Mapping of OMG IDL to Smalltalk 21

Contents

This chapter contains the following sections.

21.1 *Mapping Summary*

Table 21-1 provides a brief description of the mapping of OMG IDL constructs to the Smalltalk language, and where in this chapter they are discussed.

Table 21-1 Summary of this Chapter

OMG IDL Construct	Smalltalk Mapping	See Section
Interface	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.	"Mapping for Interfaces" on page 21-7
Object Reference	Smalltalk object that represents a CORBA object. The Smalltalk object must respond to all messages defined by a CORBA object's interface.	"Mapping for Objects" on page 21-8

Table 21-1 Summary of this Chapter *(Continued)*

OMG IDL Construct	Smalltalk Mapping	See Section
Operation	Smalltalk message.	"Mapping for Operations" on page 21-15
Attribute	Smalltalk message	"Mapping for Attributes" on page 21-9
Constant	Smalltalk objects available in the CORBAConstants dictionary.	"Mapping for Constants" on page 21-9
Integral Type	Smalltalk objects that conform to the **Integer** class.	"Mapping for Basic Data Types" on page 21-10
Floating Point Type	Smalltalk objects which conform to the **Float** class.	"Mapping for Basic Data Types" on page 21-10
Boolean Type	Smalltalk **true** or **false** objects.	"Mapping for Basic Data Types" on page 21-10
Enumeration Type	Smalltalk objects which conform to the ***CORBAEnum*** protocol.	"Mapping for Enums" on page 21-12
Any Type	Smalltalk objects that can be mapped into an OMG IDL type.	"Mapping for the Any Type" on page 21-11
Structure Type	Smalltalk object that conforms to the **Dictionary** class.	"Mapping for Struct Types" on page 21-13
Fixed Type		"Mapping for Fixed Types" on page 21-13
Union Type	Smalltalk object that maps to the possible value types of the OMG IDL union or that conform to the ***CORBAUnion*** protocol.	"Mapping for Union Types" on page 21-13
Sequence Type	Smalltalk object that conforms to the **OrderedCollection** class.	"Mapping for Sequence Types" on page 21-14
String Type	Smalltalk object that conforms to the **String** class.	"Mapping for String Types" on page 21-15
Wide String Type		"Mapping for Wide String Types" on page 21-15
Array Type	Smalltalk object that conforms to the **Array** class.	"Mapping for Array Types" on page 21-15
Exception Type	Smalltalk object that conforms to the **Dictionary** class.	"Mapping for Exception Types" on page 21-15

21.2 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 **TypeCode** 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.

21.2.1 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:

- Goldberg, Adele and Robson, David. *Smalltalk-80: The Language*. Addison-Wesley Publishing Company, Reading, MA. 1989.
- *Smalltalk Portability: A Common Base*. ITSC Technical Bulletin GG24-3093, IBM, Boca Raton, FL. September 1992.

(Throughout the Smalltalk chapters, *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.

21.3 Implementation Constraints

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

21.3.1 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 “Conversion of Names to Smalltalk Identifiers” on page 21-7 for a description of the name conversion rules.

As an example of 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**. In order 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.

21.3.2 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.

```
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).

21.4 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.
- 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 which 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

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 "Mapping for Union Types" on page 21-13 for a complete description.

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.

21.5 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

addToCopyMap
describeContents

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

21.6 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.

21.7 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.

21.8 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`**.

21.9 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:

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

are used as follows in the Smalltalk language:

```
aBool := self definesProperty: aString.

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

As another example, these OMG IDL operations:

```
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:

```
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.

21.10 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:

```
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.

21.10.1 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,

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

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

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

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

21.11 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:

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.

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.

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.

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.

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.

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.

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.

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.

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.

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`**.

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.

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.

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].

21.12 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.

21.13 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.

21.14 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 21.5. 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
```

21.15 Mapping for Fixed Types

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

21.16 Mapping for Union Types

For OMG IDL union types, two binding mechanisms are provided: an *implicit* binding and an *explicit* binding.[2] 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.

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.

21.16.1 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 an **union** is returned as the result of an operation, the actual Smalltalk object which represents the value of the **union** will be returned.

21.16.2 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.

21.17 Mapping for Sequence Types

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

21.18 Mapping for String Types

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

21.19 Mapping for Wide String Types

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

21.20 Mapping for Array Types

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

21.21 Mapping for Exception Types

Each defined exception type is mapped to an instance of the **`Dictionary`** class. See "Handling Exceptions" on page 21-16 for a complete description.

21.22 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:

```
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:

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

21.23 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**.

21.24 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.

21.25 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:

`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.

`corbaRaise`

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

`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].
```

21.26 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 21-7. 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)).
```

21.26.1 The CORBAExceptionValue Protocol

When an exception is raised, the exception block is evaluated, passing it one argument which 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:

```
(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. 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.

21.27 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.

21.28 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 **`Association`**s.

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

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

`getValuesInTree: aString propName: aString`
Corresponds to the **get_values** operation with op_flags set to **`0`**. Answers a collection of **`Association`**s.

`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`**.

21.29 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`[3]
Corresponds to the **release** operation.

21.30 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`
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.

`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.

`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.

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

3. The semantics of this operation will have no meaning for those implementations that rely exclusively on the Smalltalk memory manager.

21.31 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:

`name`
Answers the name associated with the instance.

`name: aString`
Resets the name associated with instance to **`aString`**.

`value`
Answers the value associated with the instance.

`value: aCORBAObject`
Resets the value associated with instance to **`aCORBAObject`**.

`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.

21.32 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

Smalltalk object	An object defined using the Smalltalk language.
Message	Invocation of a Smalltalk method upon a Smalltalk object.
Message Selector	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 **`class`**>>**`method`** or ***`protocol`***>>**`method`** will be used to explicitly denote the class or protocol the message is associated with.
Method	The Smalltalk code associated with a message.
Class	A Smalltalk class.
Protocol	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.
CORBA Object	An object defined in OMG IDL, accessed and implemented through an ORB.
Object Reference	A value which uniquely identifies an object.
IDL compiler	Any software that accesses OMG IDL specifications and generates or maps Smalltalk code that can be used to access CORBA objects.