**Date**: May 2012

OBJECT MANAGEMENT GROUP

# Ruby CORBA Language Mapping

*Version 1.1 with change bars*

---

Normative reference:  http://www.omg.org/spec/RCLM/1.1

---

OMG Formal document number: formal/2012-05-02

# OMG's Issue Reporting Procedure

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 listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/report_issue.htm).

# Table of Contents

# Preface

## About the Object Management Group

### OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG’s specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

## OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from this URL:

*http://www.omg.org/spec*

Specifications are organized by the following categories:

### Business Modeling Specifications

### Middleware Specifications

- **CORBA/IIOP**
- **Data Distribution Services**
- **Specialized CORBA**

### IDL/Language Mapping Specifications

### Modeling and Metadata Specifications

- **UML, MOF, CWM, XMI**
- **UML Profile**

### Modernization Specifications

**Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications**

- **CORBAServices**
- **CORBAFacilities**

**OMG Domain Specifications**

**CORBA Embedded Intelligence Specifications**

**CORBA Security Specifications**

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: pubs@omg.org

Certain OMG specifications are also available as ISO standards. Please consult http://www.iso.org

## Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

**Helvetica/Arial - 10 pt. Bold:** OMG Interface Definition Language (OMG IDL) and syntax elements.

**`Courier - 10 pt. Bold:`** Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

**Note –** Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

## Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to http://www.omg.org/report_issue.htm.

# 1 Scope

The CORBA Language Mapping specifications contain language mapping information for several languages. Each language is described in a separate stand-alone volume.

This particular specification explains how OMG IDL constructs are mapped to the constructs of the Ruby programming language.

## 1.1 Alignment with CORBA

This language mapping is aligned with CORBA, v3.1 (formal/2008-01-04).

# 2 Conformance

The Ruby mapping tries to avoid limiting the implementation freedoms of ORB developers. For each OMG IDL and CORBA construct, the Ruby mapping explains the syntax and semantics of using the construct from Ruby. A client or server program conforms to this mapping (is CORBA-Ruby compliant) if it uses the constructs as described in the Ruby mapping chapters. An implementation conforms to this mapping if it correctly executes any conforming client or server program. A conforming client or server program is therefore portable across all conforming implementations.

## 2.1 Ruby Implementation Requirements

The mapping described here assumes that the target Ruby environment supports all the features described in the Programming Ruby; The Pragmatic Programmers' Guide.

## 2.2 No Implementation Descriptions

This mapping does not contain implementation descriptions. It avoids details that would constrain implementations, but still allows clients to be fully source-compatible with any compliant implementation. Some examples show possible implementations, but these are not required implementations.

# 3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

- OMG CORBA 3.1 specification (formal/2008-01-04): http://www.omg.org/spec/CORBA/3.1/
- Programming Ruby; The Pragmatic Programmers' Guide. Pragmatic Bookshelf, September 2004. ISBN 0974514055.

# 4 Terms and Definitions

For the purposes of this specification, the terms and definitions given in the normative reference section and the following Symbols apply.

# 5 Symbols

List of symbols/abbreviations.

GIOP - Generic Inter-ORB protocol

ORB - Object Request Broker

CORBA - Common Object Request Broker Architecture

IOR - Interoperable Object Reference

# 6 Additional Information

## 6.1 How to Read this Specification

The rest of this document contains the CORBA language mapping for the Ruby language.

## 6.2 Acknowledgements

The following companies submitted and/or supported parts of this specification:

- Martin Corino, Remedy IT

# 7 Ruby Language Mapping

## 7.1 Mapping Overview

The mapping of IDL to Ruby presented here does not prescribe a specific implementation. It follows the guidelines presented in Clause 1.1 of the C Language Mapping (formal/1999-07-35; available at this URL: http://www.omg.org/spec/C/1.0/). The Ruby language features used in this mapping are available since Ruby 1.8, most of them are available in previous releases.

This document covers the following aspects of implementing CORBA-based architectures in Ruby:

- Representation of IDL types, constants, and exceptions in Ruby
- Invocation of methods on a CORBA object using a generated stub
- Invoking methods dynamically
- Providing object implementations using generated stubs
- Access to ORB services

An implementation of this specification provides the predefined module CORBA. All names qualified with the CORBA module are also provided by the implementation.

## 7.2 Using Scoped Names

Ruby implements a module concept that is compatible with the IDL scoping mechanisms. Ruby naming conventions (partially hardwired into the language) differ however. The following naming conventions apply:

- Constant names (which in Ruby include module and class names) *must* start with an uppercase alphabetical character.
- Method and attribute names should start with a lowercase alphabetical character or underscore.

Scoped names are therefore translated into Ruby using the following rules:

- IDL modules are mapped onto Ruby modules. Nesting in IDL is supported without restrictions.
- IDL interfaces are mapped onto Ruby modules. The Ruby concept of Mixins applies neatly to the CORBA concept of narrowing interfaces.
- IDL definitions in global scope will also have global scope definitions in Ruby.
- The first character of IDL module, interface, or constant names is forced to uppercase when mapped into Ruby.
- The first character of IDL attribute or method names is forced to lowercase when mapped into Ruby.

To avoid conflicts, every use in OMG IDL of a Ruby keyword as an identifier is mapped into the same name preceded by the prefix ‘r_’ or ‘R_.’ For example, an IDL interface named **alias** would be named **R_alias** when its name is mapped into Ruby.

**Table 7.1 - Ruby Keywords**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| __FILE__ | and | def | end | in | or | self | unless |
| __LINE__ | begin | defined? | ensure | module | redo | super | until |
| BEGIN | break | do | false | next | rescue | then | when |
| END | case | else | for | nil | retry | true | while |
| alias | class | elsif | if | not | return | undef | yield |

Likewise every use in OMG IDL of a builtin Ruby constant name as a name for an unscoped module or interface is mapped into the same name preceded by the prefix ‘R_.’ For example, an IDL interface named **Array** would be named **R_Array** when its name is mapped into Ruby. When however the interface is declared within the scope of a module its name would be left untouched.

**Table 7.2 - Ruby reserved constant names**

| | | | | | | |
|---|---|---|---|---|---|---|
| Array | Bignum | Binding | Class | Continuation | Dir | Exception |
| FalseClass | File | Fixnum | Float | Hash | Integer | IO |
| MatchData | Method | Module | NilClass | Numeric | Object | Proc |
| Process | Range | Regexp | String | Struct | Symbol | Thread |
| ThreadGroup | Time | TrueClass | UnboundMethod | Comparable | Enumerable | Errno |
| FileTest | GC | Kernel | Marshal | Math | ObjectSpace | Signal |

Finally every use in OMG IDL of the standard name of a method of the Ruby Object class as a name for a member of an IDL construct is mapped into the same name preceded by the prefix ‘r_.’ For example, the method of an IDL interface named **to_s** would be named **r_to_s** when mapped into Ruby.

**Table 7.3 - Ruby reserved member names**

| | | | | |
|---|---|---|---|---|
| __id__ | __send__ | abort | at_exit | autoload |
| binding | callcc | caller | catch | chomp |
| chop | clone | display | dup | eval |
| exec | exit | extend | fail | fork |
| format | freeze | getc | gets | global_variables |

**Table 7.3 - Ruby reserved member names**

| | | | | |
|---|---|---|---|---|
| gsub | hash | id | initialize | initialize_copy |
| inspect | instance_eval | instance_variable_get | instance_variable_set | instance_variables |
| irb_binding | lambda | load | local_variables | loop |
| method | method_missing | methods | object_id | open |
| p | print | printf | private_methods | proc |
| protected_methods | public_methods | putc | puts | raise |
| rand | readline | readlines | remove_instance_ variable | require |
| scan | select | send | set_trace_func | singleton_method_ added |
| singleton_method_ removed | singleton_method_ undefined | singleton_methods | sleep | split |
| sprintf | srand | sub | syscall | system |
| taint | test | throw | to_a | to_s |
| trace_var | trap | type | untaint | untrace_var |
| warn | | | | |

## 7.3 Mapping for Modules

A module defines a scope and as such is mapped to a Ruby module using the naming conventions described in 7.2,'Using Scoped Names.'

```
 // IDL
module M
{
   // definitions
};
```

```
# Ruby
module M
{
     ## definitions
}
```

## 7.4 Mapping for Interfaces

An interface is mapped to a Ruby module (using the naming conventions described in 7.2,'Using Scoped Names)' that contains public definitions of the types, constants, operations, and exceptions defined in the interface.

A CORBA Ruby compliant program cannot create or hold an instance of an interface module (this is in fact prohibited by the Ruby language that does not allow creating instances of modules).

In essence the generated module is what the Ruby language calls a Mixin; an interface definition containing type definitions, constants, and instance method implementations that can be “mixed in” with a class. This example shows the behavior of the mapping of an interface.

```
// IDL
interface myIntf
{
   struct S { short field; };
};
```

```
# Ruby
# Conformant uses
s = MyIntf::S.new  ## create a struct instance
s.field = 3        ## field access
# Non-conformant uses:
# one cannot create an instance of an interface class...
a = MyIntf.new
```

### 7.4.1 Object References

The use of an interface type in OMG IDL denotes an object reference. As Ruby variables do not have an intrinsic type they can hold a reference to any object including CORBA object references. Since Ruby variables hold object references by nature that are ‘cleaned up’ using a mark/sweep garbage collection mechanism, there is also no need for special ‘reference holder’ types like the _var types from the C++ language mapping. The garbage collection mechanism has as a downside the effect that (CORBA) resources may be retained until the garbage collector kicks in at what may be, from the application developers point of view, an inopportune moment. To facilitate more ‘planned’ resource release, the implementation defines a non-standard extension to the CORBA object reference interface; the `#_free_ref()` method. This method releases any resources allocated to a CORBA object reference. After calling this method calling `CORBA::is_nil` for the object reference will return true.

Since CORBA object references are represented by standard Ruby object references performing operations on CORBA, objects and/or referencing attribute values follow normal Ruby language rules. This example shows the code to perform operations and reference attributes:

```
# Ruby
obj = ... ## somehow obtain an object reference
# perform operation
s = obj.get_string()
# reference attribute value
v = obj.my_value
```

The way Ruby handles values, objects, and argument passing does however have an effect on the way argument passing and handling OUT values and return values are mapped as is discussed in 7.23,'Mapping for Operations and Attributes.'

## 7.5 Interface Inheritance

OMG IDL interface inheritance is mapped onto Ruby module Mixin methods. In the Ruby mapping modules representing derived OMG IDL interfaces get their base modules (representing the IDL base interface(s)) mixed in as shown in this example:

```
// IDL
interface myBaseIntf { ... };
interface myDerivedIntf : myBaseIntf { ... };


# Ruby
module MyBaseIntf
  ...
end
module MyDerivedIntf
  include MyBaseIntf
  ...
end
```

### 7.5.1 Narrowing Interfaces

The mapping for an interface defines a module method named `_narrow` that returns a new object reference given an existing reference. The `_narrow` method returns a nil object reference if the given reference is nil. The parameter to `_narrow` is a reference of an object of any interface type. If the actual (runtime) type of the parameter object can be narrowed to the requested interface's type, then `_narrow` will return a valid object reference.

For example, suppose A, B, C, and D are interface types, and D inherits from C, which inherits from B, which in turn inherits from A. If an object reference to a C object is narrowed to a variable called ap, then:

- A::_narrow(ap) returns a valid object reference
- B::_narrow(ap) returns a valid object reference
- C::_narrow(ap) returns a valid object reference
- D::_narrow(ap) raises a CORBA system exception

Narrowing to A, B, and C all succeed because the object supports all those interfaces. The D::_narrow fails because the C object does not support the D interface. For another example, suppose A, B, C, and D are interface types. C inherits from B, and both B and D inherit from A. Now suppose that an object of type C is passed to a function as an A. If the function calls B::_narrow or C::_narrow, a new object reference will be returned. A call to D::_narrow will fail. If successful, the _narrow function creates a new object reference.

## 7.6 Nil Object Reference

As Ruby variables do not have an intrinsic type there is no need for a specific Nil Object Reference. Instead the Ruby **`nil`** value will be returned whenever a nil object reference is expected. The CORBA helper method **`CORBA::is_nil`** will return true for any value that is either a Ruby **`nil`** or an Object reference for which the **`_is_nil`** method returns true, i.e.,

```
# Ruby
my_ref = nil
puts "TRUE" if CORBA::is_nil(my_ref)
# ... retrieve object reference somewhere
if !my_obj.nil?
  puts "TRUE if my_obj._is_nil?()" if CORBA::is_nil(my_obj)
end
```

## 7.7 Mapping for Constants

OMG IDL constants are mapped directly to a Ruby constant definition taking into account the scoped names naming conventions as described in 7.2,'Using Scoped Names.' The following shows an example of the constants mapping.

```
// IDL
const string name = "testing";
interface A
{
   const float pi = 3.14159;
};
```

```
# Ruby
Name = "testing"

module A
...
  Pi = 3.14159
...
end
```

In certain situations, use of a constant in OMG IDL must generate the constant's value instead of the constant's name. For example:

```
// IDL
interface A
{
   const long n = 10;
   typedef long V[n];
};
```

```
// Ruby
module A
...
  N = 10
  class V < Array
    def V._tc
      @@tc_V ||= CORBA::TypeCode::Alias.new('IDL:V:1.0', 'V', self,
        CORBA::TypeCode::Array.new(CORBA._tc_long, [10]))
```

```
    end
  end # typedef V
...
end
```

### 7.7.1 Wide Character and Wide String Constants

As Ruby does not provide intrinsic language types for representing wide character and wide string constants these data types are mapped on Ruby integer and integer array types respectively. The following gives an example of this mapping.

```
// IDL
const wchar myWChar = L'a';

const wstring myWString = L"abc\u1234";


# Ruby
MyWChar = 97

MyWString = [97,98,99,4660]
```

### 7.7.2 Fixed Point Constants

This type is not supported in this version of the Language mapping.

## 7.8 Mapping for Basic Data Types

Because Ruby does not require type information for operation declarations, it is not necessary to introduce standardized type names, unlike the C or C++ mappings. Instead, the mapping of types to dynamic values is specified here. For most of the simple types, it is obvious how values of these types can be created. For the other types, the interface for constructing values is also defined. The mappings for the basic types are shown in Table 7.4.

**Table 7.4 - Basic Data Type mappings**

| | |
|---|---|
| octet | Integer |
| short | Integer |
| long | Integer |
| unsigned short | Integer |
| unsigned long | Integer |
| long long | Integer |
| unsigned long long | Integer |
| float | Float |

**Table 7.4 - Basic Data Type mappings**

| | |
|---|---|
| double | Float |
| long double | CORBA::LongDouble |
| boolean | TrueClass or FalseClass |
| char | String of length 1 or Integer |
| wchar | Integer |

For IN (INOUT) arguments the mapping implementation honors the Ruby Duck typing principal by allowing for certain data types implicit conversions using the strict type conversion methods **`#to_str`** and **`#to_in`**.

Implicit conversion is applied in the following cases:

- Where the formal expected Ruby data type is Integer passing an object responding to **`#to_int`** is allowed.
- Where the formal expected Ruby data type is String passing an object responding to **`#to_str`** is allowed.

For the boolean type Ruby defines two distinct instances **`true`** and **`false`**.

For the long double type, the following interface must be provided:

- The method **`CORBA::LongDouble.new`** creates a new CORBA::LongDouble instance from a Float, a String, or a BigDecimal with optional precision specified.
- The method **`to_f`** of a long double number converts it into a Float. For each Float f, CORBA::LongDouble(f).to_f==f.
- The CORBA::LongDouble instance has an internal representation that is capable of storing IEEE-754 compliant values, with sign, 31 bits of mantissa (offset 16383), and 112 bits of fractional mantissa. If numeric operations are provided, they offer the precision resulting from this specification.

## 7.9 Mapping for Enums

An OMG IDL enum maps to a series of Ruby integer constants as shown in the example below. Furthermore, a Ruby class is defined to carry the enum name and typecode information.

```
// IDL
enum test_enum
{
  TE_ZEROTH,
  TE_FIRST,
  TE_SECOND,
  TE_THIRD,
  TE_FOURTH
};
```

```
# Ruby
class Test_enum
  ...
end # enum Test_enum
TE_ZEROTH = 0
TE_FIRST = 1
TE_SECOND = 2
TE_THIRD = 3
TE_FOURTH = 4
```

## 7.10 Mapping for String Types

The OMG IDL string type, whether bounded or unbounded, is mapped to String. Ruby does not have a class that would match IDL-bounded strings. As a result, the programmer is responsible for enforcing the bound of bounded strings at run time.

The Ruby mapping will not implement functionality to prevent assignment of a string value to a bounded string type if the string value exceeds the bound. It will however detect attempts to pass a string value that exceeds the bound as a parameter across an interface. Such a condition will be signaled by a `MARSHAL` system exception.

## 7.11 Mapping for Wide String Types

The OMG IDL wide string type, whether bounded or unbounded, is mapped to a Ruby Array where the array elements are restricted to Fixnum instances within the 0...0xFFFF range. The Ruby mapping will not implement functionality to check insertion/addition of invalid element types to such an array. It will however detect attempts to pass arrays containing invalid element types as a parameter across an interface. Such a condition will be signaled by a `MARSHAL` system exception.

The Ruby mapping will allow passing String objects as values for IN/INOUT arguments. These values will be implicitly converted to arrays of Fixnum values.

Ruby does not have a class that would match IDL-bounded wide strings. As a result, the programmer is responsible for enforcing the bound of bounded wide string mapped arrays at run time. The Ruby mapping will not implement functionality to prevent extending a wide string mapped array beyond its bound. It will however detect attempts to pass a wide string value that exceeds the bound as a parameter across an interface. Such a condition will be signaled by a `MARSHAL` system exception.

## 7.12 Mapping for Struct Types

An OMG IDL struct maps to a Ruby class, with each OMG IDL struct member mapped to a corresponding instance variable of the Ruby class. Ruby style accessor methods with the names of the IDL struct members are provided for read and modify access to the member values. The constructor for the class expects zero or more values to initialize the instance variables in the same order as the corresponding IDL structure members. Values that are not provided in the constructor call result in initialization of the instance variable with the value `nil`.

For example, the IDL definition

```
// IDL
struct point {
  long x;
  long y;
};
```

is translated in Ruby as:

```
# Ruby
class Point
  ...
  attr_accessor :x
  attr_accessor :y
  def initialize(*param_)
    @x,
    @y = param_
  end
end
```

and can be used in Ruby code as:

```
# Ruby
pt = Point.new(10, 15)
# print coordinate
puts "Point = {#{pt.x}, #{pt.y}}"
# change coordinate
pt.x = pt.y * 2
pt.y = 1
```

The Ruby mapping will provide type information concerning the struct and its members accessible through the Ruby class for use in type_checking code when the struct class is used as a parameter to be passed across an interface.

## 7.13 Mapping for Fixed Types

This type is not supported in this version of the Language mapping.

## 7.14 Mapping for Union Types

Unions map to Ruby classes with Ruby style accessor methods for the union members and discriminant. Both read and modify accessor methods are provided.

The union class has two instance variables, one for the discriminator value and one for the active member value. The constructor initializes the union class to a nil state; i.e., the discriminator and initial member value of the union are initialized as **`nil`**. It is an error for a compliant Ruby application to use a union class instance before setting its value explicitly.

The union discriminant accessor and modifier methods have the name `_disc`. The _disc discriminator modifier can only be used to set the discriminant to a value within the same union member. Attempting to set the value outside the active union member will result in a **`BAD_PARAM`** system exception.

The discriminator value for a default case is represented by the Ruby symbol value **`:default`**. This value can be used to set the discriminant of a union with an implicit default member. A union has an implicit default member if it does not have a default case and not all permissible values of the union discriminant are listed. The Ruby mapping implementation provides the method **`_is_at_default?`** to test if the default member is active.

Setting the union value through a modifier method automatically sets the discriminant. If a modifier for a union member with multiple legal discriminant values is used to set the value of the discriminant, the union implementation is free to set the discriminant to any one of the legal values for that member. The actual discriminant value chosen under these circumstances is implementation-dependent. The discriminant accessor can be used to set another value for the discriminant as long as this value belongs to the same union member.

Accessor methods for union members provide semantics similar to that of struct data members.

The following example helps illustrate the mapping for union types.

```
// IDL
union U1 switch(long)
{
 case 0:   long m_l;
 case 1:
 case 2:   string m_str;
 default:  boolean m_bool;
};


# Ruby
class U1

  def _disc; ... end
  def _disc=(val); ... end

  def _is_at_default?; ... end

  def m_l; ... end
  def m_l=(val); ... end
  def m_str; ... end
  def m_str=(val); ... end
  def m_bool; ... end
  def m_bool=(val); ... end

end #of union U1
```

## 7.15 Mapping for Sequence Types

An IDL sequence is mapped to a Ruby Array where the element values should be restricted to the type specified in the IDL declaration. The Ruby mapping will not implement functionality to check insertion/addition of invalid element types to such an array. It will however detect attempts to pass arrays containing invalid element types as a parameter across an interface. Such a condition will be signaled by a **`MARSHAL`** system exception.

The Ruby mapping will allow the following implicit conversions for Ruby objects passed as values for Sequence type IN/INOUT arguments:

- Where a sequence of char or octet is expected, a Ruby String object is allowed.
- Where a sequence is expected, a Ruby object responding to **`#to_ary`** is allowed.

Ruby does not have a class that would match IDL-bounded sequences. As a result, the programmer is responsible for enforcing the bound of bounded sequence mapped arrays at run time. The Ruby mapping will not implement functionality to prevent extending a bounded sequence mapped array beyond its bound. It will however detect attempts to pass a sequence value that exceeds the bound as a parameter across an interface. Such a condition will be signaled by a **`MARSHAL`** system exception.

## 7.16 Mapping for Array Types

An IDL array is mapped to a Ruby Array where the element values should be restricted to the type specified in the IDL declaration. The Ruby mapping will not implement functionality to check insertion/addition of invalid element types to such an array. It will however detect attempts to pass arrays containing invalid element types as a parameter across an interface. Such a condition will be signaled by a **`MARSHAL`** system exception.

Ruby does not have a class that would match the fixed bound nature of IDL arrays. As a result, the programmer is responsible for enforcing the bound of IDL-array mapped arrays at run time. The Ruby mapping will not implement functionality to check IDL-array mapped array bounds compliance. It will however detect attempts to pass an array value that does not comply to the IDL bounds specification as a parameter across an interface. Such a condition will be signaled by a **`MARSHAL`** system exception.

In case of multi dimensional arrays the mapping specifies nested Ruby Array instances; i.e., the elements of the ‘outer’ array instance(s) are supposed to be Array instances. For example the IDL definition

```
// IDL
typedef long Long_Matrix[3][3];
```

is mapped to a Ruby Array value like

```
# Ruby
[[1,2,3],[4,5,6],[7,8,9]]
```

## 7.17 Mapping for Typedefs

A typedef creates an alias for a type. The mapping implementation will record these type aliases and their relation to structured type members and/or operation parameters for use in type checking code when passing values across interfaces.

The original type for the typedef determines the mapping to the actual Ruby type used in the implementation. For example the IDL definition

```
// IDL
typedef string<30>  TNameString;
typedef sequence<TNameString> TNames;
```

is mapped on a Ruby Array where the elements are string instances like

```
["one", "two", "three"]
```

## 7.18 Mapping for the Any Type

Because of the dynamic typing in Ruby, there is no need for a strictly type-safe mapping of the any type as in the C or C++ mappings. Instead, all that needs to be available at run-time is the value and the type code corresponding to the type of the value.

### 7.18.1 Mapping Ruby Values to Any

Values of IDL generated types (interfaces, structured types, aliases) are always associated with a type code (see also "Mapping for Typecodes" on page 23) that provides the mapping implementation of the required information.

Values of basic data types do not require an associated typecode as their mapping is implicit, based on the rules described in "Mapping for Basic Data Types" on page 9.

There are however exceptions where an additional typecode 'direction' would be necessary. This applies most particularly to the numeric basic types.

As these types have explicit subtypes in IDL that map onto a single type in Ruby, it is often necessary to direct the Any mapping to the actual type code to use when using Ruby values to pass as Any across interfaces. Without distinct direction the value would be mapped to the 'largest,' signed, subtype by default that might not match the expectations. The Ruby mapping provides support for this in the form of the `CORBA::Any` class.

The `CORBA::Any.to_any()` method can be used to wrap Ruby values requiring explicit typecode direction as an Any as shown in the following example:

```
# Ruby
# Ruby Integer value which by default is mapped to IDL type 'long'
int_val = 123

# creat CORBA::Any with specific type code direction
any_val = CORBA::Any.to_any(int_val, CORBA._tc_ushort)

puts int_val==any_val._value  # prints 'true'

puts CORBA._tc_ushort.equal?(any_val._tc) # prints 'true'
```

The type code direction requirement also applies to integer constants generated by the IDL compiler for IDL enum types (see "Mapping for Enums" on page 10) passed as Any. These values should be wrapped as follows:

```
// IDL
module Test {
 enum test_enum
 {
  TE_ZEROTH,
  TE_FIRST,
  TE_SECOND,
  TE_THIRD,
  TE_FOURTH
 };
};
```

```
# Ruby
enum_any = CORBA::Any.to_any(Test::TE_FIRST, Test::Test_enum._tc)
```

### 7.18.2 Mapping Any values to Ruby

As CORBA Any values contain all required type code information the Ruby mapping implementation is capable of automatically mapping incoming Any values to their corresponding Ruby types. This goes for basic types (using the mapping described in “Mapping for Basic Data Types” on page 9 ) as well as for all IDL defined types.

Object references mapped from Any values will also be narrowed automatically if the Any contains a specific interface type.

## 7.19 Mapping for Valuetypes

An IDL valuetype is mapped on a Ruby class (using the naming conventions described in sub clause 7.2 that contains public definitions of the types, constants, operations, attributes, and state members defined as part of the valuetype.

The CORBA::ValueBase type is mapped on the Ruby mixin module CORBA::ValueBase that is included in every concrete valuetype class.

All operations defined as part of the valuetype (or supported interfaces) are declared with a default implementation that throws a runtime exception stating the operation is unimplemented.

When a valuetype defines (or supports) operations the application developer should override the default implementation. In Ruby this can be done either in a class derived directly or indirectly from the generated valuetype class (in case multiple valuetype implementations are possible) or by “reopening” the generated valuetype class.

The valuetype null value is mapped to the Ruby **nil** value.

### 7.19.1 Valuetype data (state) members

The Ruby mapping for valuetype data members follows the same rules as the Ruby mapping for struct members. Public state members are mapped to public accessors Ruby valuetype (base) class, and private state members are mapped to protected accessors (so that derived concrete classes may access them).

For example

```
// IDL
typedef octet Bytes{64};
struct S { ... };
interface A { ... };

valuetype Val {
   public Val t;
   private long v;
   public Bytes w;
   public string x;
   private S y;
   private A z;
 };
```

could be implemented in Ruby as

```
# Ruby
class S
   ...
end

module A
   ...
end

class Val
   include CORBA::ValueBase
   ...
   attr_accessor  :t
   attr_accessor  :w
   attr_accessor  :x

protected
   attr_accessor  :v
   attr_accessor  :y
   attr_accessor  :z
   ...
end
```

### 7.19.2 Valuetype Operations

All operations declared on a valuetype are mapped on public methods with a default implementation that throws a runtime exception stating the operation is unimplemented.

When a valuetype declares operations the application developer should override the default implementation. In Ruby this can be done either in a class derived directly or indirectly from the generated valuetype class (in case multiple valuetype implementations are possible) or by "reopening" the generated valuetype class.

For example

```
// IDL
valuetype BaseNode {
   short op1();
   long op2(in BaseNode node);

   public string name;
   private long id;
};
```

could be implemented in Ruby as

```
# Ruby
class BaseNode < CORBA::Valuebase
   include CORBA:: ValueBase
   ...
   def op1
      raise RuntimeError ...
```

```
    end
    def op2 (node)
      raise RuntimeError ...
    end

    attr_accessor :name
protected
    attr_accessor :id
end
```

### 7.19.3 Value Boxes

A boxed type IDL valuetype declaration is mapped on a Ruby class that contains a single, public, Ruby accessor implementation for a standard member of the boxed type named value.

In essence this class provides a very simple container for the boxed type allowing null values to be passed as interface arguments for these types.

To fulfill the ValueBase interface all value box classes include the Ruby mixin module CORBA::Portable::BoxedValueBase that is derived from the Ruby CORBA::ValueBase module.

For example

```
// IDL
valuetype string BoxedString;
```

could be implemented in Ruby as

```
# Ruby
class BoxedString
    include CORBA:: ValueBase
    ...
    attr_accessor  :value
end
```

When declaring value boxes as argument or return types for interface operations (or attribute accessor and modifier methods) the Ruby implementation provides implicit conversion of the underlying boxed type to (for in arguments) or from (for out arguments and return values) the value box type:

```
// IDL
valuetype string BoxedString;
valuetype long BoxedLong;

interface Foo {
    void echo(in BoxedString txt);
    attribute Boxed Long count;
};
```

could be implemented in Ruby as

```
# Ruby
...
my_foo = Foo._narrow(some_obj_ref)
```

```
# passing underlying boxed type works fine
my_foo.echo ('Hello too')
my_foo.count = 1
#passing null values too
my_foo.echo (nil)
my_fo.count = nil
# passing actual value box is also possible
bs = BoxedString.new
bs.value = 'Hello'
my_foo.echo (bs)

# return values and out args are always returned as underlying boxed type
# (or nil for null values)

# returns an integer value (NOT BoxedLong) or nil
the_count = my_foo.count
```

### 7.19.4 Abstract Valuetypes

An IDL abstract valuetype is mapped on a Ruby module (using the naming conventions described in sub clause 7.2 that contains public definitions of the types, constants, operations, and attributes defined as part of the valuetype.

Abstract valuetypes cannot be instantiated and the mapping on a Ruby module ensures that (as with the interface mapping: sub clause 7.4).

As an abstract valuetype has no state members that may need marshaling/demarshaling and cannot be instantiated there are no factory classes generated for abstract valuetypes.

### 7.19.5 Valuetype Inheritance

For an IDL valuetype derived from other valuetypes or that supports interface types the following applies:

- Concrete and abstract value base classes are inherited.
- Supported interfaces are inherited with respect to ancestor type information (is_a semantics) and operations and attributes interfaces; not inherited are object reference semantics.

Valuetype classes inheriting supported interfaces do not inherit object reference semantics like narrowing methods. Also calling any method mapped from an interface inherited operation or attribute will result in a **CORBA::NO_IMPLEMENT** exception being thrown by default.

Applications can provide overriden implementations by either deriving an application specific valuetype class or by "reopening" the generated class.

In case of a valuetype supporting an interface the IDL compiler will also generate a servant skeleton in the POA namespace (see sub clause 7.25) with the same name as the valuetype class (including scoping). The servant skeleton class inherits from the valuetype class and from the servant classes for the supported interfaces.

For example

```
// IDL
interface A {
    void op();
};
```

```
valuetype B supports A {
    public short data;
};
```

could be implemented in Ruby as

```
# Ruby
# Client side mapping

module A
    ...
end

Class B
    include CORBA::ValueBase
    ...
    def op
        ...
end

# Server side mapping

module POA
    class A
        ...
    end

    Class B
        ...
        include POA::A
        include : :B
    end
end
```

### 7.19.6 Valuetype Factories

Valuetype factories are the means by which the ORB is able to instantiate new instances of (possible user derived) concrete valuetype classes at demarshaling time.

For every concrete valuetype there is an additional factory class generated. The name of the class is formed by appending the suffix "**Factory**" to the valuetype name. The base class for all factory classes is CORBA::ValueFactory.

The generated factory class implements a default factory method name "_create_default" returning a newly created instance (with default, empty, initialization) of the generated valuetype class. This method is called by the ORB on registered valuetype factories when creating new valuetype instances for the purpose of demarshaling.

Additionally for each factory method defined for a valuetype a default method implementation will be generated (having the name and arguments as specified for the IDL defined factory method) as part of the factory class. The default implementation of these factory methods will throw a runtime exception stating the operation is unimplemented.

Application derived implementations of these factory methods should return a valuetype instance of the corresponding (possibly application derived) valuetype class.

For example:

```
// IDL
valuetype Coord {
    public double x:
    public double y:
    factory setup(in double x_org, in double y_org);
};
```

could be implemented in Ruby as:

```
# Ruby
class Coord
   include CORBA::ValueBase
   ...
end

class CoordFactory
   ...
   def _create_default
     Coord.new
   end

   def setup (x_org, y_org)
     raise RuntimeError...
   end
   ...
end
```

Applications can derive and implement customized value factories by using the generated value factory classes as base class.

To enable the ORB to make use of a value factory for a certain valuetype the application must register an instance of a value factory class through the ORB::register_value_factory class.

For simple valuetypes having only state members (no operations, no attributes, and no type specific factory methods), the generated factory class is normally sufficient and needs no derivatives.

The application however, still needs to explicitly register a value factory instance with the ORB.

Valueboxes constitute a special case of state-only valuetypes and as such never require derived value factories or even factory registration.

Default value factory instances for every IDL defined valuebox type will be implicitly registered with the ORB.

### 7.19.7 Custom Marshaling

Valuetypes declared to have custom marshaling follow the same Ruby mapping rules as for normal (non-custom declared) valuetypes except for the following:

- “custom” valuetype classes do **not** get marshaling and demarshaling code generated but instead implement the Ruby mapping of the interface of CORBA::CustomMarshal abstract valuetype that declares the marshal and unmarshal methods.

The application should provide implementations for the **marshal** and **unmarshal** methods of each custom valuetype.

The CORBA::DataOutputStream and CORBA::DataInputStream arguments of these methods are mapped on Ruby classes providing Ruby mappings for the IDL valuetype operation declarations.

For example

```
// IDL
custom valuetype CustomFoo {
    public string name;
    private short id;
};
```

could be implemented in Ruby as

```
# Ruby

class CustomFoo
    ...
    attr_accessor :name
protected
    attr_accessor :id
public
    def marshal (os)
    ...
    end

    def unmarshal (is)
    ...
    end
    ...
end

class CustomFooImpl < CustomFoo
    ...
    def marshal (os)
      os.write_string(self.name)
      os.write_short(self.id)
    end
    def unmarshal (is)
      self.name = is.read_string
      self.id = is.read_short
    end
    ...
end
```

## 7.20 Mapping for Typecodes

A TypeCode represents OMG IDL type information.

The TypeCode interface is defined in IDL in sub clause 8.11 of the CORBA v3.1 specification. TypeCodes are represented by pseudo object references in the Ruby mapping.

All predefined TypeCode constants, as defined in the core specification, are available through accessor methods of the CORBA namespace as **`CORBA._tc_{type}`** where {type} refers to the typenames of the types represented by the TypeCodes such as **null** (CORBA._tc_null), **long** (CORBA._tc_long), etc.

For each basic and defined OMG IDL type, the Ruby mapping implementation provides access to a TypeCode pseudo object reference through an accessor method of the module or class representing the type like **{type}._tc**. TypeCode pseudo object references may be used to set types for Any values, as arguments for equal, and so on.

The Ruby mapping implementation provides a full range of derived TypeCode classes defined within the **`CORBA::TypeCode`** namespace (actually the `CORBA::TypeCode` class) that can be used to construct TypeCode references for user defined types.

The following code for example creates a TypeCode reference for the **`CosNaming::NamingContext`** interface:

```
CORBA::TypeCode::ObjectRef.new(
   "IDL:omg.org/CosNaming/NamingContext:1.0","NamingContext")
```

And this code creates a TypeCode reference for a typedef-fed string type defined in the scope of a module (or interface) **Test**:

```
CORBA::TypeCode::Alias.new(
   'IDL:Test/TString:1.0','TString', self,CORBA::_tc_string)
```

Since the availability of the TypeCode derived classes provides all required support the **`create_XXX_tc()`** methods of the ORB interface are not implemented in the Ruby mapping.

## 7.21 Mapping for Abstract Interfaces

The Ruby mapping for abstract interfaces is identical to that of regular interfaces except for the following:

- Ruby modules generated for abstract interfaces get the repository id of the CORBA::AbstractInterface interface added to the list of supported interfaces.
- The typecode for the generated Ruby type is an AbstractInterface typecode instead of an ObjectRef typecode.

### 7.21.1 Argument passing and return values

On the client side valuetype instances supporting an abstract interface and object references supporting the same abstract interface are interchangeable as **in** arguments to any IDL declared interface operation (or attribute modifier) specifying that abstract interface as argument type.

**Out** arguments and **return** values will be returned as either valuetype instances or object references according to the type of the object provided on the opposite side.

For server side mappings the reverse applies.

For example:

```
// IDL
abstract interface Base {
    ...
};

interface Ops : Base {
    ...
};

valuetype Node : supports Base {
    ...
};

interface Foo {
    void pass_base)in Base b)
    Base get_base ();
};
```

could be implemented in Ruby as

```
# Ruby

module Base
   ...
end

module OPs
   ...
end

class Node
   ...
end

module Foo
   ...
   def pass_base (b)
      ...
   end
   def get_base ()
      ...
   end
   ...
end

...

# 'my_node' is Node valuetype instance
# 'my_ops' is object reference narrowed to Ops

my_foo = Foo._narrow (an_object_ref)

if must_pass_object == true
   my_foo.pass_base (my_ops
```

```
else
   my_foo.pass_base (my_mode)
end

...

)retval = by_foo.get_base()

unless retval.nil?  || retval.is_a? (CORBA::ValueBase)
   #handle valuetype
   ...
else
   #handle object reference
   ...
end
```

## 7.22 Mapping for Exception Types

An IDL exception is translated into a Ruby class derived from `CORBA::UserException`. System exceptions are derived from `CORBA::SystemException`. Both base classes are derived from `CORBA::Exception` that in turn is derived from the Ruby exception class `StandardError`. The parameters of the exception are mapped in the same way as the fields of a struct definition. When raising an exception, a new instance of the class is created. The constructor expects the exception parameters. For example, the definition

```
// IDL
module My
{
 interface Intf
 {
  exception PermissionDenied { string details; };
  Intf create(in string name) raises (PermissionDenied);
 };
};
```

is mapped like

```
# Ruby
module My
  ...
  module Intf
    ...
    class PermissionDenied < CORBA::UserException
      ...
      attr_accessor :details
      def initialize(*_param)
        @details = _param
      end
    end
    ...
    def create(name)
      ...
    end
```

```
    ...
  end
end
```

and can be used as

```
# Ruby

# catch exception (possibly) raised by servant
begin
  new_intf = my_intf.create("a_name")
rescue My::Intf::PermissionDenied => exc
  puts exc.to_s
  puts exc.details
end

# raise exception
raise My::Intf::PermissionDenied.new('just a test')
```

## 7.23 Mapping for Operations and Attributes

A CORBA object reference is represented by a Ruby object at run-time. This object provides all the operations that are available on the interface of the object. The nil object is represented by **nil**.

If an operation expects parameters of the IDL Object type, any Ruby object representing an object reference might be passed as actual argument.

Operations of an interface map to methods available on the Ruby objects. Parameters with an attribute of in or inout are passed from left to right to the method, skipping out parameters. The return value of a method depends on the number of out parameters and the return type. If the operation returns a value, this value forms the first result value. All inout or out parameters form consecutive result values returned in the order in which the respective parameters were defined in IDL. The method result depends then on the number of result values, as follows:

- If there is no result value, the method returns **nil**.
- If there is exactly one result value, it is returned as a single value.
- If there is more than one result value, all of them are packed into an array, and this array is returned.

Assuming the IDL definition

```
// IDL
interface Intf
{
  oneway void stop();
  bool more_data();
  void get_data(out string name,out long age);
};
```

a client could write

```
# Ruby
names = {}
while my_Intf.more_data()
   name,age = my_Intf.get_data()
   names[name]=age
end
my_Intf.stop()
```

If an interface defines an attribute ‘firstname’, the attribute is mapped into a CORBA operation request name **`_get_firstname`**, as defined. If the attribute is not readonly, there is an additional operation name **`_set_firstname`**, as defined in Clause 7: OMG IDL Syntax and Semantics of CORBA, v3.1, Part I (formal/2008-01-04).

The Ruby mapping implementation however provides runtime accessor methods for objects implementing IDL interfaces that more naturally match the ‘attribute’ style.

For the read operation on the attribute the mapping provides an accessor method with the name of the attribute without any decoration. For the write operation (if allowed) a Ruby assignment style accessor method is provided (also with the name of the attribute). This allows a client to use a mapping for IDL like

```
// IDL
interface Intf
{
  attribute string firstname;
};
```

in the following manner

```
# Ruby
# get firstname
my_name = my_intf.firstname
# set firstname
my_intf.firstname = 'Martin'
```

## 7.24 The Dynamic Invocation Interface

The operations **`_request`** and **`_create_request`** of **`CORBA::Object`** instances return a **`CORBA::Request`** object that can be used to invoke an operation on the object reference for which the request was created in several ways.

The Ruby mapping implements the following standard CORBA methods on the Request object:

- target
- operation
- arguments
- exceptions
- add_exception
- add_in_arg

- add_out_arg
- add_inout_arg
- set_return_type
- return_value
- invoke
- send_oneway
- send_deferred
- get_response
- poll_response

## 7.25 Servant Implementation Mapping

Central to the Object Request Broker architecture is the object adapter, which communicates the requests to the servant implementation. CORBA explicitly allows for multiple object adapters, including non-standardized ones. The only object adapter that is standardized for CORBA 2.3 is the Portable Object Adapter.

This specification only defines a servant implementation mapping for the POA.

### 7.25.1 Skeleton-Based Implementation

This specification defines an inheritance-based mapping for implementing servants. There is no Ruby imposed reason to use a delegation-based approach but if needed this could be implemented on top of the inheritance-based approach.

For the POA all modules/interfaces generated from IDL definitions are contained in a top_level `POA` namespace. Following the name mapping scheme for Ruby, the Ruby class corresponding to an IDL interface can be used as a base class for the servant implementation class. For example, the following interface

```
// IDL
module Mod
{
 interface Intf
 {
  void foo();
 };
};
```

could be implemented in Ruby as

```
# Ruby
class MyIntf < POA::Mod::Intf
    def foo()
        # do something ...
    end
end
```

As Ruby only implements single inheritance a servant implementation class can only be derived from a single IDL generated skeleton class.

To accommodate implementing multiple IDL interfaces in a single servant implementation class the mapping implementation supports including IDL generated skeleton classes into the servant class as if they were Mixin modules. Using this mechanism the following IDL interfaces

```
// IDL
module Mod
{
 interface Intf
 {
  void foo();
 };

 interface Intf_2
 {
  void foo_2();
 };
};
```

could be implemented in Ruby either as

```
# Ruby
class MyIntfs < POA::Mod::Intf
    include POA::Mod::Intf_2
    def foo()
        # do something ...
    end
    def foo_2()
        # do something else ...
    end
end
```

or as

```
# Ruby
class MyIntfs < PortableServer::Servant
    include POA::Mod::Intf
    include POA::Mod::Intf_2
    def foo()
        # do something ...
    end
    def foo_2()
        # do something else ...
    end
end
```

If a servant implementation method requires returning one or more **out** parameters with or without a result value, these should always be returned as an array. If no result is expected, the object adapter will ignore any result returned from the method implementation.

The implementation method result depends on the number of result values, as follows:

- If no result value is expected, the method returns **nil**.
- If a result value is expected but **no** out parameters, a single value is returned.
- If one or more **out** parameters are expected with or without (void) a result value, they are returned as an array (of length 1 or more). If a result value is required, this will be the first value in the array.

The skeleton class (POA::Mod::Intf in the example) supports the following operations:

- **`_default_POA()`** returns the POA reference that manages that object. It can be overridden by implementations to indicate they are managed by a different POA. The standard implementation returns the same reference as **ORB.resolve_initial_references("RootPOA")**, using the default ORB.
- **_this()** returns the reference to the object that a servant incarnates during a specific call. This works even if the servant incarnates multiple objects. Outside the context of an operation invocation, it can be used to initiate the implicit activation, if the POA supports implicit activation. In any case it should return an object that supports the operations of the corresponding IDL interface.

The base class for all skeleton classes is the class **CORBA::PortableServer::Servant**.

### 7.25.2 The Dynamic Skeleton Interface

An implementation class is declared as dynamic by inheriting from **`CORBA::PortableServer::DynamicImplementation`**. Derived classes need to implement the method **invoke**, which is called whenever a request is received. The **`invoke`** method is passed a request object.

The request object provides access to the request parameters. The DSI servant implementation must first provide the request object with precise descriptions of the expected arguments before being able to access the argument values. The request 'description' includes the following information:

- The typecode for the result value in case of a two way invocation request (this includes **void** results). Without a result type the request is assumed to be oneway and no return values will be processed.
- Argument name (can be nil), argument mode (IN/OUT/INOUT), and typecode for each formal argument.

The **`invoke`** method returns either with a result according to the same specifications as for static skeletons, or by raising an appropriate exception.

The implementation class must also implement the method **`_primary_interface`**, which must return a non-nil repository id representing the most derived interface for a given object id.

The Ruby mapping implementation defines a default implementation of the **`#invoke`** method as

```
class PortableServer::DynamicImplementation
    def invoke(request)
        if self.class.const_defined?("OPTABLE") & self.class::OPT-
ABLE.has_key?(request.operation)
            request.describe(self.class::OPTABLE[request.operation])
            return self.__send__(request.operation, *request.arguments) {
request }
        else
            return self.__send__(request.operation) { request }
        end
    end
end
```

The default method implementation expects a servant implementation to define a class constant named **OPTABLE** containing a Hash of request descriptions as described above as values with the operation name as key.

The method class could look like

```
# Ruby
class DynSkel < PortableServer::DynamicImplementation
    OPTABLE = {
        'echo' => { :result_type => CORBA::_tc_string,
                    :arg_list => [
                    ['parm1', CORBA::ARG_IN, CORBA::_tc_string],
                    ['parm2', CORBA::ARG_OUT, CORBA::_tc_long] ] }
    }
    ...
    def echo(str)
        [message.to_s, message.to_s.size]
    end

    def shutdown
        ... # handle shutdown request
    end
    ...
    def _primary_interface(oid, poa)
        return 'IDL:Foo:1.0'
    end
    ...
end
```

Of course a servant implementation could overload the default **#invoke** implementation and provide a different dispatching mechanism for requests.

## 7.26 Ruby Definitions for CORBA

This sub clause provides a partial set of Ruby definitions for the CORBA module. The definitions appear within the Ruby namespace named CORBA.

```
# Ruby
module CORBA
  ...
end
```

Any implementations shown here are merely sample implementations: they are not the required definitions for these types. Furthermore, in some cases these types do not define the complete interfaces of their IDL counterparts; if any type is missing one or more operations, those operations are assumed to follow normal Ruby mapping rules for their signatures, parameter passing rules, etc.

### 7.26.1 CORBA namespace

```
module CORBA
  def CORBA.ORB_init(*args); ... end
  def CORBA.is_nil(obj); ... end
end
```

### 7.26.2 Exception classes

```
module CORBA
  class Exception < StandardError
  end
  class UserException < CORBA::Exception
  end
  class SystemException < CORBA::Exception
    def initialize(reason="", minor=0, completed=nil); ... end
    attr_accessor :reason, :minor, :completed
  end
end
```

### 7.26.3 ORB class

```
module CORBA
  module ORB
    def object_to_string(obj); ... end
    def string_to_object(str); ... end
    def create_list(count); ... end
    def create_operation_list(oper); ... end
    def get_default_context(); ... end
    def send_multiple_request_oneway(req); ... end
    def send_multiple_request_deferred(req); ... end
    def poll_next_response(); ... end
    def get_next_response(); ... end
    def get_service_information(service_type); ... end
    def list_initial_services(); ... end
    def resolve_initial_references(identifier); ... end
    def register_initial_reference(identifier, obj); ... end
    def create_struct_tc(id, name, members); ... end
    def create_union_tc(id, name, discriminator_type, members); ... end
    def create_enum_tc(id, name, members); ... end
    def create_alias_tc(id, name, original_type); ... end
    def create_exception_tc(id, name, members); ... end
    def create_interface_tc(id, name); ... end
    def create_string_tc(bound); ... end
    def create_wstring_tc(bound); ... end
    def create_fixed_tc(digits, scale); ... end
    def create_sequence_tc(bound, element_type); ... end
    def create_array_tc(length, element_type); ... end
    def create_value_tc(id, name, type_modifier, concrete_base, members); ...
end
    def create_value_box_tc (id, name, boxed_type); ... end
```

```
    def create_native_tc(id, name); ... end
    def create_recursive_tc(id); ... end
    def create_abstract_interface_tc(id, name); ... end
    def work_pending(); ... end
    def perform_work(); ... end
    def run(); ... end
    def shutdown(wait_for_completion); ... end
    def destroy(); ... end
    def create_policy(type, val); ... end
    def register_value_factory(id, factory); ... end
    def unregister_value_factory(id); ... end
    def lookup_value_factory(id); ... end
  end # ORB
end
```

### 7.26.4 Object class

```
module CORBA
  module Object
    def _get_interface(); ... end
    def _is_nil?(); ... end
    def _duplicate(); ... end
    def _release(); ... end
    def _is_a?(logical_type_id); ... end
    def _non_existent?(); ... end
    def _is_equivalent?(other_object); ... end
    def _hash(maximum); ... end
    def _repository_id(); ... end
    def _interface_repository_id(); ... end
    def _get_policy(policy_type); ... end
    def _set_policy_overrides(policies, set_add ); ... end
    def _get_orb(); ... end
    def _create(opname); ... end
    def _create_request(opname, arglist, result, exceptions=nil); ... end
    def _free_ref(); ... end
  end # Object
end # CORBA
```

### 7.26.5 Any class

```
module CORBA
  class Any
    def _tc(); ... end
    def _value(); ... end
    def Any.to_any(value, tc); ... end
    def Any.typecode_for_any(any); ... end
    def Any.value_for_any(any); ... end
  end
end
```

### 7.26.6 Request class

```
module CORBA
  class Request
    def target(); ... end
    def operation(); ... end
    def arguments(); ... end
    def exceptions(); ... end
    def exceptions=(exception_typecodes); ... end
    def add_in_arg(arg_tc, arg_val, arg_name=nil); ... end
    def add_out_arg(arg_tc, arg_name=nil); ... end
    def add_inout_arg(arg_tc, arg_val, arg_name=nil);; ... end
    def set_return_type(return_tc); ... end
    def return_value(); ... end
    def invoke(arg_list=nil, return_type=nil, exceptions=nil); ... end
    def send_oneway(arg_list=nil); ... end
    def send_deferred(); ... end
    def get_response(); ... end
    def poll_response(); ... end
  end
end
```

### 7.26.7 TypeCode class

```
module CORBA
  class TypeCode
    def kind; ... end
    def get_compact_typecode; ... end
    def equal?(tc); ... end
    def equivalent?(tc); ... end
    def id; ... end
    def name; ... end
    def member_count; ... end
    def member_name(index); ... end
    def member_type(index); ... end
    def member_label(index); ... end
    def discriminator_type; ... end
    def default_index; ... end
    def length; ... end
    def content_type; ... end
    def fixed_digits; ... end
    def fixed_scale; ... end
    def member_visibility; ... end
    def type_modifier; ... end
    def concrete_base_type; ... end
  end
end
```

## 7.27 Not implemented Deprecated Definitions

The Ruby mapping does not implement the following deprecated IDL definitions:

- Context
- Environment and non-native exceptions
- Principal