</tr>
<tr>
<td>C++</td>
<td>CORBA 2.3</td>
</tr>
<tr>
<td>COBOL</td>
<td>CORBA 2.1</td>
</tr>
</tbody>
</table>
0.2 Definition of CORBA Compliance

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

Interoperability and Interworking are separate compliance points. For detailed information about Interworking compliance, refer to the Common Object Request Broker: Architecture and Specification, Interworking Architecture chapter.

As described in the OMA Guide, the OMG’s Core Object Model consists of a core and components. Likewise, the body of CORBA specifications is divided into core and component-like specifications. The structure of this manual reflects that division.

The CORBA specifications are divided into these volumes:

1. The Common Object Request Broker: Architecture and Specification, which includes the following chapters:
   - CORBA Core, as specified in Chapters 1-11
   - CORBA Interoperability, as specified in Chapters 12-16
   - CORBA Interworking, as specified in Chapters 17-21

2. The Language Mapping Specifications, which are organized into the following stand-alone volumes:
   - Mapping of OMG IDL to the Ada programming language
   - Mapping of OMG IDL to the C programming language
   - Mapping of OMG IDL to the C++ programming language
   - Mapping of OMG IDL to the COBOL programming language
   - Mapping of OMG IDL to the Java programming language
   - Mapping of Java programming language to OMG/IDL
   - Mapping of OMG IDL to the Smalltalk programming language

0.3 Acknowledgements

The following companies submitted parts of the specifications that were approved by the Object Management Group to become CORBA (including the Language Mapping specifications):

<table>
<thead>
<tr>
<th>Language Mapping</th>
<th>Aligned with CORBA version</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL to Java</td>
<td>CORBA 2.3</td>
</tr>
<tr>
<td>Java to IDL</td>
<td>CORBA 2.3</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>CORBA 2.0</td>
</tr>
</tbody>
</table>
In addition to the preceding contributors, the OMG would like to acknowledge Mark Linton at Silicon Graphics and Doug Lea at the State University of New York at Oswego for their work on the C++ mapping specification.

0.4 References

The following list of references applies to CORBA and/or the Language Mapping specifications:

IDL Type Extensions RFP, March 1995. OMG TC Document 95-1-35.


Note – The Smalltalk Language Mapping specification is aligned with CORBA version 2.0.

Contents

This chapter contains the following sections.

<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smalltalk Overview</td>
<td></td>
</tr>
<tr>
<td>“Mapping Summary”</td>
<td>1-3</td>
</tr>
<tr>
<td>“Key Design Decisions”</td>
<td>1-4</td>
</tr>
<tr>
<td>Mapping of OMG IDL to Smalltalk</td>
<td></td>
</tr>
<tr>
<td>“Implementation Constraints”</td>
<td>1-5</td>
</tr>
<tr>
<td>“Smalltalk Implementation Requirements”</td>
<td>1-6</td>
</tr>
<tr>
<td>“Conversion of Names to Smalltalk Identifiers”</td>
<td>1-7</td>
</tr>
<tr>
<td>“Mapping for Interfaces”</td>
<td>1-8</td>
</tr>
<tr>
<td>“Memory Usage”</td>
<td>1-8</td>
</tr>
<tr>
<td>“Mapping for Objects”</td>
<td>1-8</td>
</tr>
<tr>
<td>“Invocation of Operations”</td>
<td>1-8</td>
</tr>
<tr>
<td>“Mapping for Attributes”</td>
<td>1-9</td>
</tr>
<tr>
<td>“Mapping for Basic Data Types”</td>
<td>1-10</td>
</tr>
<tr>
<td>“Mapping for the Any Type”</td>
<td>1-12</td>
</tr>
<tr>
<td>Section Title</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>“Mapping for Enums”</td>
<td>1-12</td>
</tr>
<tr>
<td>“Mapping for Struct Types”</td>
<td>1-13</td>
</tr>
<tr>
<td>“Mapping for Fixed Types”</td>
<td>1-14</td>
</tr>
<tr>
<td>“Mapping for Union Types”</td>
<td>1-14</td>
</tr>
<tr>
<td>“Mapping for Sequence Types”</td>
<td>1-15</td>
</tr>
<tr>
<td>“Mapping for String Types”</td>
<td>1-15</td>
</tr>
<tr>
<td>“Mapping for Wide String Types”</td>
<td>1-15</td>
</tr>
<tr>
<td>“Mapping for Array Types”</td>
<td>1-15</td>
</tr>
<tr>
<td>“Mapping for Exception Types”</td>
<td>1-15</td>
</tr>
<tr>
<td>“Mapping for Operations”</td>
<td>1-16</td>
</tr>
<tr>
<td>“Implicit Arguments to Operations”</td>
<td>1-16</td>
</tr>
<tr>
<td>“Argument Passing Considerations”</td>
<td>1-17</td>
</tr>
<tr>
<td>“Handling Exceptions”</td>
<td>1-17</td>
</tr>
<tr>
<td>“Exception Values”</td>
<td>1-18</td>
</tr>
<tr>
<td><strong>Mapping of Pseudo Objects to Smalltalk</strong></td>
<td></td>
</tr>
<tr>
<td>“CORBA::Request”</td>
<td>1-19</td>
</tr>
<tr>
<td>“CORBA::Context”</td>
<td>1-20</td>
</tr>
<tr>
<td>“CORBA::Object”</td>
<td>1-21</td>
</tr>
<tr>
<td>“CORBA::ORB”</td>
<td>1-22</td>
</tr>
<tr>
<td>“CORBA::NamedValue”</td>
<td>1-23</td>
</tr>
<tr>
<td>“CORBA::NVList”</td>
<td>1-24</td>
</tr>
<tr>
<td>Appendix A, “Glossary Terms”</td>
<td>1-25</td>
</tr>
</tbody>
</table>
1.1 Mapping Summary

Table 1-1 provides a brief description of the mapping of OMG IDL constructs to the Smalltalk language, and where in this chapter they are discussed.

<table>
<thead>
<tr>
<th>OMG IDL Construct</th>
<th>Smalltalk Mapping</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>Set of messages that Smalltalk objects which represent object references must respond to. The set of messages corresponds to the attributes and operations defined in the interface and inherited interfaces.</td>
<td>“Mapping for Interfaces” on page 1-8</td>
</tr>
<tr>
<td>Object Reference</td>
<td>Smalltalk object that represents a CORBA object. The Smalltalk object must respond to all messages defined by a CORBA object’s interface.</td>
<td>“Mapping for Objects” on page 1-8</td>
</tr>
<tr>
<td>Operation</td>
<td>Smalltalk message.</td>
<td>“Mapping for Operations” on page 1-16</td>
</tr>
<tr>
<td>Attribute</td>
<td>Smalltalk message</td>
<td>“Mapping for Attributes” on page 1-9</td>
</tr>
<tr>
<td>Constant</td>
<td>Smalltalk objects available in the CORBAConstants dictionary.</td>
<td>“Mapping for Constants” on page 1-10</td>
</tr>
<tr>
<td>Integral Type</td>
<td>Smalltalk objects that conform to the Integer class.</td>
<td>“Mapping for Basic Data Types” on page 1-10</td>
</tr>
<tr>
<td>Floating Point Type</td>
<td>Smalltalk objects which conform to the Float class.</td>
<td>“Mapping for Basic Data Types” on page 1-10</td>
</tr>
<tr>
<td>Boolean Type</td>
<td>Smalltalk true or false objects.</td>
<td>“Mapping for Basic Data Types” on page 1-10</td>
</tr>
<tr>
<td>Enumeration Type</td>
<td>Smalltalk objects which conform to the CORBAEnum protocol.</td>
<td>“Mapping for Enums” on page 1-12</td>
</tr>
<tr>
<td>Any Type</td>
<td>Smalltalk objects that can be mapped into an OMG IDL type.</td>
<td>“Mapping for the Any Type” on page 1-12</td>
</tr>
<tr>
<td>Structure Type</td>
<td>Smalltalk object that conforms to the Dictionary class.</td>
<td>“Mapping for Struct Types” on page 1-13</td>
</tr>
<tr>
<td>Fixed Type</td>
<td></td>
<td>“Mapping for Fixed Types” on page 1-14</td>
</tr>
<tr>
<td>Union Type</td>
<td>Smalltalk object that maps to the possible value types of the OMG IDL union or that conform to the CORBAUnion protocol.</td>
<td>“Mapping for Union Types” on page 1-14</td>
</tr>
</tbody>
</table>
1.2 Key Design Decisions

The mapping of OMG IDL to the Smalltalk programming language was designed with the following goals in mind:

- The Smalltalk mapping does not prescribe a specific implementation. Smalltalk class names are specified, as needed, since client code will need the class name when generating instances of datatypes. A minimum set of messages that classes must support is listed for classes that are not documented in the Smalltalk Common Base. The inheritance structure of classes is never specified.
- Whenever possible, OMG IDL types are mapped directly to existing, portable Smalltalk classes.
- The Smalltalk constructs defined in this mapping rely primarily upon classes and methods described in the Smalltalk Common Base document.
- The Smalltalk mapping only describes the public (client) interface to Smalltalk classes and objects supporting IDL. Individual IDL compilers or CORBA implementations might define additional private interfaces.
- The implementation of OMG IDL interfaces is left unspecified. Implementations may choose to map each OMG IDL interface to a separate Smalltalk class; provide one Smalltalk class to map all OMG IDL interfaces; or allow arbitrary Smalltalk classes to map OMG IDL interfaces.
- Because of the dynamic nature of Smalltalk, the mapping of the any and union types is such that an explicit mapping is unnecessary. Instead, the value of the any and union types can be passed directly. In the case of the any type, the Smalltalk mapping will derive a TipoCode which can be used to represent the value. In the case of the union type, the Smalltalk mapping will derive a discriminator which can be used to represent the value.
- The explicit passing of environment and context values on operations is not required.
• Except in the case of object references, no memory management is required for data parameters and return results from operations. All such Smalltalk objects reside within Smalltalk memory, so garbage collection will reclaim their storage when they are no longer used.
• The proposed language mapping has been designed with the following vendor’s Smalltalk implementations in mind: VisualWorks, Smalltalk/V, and VisualAge.

### 1.2 Consistency of Style, Flexibility and Portability of Implementation

To ensure flexibility and portability of implementations, and to provide a consistent style of language mapping, the Smalltalk chapters use the programming style and naming conventions as described in the following documents:


(Throughout the Smalltalk chapter, *Smalltalk Portability: A Common Base* is referred to as *Smalltalk Common Base*.)

The items listed below are the same for all Smalltalk classes used in the Smalltalk mapping:

- If the class is described in the Smalltalk Common Base document, the class must conform to the behavior specified in the document. If the class is not described in the Smalltalk Common Base document, the minimum set of class and instance methods that must be available is described for the class.
- All data types (except object references) are stored completely within Smalltalk memory, so no explicit memory management is required.

The mapping is consistent with the common use of Smalltalk. For example, `sequence` is mapped to instances of `OrderedCollection`, instead of creating a Smalltalk class for the mapping.

### 1.3 Implementation Constraints

This section describes how to avoid potential problems with an OMG IDL-to-Smalltalk implementation.

#### 1.3.1 Avoiding Name Space Collisions

There is one aspect of the language mapping that can cause an OMG IDL compiler to map to incorrect Smalltalk code and cause name space collisions. Because Smalltalk implementations generally only support a global name space, and disallow underscore characters in identifiers, the mapping of identifiers used in OMG IDL to Smalltalk identifiers can result in a name collision. See Section 1.5, “Conversion of Names to Smalltalk Identifiers,” on page 1-7 for a description of the name conversion rules.

As an example of a name collision, consider the following OMG IDL declaration:
interface Example {
    void sample_op () ;
    void sampleOp () ;
};

Both of these operations map to the Smalltalk selector `sampleOp`. To prevent name collision problems, each implementation must support an explicit naming mechanism, which can be used to map an OMG IDL identifier into an arbitrary Smalltalk identifier. For example, `#pragma` directives could be used as the mechanism.

### 1.3.2 Limitations on OMG IDL Types

This language mapping places limitations on the use of certain types defined in OMG IDL.

For the `any` and `union` types, specific integral and floating point types may not be able to be specified as values. The implementation will map such values into an appropriate type, but if the value can be represented by multiple types, the one actually used cannot be determined.\(^1\) For example, consider the `union` definition below.

```plaintext
union Foo switch (long) {
    case 1: long x;
    case 2: short y;
}
```

When a Smalltalk object corresponding to this union type has a value that fits in both a `long` and a `short`, the Smalltalk mapping can derive a discriminator 1 or 2, and map the integral value into either a `long` or `short` value (corresponding to the value of the discriminator determined).

### 1.4 Smalltalk Implementation Requirements

This mapping places requirements on the implementation of Smalltalk that is being used to support the mapping. These are:

- An integral class, conforming to the `Integer` class definition in the Smalltalk Common Base.
- A floating point class, conforming to the `Float` class definition in the Smalltalk Common Base.
- A class named `Character` conforming to the `Character` class definition in the Smalltalk Common Base.
- A class named `Array` conforming to the `Array` class definition in the Smalltalk Common Base.

\(^1\) To avoid this limitation for union types, the mapping allows programmers to specify an explicit binding to retain the value of the discriminator. See Section 1.16, “Mapping for Union Types,” on page 1-14 for a complete description.
• A class named **OrderedCollection** conforming to the
  **OrderedCollection** class definition in the Smalltalk Common Base.
• A class named **Dictionary** conforming to the **Dictionary** class definition
  in the Smalltalk Common Base.
• A class named **Association** conforming to the **Association** class definition
  in the Smalltalk Common Base.
• A class named **String** conforming to the **String** class definition in the
  Smalltalk Common Base.
• Objects named **true**, **false** conforming to the methods defined for **Boolean**
  objects, as specified in the Smalltalk Common Base.
• An object named **nil**, representing an object without a value.
• A global variable named **Processor**, which can be sent the message
  **activeProcess** to return the current Smalltalk process, as defined in the
document **Smalltalk-80: The Language**. This Smalltalk process must respond to
the messages **corbaContext:** and **corbaContext**.
• A class that conforms to the **CORBAParameter** protocol. This protocol defines
Smalltalk instance methods used to create and access **inout** and **out** parameters.
The protocol must support the following instance messages:

  **value**
  Answers the value associated with the instance

  **value**: **anObject**
  Resets the value associated with the instance to **anObject**

To create an object that supports the **CORBAParameter** protocol, the message
**asCORBAParameter** can be sent to any Smalltalk object. This will return a
Smalltalk object conforming to the **CORBAParameter** protocol, whose value will be
the object it was created from. The value of that **CORBAParameter** object can be
subsequently changed with the **value** message.

### 1.5 Conversion of Names to Smalltalk Identifiers

The use of underscore characters in OMG IDL identifiers is not allowed in all
Smalltalk language implementations. Thus, a conversion algorithm is required to
convert names used in OMG IDL to valid Smalltalk identifiers.

To convert an OMG IDL identifier to a Smalltalk identifier, remove each underscore
and capitalize the following letter (if it exists). In order to eliminate possible
ambiguities which may result from these conventions, an explicit naming mechanism
must also be provided by the implementation. For example, the **#pragma** directive
could be used.

For example, the OMG IDL identifiers:

  **add_to_copy_map**
  **describe_contents**

become Smalltalk identifiers
Smalltalk implementations generally require that class names and global variables have an uppercase first letter, while other names have a lowercase first letter.

1.6 Mapping for Interfaces

Each OMG IDL interface defines the operations that object references with that interface must support. In Smalltalk, each OMG IDL interface defines the methods that object references with that interface must respond to.

Implementations are free to map each OMG IDL interface to a separate Smalltalk class, map all OMG IDL interfaces to a single Smalltalk class, or map arbitrary Smalltalk classes to OMG IDL interfaces.

1.7 Memory Usage

One of the design goals is to make every Smalltalk object used in the mapping a pure Smalltalk object: namely datatypes used in mappings do not point to operating system-defined memory. This design goal permits the mapping and users of the mapping to ignore memory management issues, since Smalltalk handles this itself (via garbage collection). Smalltalk objects which are used as object references may contain pointers to operating system memory, and so must be freed in an explicit manner.

1.8 Mapping for Objects

A CORBA object is represented in Smalltalk as a Smalltalk object called an object reference. The object must respond to all messages defined by that CORBA object's interface.

An object reference can have a value which indicates that it represents no CORBA object. This value is the standard Smalltalk value nil.

1.9 Invocation of Operations

OMG IDL and Smalltalk message syntaxes both allow zero or more input parameters to be supplied in a request. For return values, Smalltalk methods yield a single result object, whereas OMG IDL allows an optional result and zero or more out or inout parameters to be returned from an invocation. In this binding, the non-void result of an operation is returned as the result of the corresponding Smalltalk method, whereas out and inout parameters are to be communicated back to the caller via instances of a class conforming to the CORBAParameter protocol, passed as explicit parameters.

For example, the following operations in OMG IDL:

```omg
boolean definesProperty(in string key);
```
void defines_property(
  in string key,
  out boolean is_defined);

are used as follows in the Smalltalk language:

```smalltalk

self
definesProperty: aString
isDefined: (aBool := nil asCORBAParameter).
```

As another example, these OMG IDL operations:

```idl
boolean has_property_protection(in string key,
  out Protection pval);

ORBStatus create_request (in Context ctx,
  in Identifier operation,
  in NVList arg_list,
  inout DynamicInvocation::NamedValue result,
  out Request request,
  in Flags req_flags);
```

would be invoked in the Smalltalk language as:

```smalltalk
aBool := self
hasPropertyProtection: aString
  pval: (protection := nil asCORBAParameter).

aStatus := ORBObject
  createRequest: aContext
  operation: anIdentifier
  argList: anNVList
  result: (result := aNamedValue asCORBAParameter)
  request: (request := nil asCORBAParameter)
  reqFlags: aFlags.
```

The return value of OMG IDL operations that are specified with a `void` return type is undefined.

### 1.10 Mapping for Attributes

OMG IDL attribute declarations are a shorthand mechanism to define pairs of simple accessing operations; one to get the value of the attribute and one to set it. Such accessing methods are common in Smalltalk programs as well, thus attribute declarations are mapped to standard methods to get and set the named attribute value, respectively.

For example:

```smalltalk
attribute   string   title;
readonly attribute  string my_name;
```
means that Smalltalk programmers can expect to use `title` and `title:` methods to get and set the `title` attribute of the CORBA object, and the `myName` method to retrieve the `my_name` attribute.

### 1.10.1 Mapping for Constants

OMG IDL allows constant expressions to be declared globally as well as in interface and module definitions. OMG IDL constant values are stored in a dictionary named `CORBAConstants` under the fully qualified name of the constant, not subject to the name conversion algorithm. The constants are accessed by sending the `at:` message to the dictionary with an instance of a `String` whose value is the fully qualified name.

For example, given the following OMG IDL specification,

```idl
module ApplicationBasics{
    const CopyDepth shallow_cpy = 4;
};
```

the `ApplicationBasics::shallow_cpy` constant can be accessed with the following Smalltalk code

```smalltalk
value := CORBAConstants at: '::ApplicationBasics::shallow_cpy'.
```

After this call, the `value` variable will contain the integral value 4.

### 1.11 Mapping for Basic Data Types

The following basic datatypes are mapped into existing Smalltalk classes. In the case of `short`, `unsigned short`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float`, `double`, `long double` and `octet`, the actual class used is left up to the implementation, for the following reasons:

- There is no standard for Smalltalk that specifies integral and floating point classes and the valid ranges of their instances.
- The classes themselves are rarely used in Smalltalk. Instances of the classes are made available as constants included in code, or as the result of computation.

The basic data types are mapped as follows.

#### 1.11.0.1 short

An OMG IDL `short` integer falls in the range \([-2^{15}, 2^{15}-1]\). In Smalltalk, a short is represented as an instance of an appropriate integral class.

#### 1.11.0.2 long

An OMG IDL `long` integer falls in the range \([-2^{31}, 2^{31}-1]\). In Smalltalk, a long is represented as an instance of an appropriate integral class.
1.11.0.3 long long
An OMG IDL long long integer falls in the range \([-2^{63},2^{63}-1]\). In Smalltalk, a long long is represented as an instance of an appropriate integral class.

1.11.0.4 unsigned short
An OMG IDL unsigned short integer falls in the range \([0,2^{16}-1]\). In Smalltalk, an unsigned short is represented as an instance of an appropriate integral class.

1.11.0.5 unsigned long
An OMG IDL unsigned long integer falls in the range \([0,2^{32}-1]\). In Smalltalk, an unsigned long is represented as an instance of an appropriate integral class.

1.11.0.6 unsigned long long
An OMG IDL unsigned long long integer falls in the range \([0,2^{64}-1]\). In Smalltalk, an unsigned long long is represented as an instance of an appropriate integral class.

1.11.0.7 float
An OMG IDL float conforms to the IEEE single-precision (32-bit) floating point standard (ANSI/IEEE Std 754-1985). In Smalltalk, a float is represented as an instance of an appropriate floating point class.

1.11.0.8 double
An OMG IDL double conforms to the IEEE double-precision (64-bit) floating point standard (ANSI/IEEE Std 754-1985). In Smalltalk, a double is represented as an instance of an appropriate floating point class.

1.11.0.9 long double
An OMG IDL long double conforms to the IEEE double extended (a mantissa of at least 64 bits, a sign bit, and an exponent of at least 15 bits) floating point standard (ANSI/IEEE Std 754-1985). In Smalltalk, a long double is represented as an instance of an appropriate floating-point class.

1.11.0.10 char
An OMG IDL character holds an 8-bit quantity mapping to the ISO Latin-1 (8859.1) character set. In Smalltalk, a character is represented as an instance of Character.
1.11.0.11 wchar

An OMG IDL wchar defines a wide character from any character set. A wide character is represented as an instance of the Character class.

1.11.0.12 boolean

An OMG IDL boolean may hold one of two values: TRUE or FALSE. In Smalltalk, a boolean is represented by the values true or false, respectively.

1.11.0.13 octet

An OMG IDL octet is an 8-bit quantity that undergoes no conversion during transmission. In Smalltalk, an octet is represented as an instance of an appropriate integral class with a value in the range [0,255].

1.12 Mapping for the Any Type

Due to the dynamic nature of Smalltalk, where the class of objects can be determined at runtime, an explicit mapping of the any type to a particular Smalltalk class is not required. Instead, wherever an any is required, the user may pass any Smalltalk object which can be mapped into an OMG IDL type. For instance, if an OMG IDL structure type is defined in an interface, a Dictionary for that structure type will be mapped. Instances of this class can be used wherever an any is expected, since that Smalltalk object can be mapped to the OMG IDL structure.

Likewise, when an any is returned as the result of an operation, the actual Smalltalk object which represents the value of the any data structure will be returned.

1.13 Mapping for Enums

OMG IDL enumerators are stored in a dictionary named CORBAConstants under the fully qualified name of the enumerator, not subject to the name conversion algorithm. The enumerators are accessed by sending the at: message to the dictionary with an instance of a String whose value is the fully qualified name.

These enumerator Smalltalk objects must support the CORBAEnum protocol, to allow enumerators of the same type to be compared. The order in which the enumerators are named in the specification of an enumeration defines the relative order of the enumerators. The protocol must support the following instance methods:

< aCORBAEnum
Answers true if the receiver is less than aCORBAEnum, otherwise answers false.

<= aCORBAEnum
Answers true if the receiver is less than or equal to aCORBAEnum, otherwise answers false.
= aCORBAEnum
Answers true if the receiver is equal to aCORBAEnum, otherwise answers false.

> aCORBAEnum
Answers true if the receiver is greater than aCORBAEnum, otherwise answers false.

>= aCORBAEnum
Answers true if the receiver is greater than or equal to aCORBAEnum, otherwise answers false.

For example, given the following OMG IDL specification,

```
module Graphics{
    enum ChartStyle
        {lineChart, barChart, stackedBarChart, pieChart};
};
```

the Graphics::lineChart enumeration value can be accessed with the following Smalltalk code

```
value := CORBAConstants at: '::Graphics::lineChart'.
```

After this call, the value variable is assigned to a Smalltalk object that can be compared with other enumeration values.

### 1.14 Mapping for Struct Types

An OMG IDL struct is mapped to an instance of the Dictionary class. The key for each OMG IDL struct member is an instance of Symbol whose value is the name of the element converted according to the algorithm in Section 1.5. For example, a structure with a field of my_field would be accessed by sending the at: message with the key #myField.

For example, given the following OMG IDL declaration:

```
struct Binding{
    Name binding_name;
    BindingType binding_type;
};
```

the binding_name element can be accessed as follows:

```
aBindingStruct at: #bindingName
```

and set as follows:

```
aBindingStruct at: #bindingName put: aName
```
1.15 Mapping for Fixed Types

An OMG IDL fixed is represented as an instance of an appropriate fractional class with a fixed denominator.

1.16 Mapping for Union Types

For OMG IDL union types, two binding mechanisms are provided: an implicit binding and an explicit binding. The implicit binding takes maximum advantage of the dynamic nature of Smalltalk and is the least intrusive binding for the Smalltalk programmer. The explicit binding retains the value of the discriminator and provides greater control for the programmer.

Although the particular mechanism for choosing implicit vs. explicit binding semantics is implementation-specific, all implementations must provide both mechanisms.

Binding semantics is expected to be specifiable on a per-union declaration basis, for example using the #pragma directive.

1.16.1 Implicit Binding

Wherever a union is required, the user may pass any Smalltalk object that can be mapped to an OMG IDL type, and whose type matches one of the types of the values in the union. Consider the following example:

structure S { long x; long y; };

union U switch (short) {
  case 1: S s;
  case 2: long l;
  default: char c;
};

In the example above, a Dictionary for structure S will be mapped. Instances of Dictionary with runtime elements as defined in structure S, integral numbers, or characters can be used wherever a union of type U is expected. In this example, instances of these classes can be mapped into one of the S, long, or char types, and an appropriate discriminator value can be determined at runtime.

Likewise, when a union is returned as the result of an operation, the actual Smalltalk object which represents the value of the union will be returned.

---

2. Although not required, implementations may choose to provide both implicit and explicit mappings for other OMG IDL types, such as structs and sequences. In the explicit mapping, the OMG IDL type is mapped to a user specified Smalltalk class.
1.16.2 Explicit Binding

Use of the explicit binding will result in specific Smalltalk classes being accepted and returned by the ORB. Each union object must conform to the CORBAUnion protocol. This protocol must support the following instance methods:

- **discriminator**
  Answers the discriminator associated with the instance.

- **discriminator: anObject**
  Sets the discriminator associated with the instance.

- **value**
  Answers the value associated with the instance.

- **value: anObject**
  Sets the value associated with the instance.

To create an object that supports the CORBAUnion protocol, the instance method `asCORBAUnion: aDiscriminator` can be invoked by any Smalltalk object. This method will return a Smalltalk object conforming to the CORBAUnion protocol, whose discriminator will be set to `aDiscriminator` and whose value will be set to the receiver of the message.

1.17 Mapping for Sequence Types

Instances of the OrderedCollection class are used to represent OMG IDL elements with the sequence type.

1.18 Mapping for String Types

Instances of the Smalltalk String class are used to represent OMG IDL elements with the string type.

1.19 Mapping for Wide String Types

An OMG IDL wide string is represented as an instance of an appropriate Smalltalk string class.

1.20 Mapping for Array Types

Instances of the Smalltalk Array class are used to represent OMG IDL elements with the array type.

1.21 Mapping for Exception Types

Each defined exception type is mapped to an instance of the Dictionary class. See Section 1.25, “Handling Exceptions,” on page 1-17 for a complete description.
1.22 Mapping for Operations

OMG IDL operations having zero parameters map directly to Smalltalk unary messages, while OMG IDL operations having one or more parameters correspond to Smalltalk keyword messages. To determine the default selector for such an operation, begin with the OMG IDL operation identifier and concatenate the parameter name of each parameter followed by a colon, ignoring the first parameter. The mapped selector is subject to the identifier conversion algorithm. For example, the following OMG IDL operations:

```c
void add_to_copy_map(
    in CORBA::ORBId id,
    in LinkSet link_set);

void connect_push_supplier(
    in EventComm::PushSupplier push_supplier);

void add_to_delete_map(
    in CORBA::ORBId id,
    in LinkSet link_set);
```

become selectors:

```smalltalk
addToCopyMap:linkSet:
connectPushSupplier:
addToDeleteMap:linkSet:
```

1.23 Implicit Arguments to Operations

Unlike the C mapping, where an object reference, environment, and optional context must be passed as parameters to each operation, this Smalltalk mapping does not require these parameters to be passed to each operation.

The object reference is provided in the client code as the receiver of a message. So although it is not a parameter on the operation, it is a required part of the operation invocation.

This mapping defines the `CORBAExceptionEvent` protocol to convey exception information in place of the environment used in the C mapping. This protocol can either be mapped into native Smalltalk exceptions or used in cases where native Smalltalk exception handling is unavailable.

A context expression can be associated with the current Smalltalk process by sending the message `corbaContext:` to the current process, along with a valid context parameter. The current context can be retrieved by sending the `corbaContext` message to the current process.

The current process may be obtained by sending the message `activeProcess` to the Smalltalk global variable named `Processor`. 
1.24 Argument Passing Considerations

All parameters passed into and returned from the Smalltalk methods used to invoke operations are allocated in memory maintained by the Smalltalk virtual machine. Thus, explicit `free()`ing of the memory is not required. The memory will be garbage collected when it is no longer referenced.

The only exception is object references. Since object references may contain pointers to memory allocated by the operating system, it is necessary for the user to explicitly free them when no longer needed. This is accomplished by using the operation `release` of the `CORBA::Object` interface.

1.25 Handling Exceptions

OMG IDL allows each operation definition to include information about the kinds of run-time errors which may be encountered. These are specified in an exception definition which declares an optional error structure which will be returned by the operation should an error be detected. Since Smalltalk exception handling classes are not yet standardized between existing implementations, a generalized mapping is provided.

In this binding, an IDL compiler creates exception objects and populates the `CORBAConstants` dictionary. These exception objects are accessed from the `CORBAConstants` dictionary by sending the `at:` message with an instance of a `String` whose value is the fully qualified name. Each exception object must conform to the `CORBAExceptionEvent` protocol. This protocol must support the following instance methods:

```smalltalk
corbaHandle: aHandlerBlock do: aBlock
```

Exceptions may be handled by sending an exception object the message `corbaHandle:do:` with appropriate handler and scoping blocks as parameters. The `aBlock` parameter is the Smalltalk block to evaluate. It is passed no parameters. The `aHandlerBlock` parameter is a block to evaluate when an exception occurs. It has one parameter: a Smalltalk object which conforms to the `CORBAExceptionValue` protocol.

```smalltalk
corbaRaise
```

Exceptions may be raised by sending an exception object the message `corbaRaise`.

```smalltalk
corbaRaiseWith: aDictionary
```

Exceptions may be raised by sending an exception object the message `corbaRaiseWith:`. The parameter is expected to be an instance of the Smalltalk `Dictionary` class, as described below.

For example, given the following OMG IDL specification,
interface NamingContext {
    ...

    exception NotEmpty {};
    void destroy ()
        raises (NotEmpty);
    ...
}

the NamingContext::NotEmpty exception can be raised as follows:

(CORBAConstants at: '::NamingContext::NotEmpty')
corbaRaise.

The exception can be handled in Smalltalk as follows:

(CORBAConstants at: '::NamingContext::NotEmpty')
corbaHandle: [:ev | "error handling logic here"]
do: [aNamingContext destroy].

1.26 Exception Values

OMG IDL allows values to be returned as part of the exception. Exception values are constructed using instances of the Smalltalk Dictionary class. The keys of the dictionary are the names of the elements of the exception, the names of which are converted using the algorithm in “Conversion of Names to Smalltalk Identifiers” on page 1-7. The following example illustrates how exception values are used:

interface NamingContext {
    ...

    exception CannotProceed {
        NamingContext cxt;
        Name rest_of_name;
    };
    Object resolve (in Name n)
        raises (CannotProceed);
    ...
}

would be raised in Smalltalk as follows:

(CORBAConstants at: '::NamingContext::CannotProceed')
corbaRaiseWith: (Dictionary
    with: (Association key: #cxt value:
        aNamingContext)
    with: (Association key: #restOfName value:
        aName)).
1.26.1 The CORBAExceptionValue Protocol

When an exception is raised, the exception block is evaluated, passing to it one argument that conforms to the CORBAExceptionValue protocol. This protocol must support the following instance messages:

**corbaExceptionValue**

Answers the Dictionary the exception was raised with.

Given the NamingContext interface defined in the previous section, the following code illustrates how exceptions are handled:

```smalltalk
(CORBAConstants at: '::NamingContext::NotEmpty')
corbaHandle: [:ev |
cxt := ev corbaExceptionValue at: #cxt.
restOfName := ev corbaExceptionValue at: #restOfName]
do: [aNamingContext destroy].
```

In this example, the cxt and restOfName variables will be set to the respective values from the exception structure, if the exception is raised.

1.26.2 Pseudo-Objects Mapping Overview

CORBA defines a small set of standard interfaces which define types and operations for manipulating object references, for accessing the Interface Repository, and for Dynamic Invocation of operations. Other interfaces are defined in pseudo OMG IDL (PIDL) to represent in a more abstract manner programmer access to ORB services which are provided locally. These PIDL interfaces sometimes resort to non-OMG IDL constructs, such as pointers, which have no meaning to the Smalltalk programmer. This chapter specifies the minimal requirements for the Smalltalk mapping for PIDL interfaces. The operations are specified below as protocol descriptions.

Parameters with the name aCORBAObject are expected to be Smalltalk objects, which can be mapped to an OMG IDL interface or data type.

Unless otherwise specified, all messages are defined to return undefined objects.

1.27 CORBA::Request

The CORBA::Request interface is mapped to the CORBARequest protocol, which must include the following instance methods:

**addArg**: aCORBANamedValue

Corresponds to the add_arg operation.
invoke
Corresponds to the `invoke` operation with the `invoke_flags` set to 0.

invokeOneway
Corresponds to the `invoke` operation with the `invoke_flags` set to `CORBA::INV_NO_RESPONSE`.

send
Corresponds to the `send` operation with the `invoke_flags` set to 0.

sendOneway
Corresponds to the `send` operation with the `invoke_flags` set to `CORBA::INV_NO_RESPONSE`.

pollResponse
Corresponds to the `get_response` operation, with the `response_flags` set to `CORBA::RESP_NO_WAIT`. Answers `true` if the response is complete, `false` otherwise.

getResponse
Corresponds to the `get_response` operation, with the `response_flags` set to 0.

1.28 CORBA::Context

The CORBA::Context interface is mapped to the `CORBAContext` protocol, which must include the following instance methods:

`setOneValue: anAssociation`
Corresponds to the `set_one_value` operation.

`setValues: aCollection`
Corresponds to the `set_values` operation. The parameter passed in should be a collection of `Associations`.

`getValues: aString`
Corresponds to the `get_values` operation without a scope name and op_flags = `CXT_RESTRICT_SCOPE`. Answers a collection of `Associations`.

`getValues: aString propName: aString`
Corresponds to the `get_values` operation with `op_flags` set to `CXT_RESTRICT_SCOPE`. Answers a collection of `Associations`.

`getValuesInTree: aString propName: aString` 

Corresponds to the `get_values` operation with `op_flags` set to `0`. Answers a collection of `Associations`.

`deleteValues: aString` 

Corresponds to the `delete_values` operation.

`createChild: aString` 

Corresponds to the `create_child` operation. Answers a Smalltalk object conforming to the `CORBAContext` protocol.

`delete` 

Corresponds to the `delete` operation with flags set to `0`.

`deleteTree` 

Corresponds to the `delete` operation with flags set to `CTX_DELETE_DESCENDENTS`.

### 1.29 CORBA::Object

The `CORBA::Object` interface is mapped to the `CORBAObject` protocol, which must include the following instance methods:

`getImplementation` 

Corresponds to the `get_implementation` operation. Answers a Smalltalk object conforming to the `CORBAImplementationDef` protocol.

`getInterface` 

Corresponds to the `get_interface` operation. Answers a Smalltalk object conforming to the `CORBAInterfaceDef` protocol.

`isNil` 

Corresponds to the `is_nil` operation. Answers `true` or `false` indicating whether or not the object reference represents an object.

`createRequest: aCORBAContext operation: aCORBAIdentifier argList: aCORBANVListOrNil`
result: aCORBAParameter
request: aCORBAParameter
reqFlags: flags

Corresponds to the create_request operation.

duplicate

Corresponds to the duplicate operation. Answers a Smalltalk object representing an object reference, conforming to the interface of the CORBA object.

release

Corresponds to the release operation.

1.30 CORBA::ORB

The CORBA::ORB interface is mapped to the CORBAORB protocol, which must include the following instance methods:

objectToString: aCORBAObject

Corresponds to the object_to_string operation. Answers an instance of the String class.

stringToObject: aString

Corresponds to the string_to_object operation. Answers an object reference, which will be an instance of a class which corresponds to the InterfaceDef of the CORBA object.

createOperationList: aCORBAOperationDef

Corresponds to the create_operation_list operation. Answers an instance of OrderedCollection of Smalltalk objects conforming to the CORBANamedValue protocol.

getDefaultContext

Corresponds to the get_default_context operation. Answers a Smalltalk object conforming to the CORBAContext protocol.

sendMultipleRequests: aCollection

3. The semantics of this operation will have no meaning for those implementations that rely exclusively on the Smalltalk memory manager.
Corresponds to the `send_multiple_requests` operation with the `invoke_flags` set to 0. The parameter passed in should be a collection of Smalltalk objects conforming to the `CORBARequest` protocol.

```smalltalk
sendMultipleRequestsOneway: aCollection
```

Corresponds to the `send_multiple_requests` operation with the `invoke_flags` set to `CORBA::INV_NO_RESPONSE`. The parameter passed in should be a collection of Smalltalk objects conforming to the `CORBARequest` protocol.

```smalltalk
pollNextResponse
```

Corresponds to the `get_next_response` operation, with the `response_flags` set to `CORBA::RESP_NO_WAIT`. Answers `true` if there are completed requests pending, `false` otherwise.

```smalltalk
getNextResponse
```

Corresponds to the `get_next_response` operation, with the `response_flags` set to 0.

### 1.31 CORBA::NamedValue

PIDL for C defines `CORBA::NamedValue` as a struct while C++-PIDL specifies it as an interface. `CORBA::NamedValue` in this mapping is specified as an interface that conforms to the `CORBANamedValue` protocol. This protocol must include the following instance methods:

```smalltalk
name
```

Answers the name associated with the instance.

```smalltalk
name: aString
```

Resets the name associated with instance to `aString`.

```smalltalk
value
```

Answers the value associated with the instance.

```smalltalk
value: aCORBAObject
```

Resets the value associated with instance to `aCORBAObject`.

```smalltalk
flags
```

Answers the flags associated with the instance.
flags: argModeFlags

 Resets the flags associated with instance to argModeFlags.

 To create an object that supports the CORBANamedValue protocol, the instance method
asCORBANamedValue: aName flags: argModeFlags can be
invoked by any Smalltalk object. This method will return a Smalltalk object
conforming to the CORBANamedValue protocol, whose attributes associated with
the instance will be set appropriately.

1.32 CORBA::NVList

 The CORBA::NVList interface is mapped to the equivalent of the OMG IDL
definition typedef sequence<NamedValue> NVList;

 Thus, Smalltalk objects representing the NVList type should be instances of the
OrderedCollection class, whose elements are Smalltalk objects conforming to the
CORBANamedValue protocol.
Appendix A - Glossary

This appendix includes a list of Smalltalk terms.

A.1 Glossary Terms

<table>
<thead>
<tr>
<th>Smalltalk object</th>
<th>An object defined using the Smalltalk language.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message</td>
<td>Invocation of a Smalltalk method upon a Smalltalk object.</td>
</tr>
<tr>
<td>Message Selector</td>
<td>The name of a Smalltalk message. In this document, the message selectors are denoted by just the message name when the class or protocol they are associated with is given in context, otherwise the notation <code>class&gt;&gt;method</code> or <code>protocol&gt;&gt;method</code> will be used to explicitly denote the class or protocol the message is associated with.</td>
</tr>
<tr>
<td>Method</td>
<td>The Smalltalk code associated with a message.</td>
</tr>
<tr>
<td>Class</td>
<td>A Smalltalk class.</td>
</tr>
<tr>
<td>Protocol</td>
<td>A set of messages that a Smalltalk object must respond to. Protocols are used to describe the behavior of Smalltalk objects without specifying their class.</td>
</tr>
<tr>
<td>CORBA Object</td>
<td>An object defined in OMG IDL, accessed and implemented through an ORB.</td>
</tr>
<tr>
<td>Object Reference</td>
<td>A value which uniquely identifies an object.</td>
</tr>
<tr>
<td>IDL compiler</td>
<td>Any software that accesses OMG IDL specifications and generates or maps Smalltalk code that can be used to access CORBA objects.</td>
</tr>
</tbody>
</table>
Index

A
aBool 1-9
aCORBAObject 1-19, 1-23
addArg 1-19
aDiscriminator instance method 1-15
Alternative Mappings for C++ 1-25
any type 1-4, 1-12

B
boolean type 1-12

C
corruptors iv
corruptors iv
getResponse instance method 1-20
cORBAConstants 1-10, 1-12, 1-17, 1-18
cORBAContext protocol 1-20, 1-22
cORBAEnum protocol 1-12
cORBAExceptionEvent 1-17
cORBAExceptionEvent protocol 1-16
cORBAExceptionValue protocol 1-17, 1-19
cORBANamedValue protocol 1-22, 1-24
cORBAObject protocol 1-21
cORBAORB protocol 1-22
cORBAParameter 1-8
corbaRaise message 1-17
cORBARequest protocol 1-19
cORBAUnion protocol 1-15
core, compliance iv
createRequest 1-9

discriminator instance method 1-15
double 1-11

D
dictionary 1-18
dictionary 1-18
discriminator instance method 1-15
discriminator instance method 1-15
double 1-11

F
float type 1-11

G
global name
and Smalltalk 1-5

I
interoperability, compliance iv
interworking
compliance iv

L
long double type 1-11
long long type 1-11
long type 1-10

N
namespace 1-6
NamingContext interface
    mapped to Smalltalk 1-18
nil 1-8
NVList 1-24

O
octet type 1-12

P
pragma directive
use in Smalltalk mapping 1-14

S
sequence type 1-15
short type 1-10
Smalltalk 1-11
aBindingStruct 1-13
aBool 1-9
aCORBAObject 1-19, 1-23
active Process message 1-16
add_arg operation 1-19
addArg instance method 1-19
aDiscriminator instance method 1-15
any 1-12
argList 1-9
array class 1-15
array type 1-15
Association 1-20
at message 1-12, 1-17
boolean 1-12
char 1-11
Character 1-11
Common Base 1-5
cORBAConstants 1-10, 1-12, 1-17
corbaContext message 1-16
cORBAContext protocol 1-20, 1-22
cORBAEnum protocol 1-12
cORBAExceptionEvent 1-16
cORBAExceptionValue protocol 1-17, 1-19
cORBANamedValue protocol 1-22, 1-24
cORBAObject protocol 1-21
cORBAORB protocol 1-22
cORBAParameter protocol 1-8
corbaRaise message 1-17
cORBARequest protocol 1-19
cORBAUnion protocol 1-15
create_child operation 1-21
create_operation_list operation 1-22
create_request operation 1-22
createChild instance method 1-21
createOperationList instance method 1-22
createRequest 1-9, 1-22
dict 1-19
delete instance method 1-21
delete operation 1-21
design of mapping 1-4
dictionary 1-13, 1-15
discriminator instance method 1-15
duplicate 1-22
exceptions 1-15
explicit vs implicit mappings 1-14
flags instance method 1-24
float 1-11
garbage collection 1-8, 1-17
get_next_response operation 1-23
get_response operation 1-20
getAddress instance method 1-22
invoke instance method 1-20

Smalltalk Language Mapping
Index

invoke operation 1-20
invokeOneway instance method 1-20
long 1-10
long double 1-11
long long 1-11
memory management 1-8, 1-22
memory management for object references 1-17
mini-glossary 1-25
name instance method 1-23
namespace 1-6
nil 1-8
NVlist type and OrderedCollection class 1-24
object_to_string operation 1-22
objectToString instance method 1-22
octet 1-12
operation 1-9
OrderedCollection class 1-15
overview of mapping 1-3
pollNextResponse instance method 1-23
pollResponse instance method 1-20
Processor variable 1-16
reference books 1-5
release operation 1-22
reqFlags 1-9
result 1-9
send instance method 1-20
send operation 1-20
send_multiple_requests operation 1-22
sendMultipleRequests instance method 1-22
sequence 1-15
set and get value instance methods 1-20
set and get value operations 1-20
set value operations 1-20
short 1-10
String class 1-15
string type 1-15
struct type 1-13
underscore characters in mapping 1-7
unsigned long 1-11
unsigned long long 1-11
unsigned short and long 1-11
value instance method 1-15, 1-23
Value instance methods 1-20
wchar 1-12

code 1-4

union type 1-4
unsigned long long type 1-11
unsigned long type 1-11
unsigned short type 1-11
value instance method 1-15

wchar type 1-12