IDL to PHP™ LanguageMapping Specification

October 2003
Version 1.0
formal/03-10-31

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

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

LICENSES

The author listed above has granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. The copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.
Subject to all of the terms and conditions below, the owner of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you break any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

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

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE AUTHOR LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.
IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR THE AUTHOR LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owner is indicated above and may be contacted through the Object Management Group, 250 First Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

COMPLIANCE

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents & Specifications, Report a Bug/Issue.

Contents

Contents

Preface

About This Document

Under the terms of the collaboration between OMG and The Open Group, this document is a candidate for adoption by The Open Group, as an Open Group Technical Standard. The collaboration between OMG and The Open Group ensures joint review and cohesive support for emerging object-based specifications.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based. More information is available at http://www.omg.org/.

The Open Group

The Open Group, a vendor and technology-neutral consortium, is committed to delivering greater business efficiency by bringing together buyers and suppliers of information technology to lower the time, cost, and risks associated with integrating new technology across the enterprise.

The mission of The Open Group is to drive the creation of boundaryless information flow achieved by:

- Working with customers to capture, understand and address current and emerging requirements, establish policies, and share best practices;
- Working with suppliers, consortia and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate specifications and open source technologies;
- Offering a comprehensive set of services to enhance the operational efficiency of consortia; and
- Developing and operating the industry's premier certification service and encouraging procurement of certified products.

The Open Group has over 15 years experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification. The Open Group portfolio of test suites includes tests for CORBA, the Single UNIX Specification, CDE, Motif, Linux, LDAP, POSIX.1, POSIX.2, POSIX Realtime, Sockets, UNIX, XPG4, XNFS, XTI, and X11. The Open Group test tools are essential for proper development and maintenance of standards-based products, ensuring conformance of products to industrystandard APIs, applications portability, and interoperability. In-depth testing identifies defects at the earliest possible point in the development cycle, saving costs in

development and quality assurance.

More information is available at http://www.opengroup.org/ .

About CORBA Language Mapping Specifications

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

Alignment with CORBA

This language mapping is aligned with CORBA, v2.4.

Associated Documents

The CORBA documentation set includes the following books:

- Object Management Architecture Guide defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBAservices: Common Object Services Specification* contains specifications for the Object Services.
- *CORBAfacilities: Common Facilities Architecture* contains the architecture for Common Facilities.

OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote.

You can download the OMG formal documents free-of-charge from our web site in Post-Script and PDF format. Please note the OMG address and telephone numbers below:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Definition of CORBA Compliance

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

Interoperability and Interworking are separate compliance points. For detailed information about Interworking compliance, refer to the *CORBA/IIOP Specification (The Common Object Request Broker: Architecture and Specification)*, *Interworking Architecture* chapter.

As described in the *OMA Guide*, the OMG's Core Object Model consists of a core and components. Likewise, the body of *CORBA* specifications is divided into core and

component-like specifications. The *CORBA* specifications are divided into these volumes:

1. The *CORBA/IIOP Specification (The Common Object Request Broker: Architecture and Specification)*, which includes the following chapters:
   - **CORBA Core**, as specified in Chapters 1-11
   - **CORBA Interoperability**, as specified in Chapters 12-16
   - **CORBA Interworking**, as specified in Chapters 17-21
   - **CORBA Quality of Service**, as specified in Chapters 22-24
2. The Language Mapping Specifications, which are organized into the following stand-alone volumes:
   - **Ada Mapping to OMG IDL**
   - **C Mapping to OMG IDL**
   - **C++ Mapping to OMG IDL**
   - **COBOL Mapping to OMG IDL**
   - **IDL Script Mapping**
   - **IDL to Java Mapping**
   - **Java Mapping to OMG IDL**
   - **Lisp Mapping to OMG IDL**
   - **Python Mapping to OMG IDL**
   - **Smalltalk Mapping to OMG IDL**

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.

Helvetica bold - OMG Interface Definition Language (OMG IDL) and syntax elements.
`Courier bold` - Programming language elements.
Helvetica – Exceptions

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

Acknowledgements

This specification was based upon "IDL to Java™ Language Mapping Specification"(formal/02-08-05).

OMG IDL to PHP Language Mapping

This chapter describes the complete mapping of IDL into the PHP language.

Examples of the mapping are provided. It should be noted that the examples are code fragments that try to illustrate only the language construct being described. Normally they will be embedded in some module.

Alignment

The OMG IDL to PHP Language Mapping specification is aligned with CORBA, version 2.4.

Contents

This chapter contains the following sections.

1.1 Introduction

This section describes the complete mapping of IDL into the PHP language. It is based upon PHP version 5.0 and above.

Examples of the mapping are provided. It should be noted that the examples are code fragments that try to illustrate only the language construct being described.

In a variety of places, methods are shown as throwing particular system exceptions. These instances are identified because it was felt that the particular system exception was more likely to occur and hence is called out as a “hint” to callers (and implementers). Please remember that, except where specifically mandated by this specification, it is implementation dependent (and legal) for the ORB runtime to raise other system exceptions.

There are two cases in which methods in the **org::omg namespace** are specified as throwing the exception **org__omg__CORBA__NO_IMPLEMENT**:

1. Deprecated methods - Implementations are not mandatory for these methods. They are clearly marked with a phpdoc comment.

2. Non abstract methods (not deprecated) - that have been added since the PHP language mapping was first approved. To maintain binary compatibility, these methods are not defined as abstract, but as concrete and throw **org__omg__CORBA__NO_IMPLEMENT**. Conforming implementations shall override these methods to provide the specified semantics.

3. In certain rare cases, the actual body of a method must be replaced by a vendor specific implementation. These cases are clearly identified in this specification and by comments in the **org::omg namespace**.

In various places the notation **{...}** is used in describing PHP code. This indicates that concrete PHP code will be generated for the method body and that the method is concrete, not abstract. Normally the generated code is specific to a particular vendor's implementation and is "internal" to their implementation.

1.1.1 org::omg Namespace

The PHP language mapping is highly dependent upon the specification of a set of standard PHP packages contained in **org::omg namespace**. This includes PHP classes for all PIDL, native types, and ORB portability interfaces.

It is probable that OMG specifications that are adopted in the future may make changes to these classes (e.g., to add a method to the ORB). Care must be taken to ensure that such future changes do not break binary compatibility with previous versions.

1.1.1.1 Allowable Modifications

Conforming implementations may not add or subtract anything from the definitions contained in the org::omg namespace except as follows:

- Vendor-specific implementations for the init methods of **org__omg__CORBA__ORB** must be supplied. Since these methods are static, they cannot be overridden by the vendor-specific **ORB** subclass, but must be provided in the **org__omg__CORBA__ORB** class itself.
- The addition of *phpdoc* comments for documenting ORB APIs. Removal of specified *phpdoc* comments, in particular comments marking code as deprecated, is forbidden.
- The names of formal parameters for methods for which the entire implementation is provided by the vendor.

1.2 Names

In general IDL names and identifiers are mapped to PHP names and identifiers with no change. If a name collision could be generated in the mapped PHP code, the name collision is resolved by prepending an underscore (_) to the mapped name.

In addition, because of the nature of the PHP language, a single IDL construct may be mapped to several (differently named) PHP constructs. The "additional" names are constructed by appending a descriptive suffix. For example, the IDL interface **foo** is

mapped to the PHP interfaces **foo** and **fooOperations**, and additional PHP classes **fooHelper**, **fooPOA**, and optionally **fooPOATie**. If more than one reserved suffix is present in an IDL name, than an additional underscore is prepended to the mapped name for each additional suffix.

In those exceptional cases that the "additional" names could conflict with other mapped IDL names, the resolution rule described above is applied to the other mapped IDL names. The naming and use of required "additional" names takes precedence.

For example, an interface whose name is **fooHelper** is mapped to **_fooHelper**, regardless of whether an interface named **foo** exists. The helper classe for interface **fooHelper** are named **_fooHelperHelper**.

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

1.2.1 Reserved Names

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

- The PHP class ***<type>*Helper**, where ***<type>*** is the name of an IDL defined type.
- The PHP classes ***<interface>*Operations**, ***<interface>POA***, and ***<interface>*POATie**, where ***<interface>*** is the name of an IDL interface type.
- The nested scope PHP package name <type>Namespace, where <type>is the name of an IDL interface, valuetype, struct, union or exception (Section 1.17, "Mapping for Certain Nested Types," on page 1-65).
- The keywords in the PHP language:

From the PHP4 Manual - Appendix G. List of Reserved Words				
`and`	`array`	`as`	`break`	`case`
`cfunction`	`class`	`const`	`continue`	`declare`
`default`	`die`	`do`	`echo`	`else`
`elseif`	`empty`	`enddeclare`	`endfor`	`endforeach`
`endif`	`endswitch`	`endwhile`	`eval`	`exit`
`extends`	`for`	`foreach`	`function`	`global`
`if`	`include`	`include_once`	`isset`	`list`
`new`	`oldfunction`	`or`	`print`	`require`
`require_once`	`return`	`static`	`switch`	`unset`
`use`	`var`	`while`	`xor`	`__FUNCTION__`
`__CLASS__`	`__FILE__`	`__LINE__`		

From the PHP5 new Reserved Words				
`abstract`	`catch`	`final`	`implements`	`instanceof`
`interface`	`private`	`protected`	`public`	`throw`
`try`				

- The additional PHP constants:
  `true false null`

The use of any of these names for a user defined IDL type or interface (assuming it is also a legal IDL name) will result in the mapped name having an underscore (_) prepended.

1.3 Mapping of Module

Since PHP does not have namespaces, an IDL module is not directly mapped to a PHP language structure. Instead, all IDL type declarations within the module are mapped to corresponding PHP class or interface declarations prefixed with the same name as all the modules up in the hierarchy, each module separeted by a double underscore "__".

IDL declarations not enclosed in any modules are mapped into the (unnamed) PHP global scope(without prefixed underscores).

1.3.1 Example

```
// IDL
module Example {
    module Easy {
        interface sample {
        }
    }
    interface OneLevel {
    }
}

// generated PHP
class Example__Easy__sample { }
class Example__OneLevel { }
```

1.4 Mapping for Basic Types

1.4.1 Introduction

Table 1-1 on page 1-4 shows the basic mapping. In some cases where there is a potential mismatch between an IDL type and its mapped PHP type, the Exceptions column lists the standard CORBA exceptions that may be (or are) raised. See Section 1.15, "Mapping for Exception," on page 1-55 for details on how IDL system exceptions are mapped.

The potential mismatch can occur when the range of the PHP type is "larger" than IDL. The value must be effectively checked at runtime when it is marshaled as an in parameter (or on input for an inout).

Table 1-1 Basic Type Mappings

IDL Type	PHP type	Exceptions
boolean	boolean	CORBA__DATA_CONVERSION
char	ANSI string	CORBA__DATA_CONVERSION
wchar	unicode string	CORBA__DATA_CONVERSION
octet	int	CORBA__DATA_CONVERSION
string	ANSI string	CORBA__MARSHAL
wstring	unicode string	CORBA__MARSHAL CORBA__DATA_CONVERSION
short	int	CORBA__DATA_CONVERSION
unsigned short	int	CORBA__DATA_CONVERSION
long	int	CORBA__DATA_CONVERSION
unsgined long	int	CORBA__DATA_CONVERSION

IDL Type	PHP type	Exceptions
float	double	CORBA__DATA_CONVERSION
double	double	CORBA__DATA_CONVERSION

1.4.1.1 Future Support

In the future it is expected that the IDL types **fixed, long long** and **long double** will be supported directly by PHP. Currently there is no support for these types in PHP5, so they are mapped to PHP Classes in php::**math** namespace.

IDL Type	PHP type	Exceptions
long long	php__math__BigInteger	CORBA__DATA_CONVERSION
unsigned long long	php__math__BigInteger	CORBA__DATA_CONVERSION
fixed	php__math__BigDecimal	CORBA__DATA_CONVERSION
long double	php__math__BigFloat	CORBA__DATA_CONVERSION

1.4.1.2 IDLEntity

Many of the PHP interfaces and classes generated from IDL are marked by implementing or extending an empty marker interface **IDLEntity** which has no methods. The following sections identify the specifics. The IIOP serialization classes specified in the reverse PHP to IDL mapping (see the *PHP to IDL Language Mapping* specification) will detect these instances and marshal them using the generated marshaling code in the Helper class.

```
// PHP
interface org__omg__CORBA__portable__IDLEntity
   implements org__omg__CORBA__portable__spl__Serializable {}
```

1.4.1.3 PHP Serialization

Those generated classes that are not abstract, including the stub classes, shall support PHP object serialization semantics. For example, generated helper classes do not have to be serializable. The following classes support PHP object serialization semantics:

- Stub classes
- Abstract base classes for concrete valuetypes
- Implementation classes for concrete valuetypes
- Any class that implements IDLEntity

1.4.1.4 Holder Classes

Since PHP has support for pass-by-reference parameters, there is no need for Holder classes.

*1.4.1.5 Use of PHP **null***

The PHP **null** may only be used to represent the “null” object reference or value type. For example, a zero length string rather than **null** must be used to represent the empty string. Similarly for arrays.

1.4.2 Boolean

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

1.4.3 Character Types

IDL characters are mapped to a PHP one-character-length string. In order to enforce type safety, the PHP CORBA runtime asserts range validity of all PHP string(char) mapped to IDL char when parameters are marshalled during method invocation. If the string(char) are oversized, a CORBA__DATA_CONVERSION exception shall be thrown.

The IDL **wchar** maps to a UTF-8 one-character-length string in PHP. In order to enforce type safety, the PHP CORBA runtime asserts range validity of all PHP string (wchar) mapped to IDL wchar when parameters are marshalled during method invocation. If the string(wchar) falls outside the range defined by the character set, a CORBA__DATA_CONVERSION exception shall be thrown.

1.4.4 Octet

The IDL type **octet**, an 8-bit quantity, is mapped to the PHP type **int**. In order to enforce type safety, the PHP CORBA runtime asserts range validity of all PHP int mapped to IDL byte. If the int falls outside the range defined by the byte type, a CORBA__DATA_CONVERSION exception shall be thrown.

1.4.5 String Types

The IDL **string**, both bounded and unbounded variants, are mapped to **string** in PHP. Bounds checking of the string is done at marshal time. Character range violations cause a CORBA__DATA_CONVERSION exception to be raised. Bounds violations cause a CORBA__BAD_PARAM exception to be raised.

The IDL **wstring**, both bounded and unbounded variants, are mapped to **string** in PHP. Range checking for characters in the string as well as bounds checking of the string is done at marshal time. Character range violations cause a CORBA__DATA_CONVERSION exception to be raised.

1.4.6 Integer Types

The basic integer types map as shown in Table 1-1 on page 1-4.
The integer types **long long** and **unsigned long long** map to php__**math__BigInteger**. Range checking is done at marshal time. Integer range violations cause a CORBA__DATA_CONVERSION exception to be raised.

1.4.7 Floating Point Types

The IDL **float** and **double** map as shown in Table 1-1 on page 1-4.
The float type long double maps to php__**math__BigFloat**. Range checking is done at marshal time. Integer range violations cause a CORBA__DATA_CONVERSION exception to be raised.

1.4.8 Fixed Point Types

The IDL **fixed** type is mapped to the PHP php__**math__BigDecimal** class. Range

checking is done at marshal time. Size violations cause a CORBA__DATA_CONVERSION exception to be raised.

1.5 Helpers

All user defined IDL types have an additional "helper" PHP class with the suffix **Helper** appended to the type name generated. (Note that in this context user defined includes IDL types that are defined in OMG specifications such as those for the Interface Repository and other OMG services.)

1.5.1 Helpers for Boxed Values

Although helper classes are generated for boxed value types, some of their specifics differ from the helpers for other user defined types. See Section 1.14, "Value Box Types," on page 1-49 for the details.

1.5.2 Helper Classes (except Boxed Values)

Several static methods needed to manipulate the type are supplied. These include **Any** insert and extract operations for the type, getting the repository id, getting the typecode, and reading and writing the type from and to a stream.

The helper class for a mapped IDL interface or abstract interface also has narrow and unchecked_narrow operations defined in the template below.

For any user-defined(non boxed value) complex type <**typename**>, except Array and Sequence, the following is the PHP code generated:

```
// generated PHP helper - non boxed value complex types (non
Array or Sequence)
abstract class <typename>Helper
{
   public static function enforceTypeChecking(&$t)
   {
      if (is_object($t)) {
         if (! $t instanceof <typename>) {
            throw new CORBA__DATA_CONVERSION;
         }
      } else {
         throw new CORBA__DATA_CONVERSION;
      }
   }

   public static function insert(org__omg__CORBA__Any $a,
      <typename> $t)  {
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
      self::enforceTypeChecking($ret);
      return $ret;
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
   }
   public static function read(
```

```
        org__omg__CORBA__portable__InputStream $is)
    {
        ...
        self::enforceTypeChecking($ret);
        return $ret;
    }
    public static void write(
        org__omg__CORBA__portable__OutputStream $os,
        <typename> $val)
    {
        ...
    }
    public static function narrow(org__omg__CORBA__Object $obj)
    {
        ...
    }
    public static function unchecked_narrow(
        org__omg__CORBA__Object $obj)           {
        ...
    }

    // for each factory declaration in non abstract
    // value type
    public static function <factoryname> (
        org__omg__CORBA__ORB $orb
        [ " ," <factoryarguments>] )
    {
        ...
    }
}
```

For any user defined(non-boxed value) basic type IDL type, **< typename>**, the following is the PHP code generated for the type.

```
// generated PHP helper - non boxed value basic types
abstract class <typename>Helper
{
    public static function enforceTypeChecking(&$t)
    {
        if (gettype($t) != "<typename>") {
            throw new CORBA__DATA_CONVERSION;
        }
    }
    public static function insert(org__omg__CORBA__Any $a, $t)
    {
        self::enforceTypeChecking($t);
        ...
    }
    public static function extract(org__omg__CORBA__Any $a)
    {
        ...
        self::enforceTypeChecking($ret);
        return $ret;
    }
    public static function type()
    {
        ...
    }
    public static function id()
    {
        ...
    }
    public static function read(
        org__omg__CORBA__portable__InputStream $is)
    {
        ...
        self::enforceTypeChecking($ret);
```

```
      return $ret;
   }
   public static void write(
        org__omg__CORBA__portable__OutputStream $os, $val)
   {
      self::enforceTypeChecking($val);
      ...
   }
   public static function narrow(org__omg__CORBA__Object $obj)
   {
      ...
   }
   public static function unchecked_narrow(
      org__omg__CORBA__Object $obj)           {
      ...
   }

   // for each factory declaration in non abstract
   // value type
   public static function <factoryname> (
        org__omg__CORBA__ORB $orb
        [ " ," <factoryarguments>] )
   {
      ...
   }
}
```

For any user defined(non-boxed value) Array or Sequence **< typename>**, the following is the PHP code generated for the type.

```
// generated PHP helper - non boxed value Array or Sequence
abstract class <typename>Helper
{
   public static function enforceTypeChecking(&$t)
   {
      if (gettype($t) != "array") {
         throw new CORBA__DATA_CONVERSION;
      }
      $dimensions = array (
         <dimension> => array(<number-of-elements>,"element-
type"),
         ...
      );

      if (! check_array($t, $dimensions)) {
         throw new CORBA__DATA_CONVERSION;
      }
   }

   public static function insert(org__omg__CORBA__Any $a, $t)
   {
      self::enforceTypeChecking($t);
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
      self::enforceTypeChecking($ret);
      return $ret;
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
```

```
    }
    public static function read(
         org__omg__CORBA__portable__InputStream $is)
    {
       ...
       self::enforceTypeChecking($ret);
       return $ret;
    }
    public static void write(
         org__omg__CORBA__portable__OutputStream $os, $val)
    {
       self::enforceTypeChecking($val);
       ...
    }
    public static function narrow(org__omg__CORBA__Object $obj)
    {
       ...
       self::enforceTypeChecking($ret);
       return $ret;
    }
    public static function unchecked_narrow(
       org__omg__CORBA__Object $obj)           {
       ...
    }

    // for each factory declaration in non abstract
    // value type
    public static function <factoryname> (
         org__omg__CORBA__ORB $orb
         [ " ," <factoryarguments>] )
    {
       ...
    }
}
```

1.5.2.1 Value type Factory Convenience Methods

For each factory declaration in a value type declaration, a corresponding static convenience method is generated in the helper class for the value type. The name of this method is the name of the factory.

This method takes an ORB instance and all the arguments specified in the factory argument list. The implementation of each of these methods will locate a ***<typename>*****ValueFactory** (see Section 1.13.8, “Value Factory and Marshaling,” on page 1-48) and call the identically named method on the **ValueFactory** passing in the supplied arguments.

1.5.3 Examples

```
// IDL - named type
module foo {
   struct stfoo {long f1; string f2;};
}

// generated PHP
abstract class foo__stfooHelper
{
   public static function enforceTypeChecking($t)
   {
      if (is_object($t)) {
         if (! $t instanceof foo__stfoo) {
            throw new CORBA__DATA_CONVERSION;
         }
      } else {
```

```
            throw new CORBA__DATA_CONVERSION;
         }
      }
      public static function insert(
            org__omg__CORBA__Any $a, foo__stfoo $t)
      {
         ...
      }
      public static function extract(org__omg__CORBA__Any $a)
      {
         ...
         self::enforceTypeChecking($ret);
         return $ret;
      }
      public static function type()
      {
         ...
      }
      public static function id()
      {
         ...
      }
      public static function read(
            org__omg__CORBA__portable__InputStream $is)
      {
         ...
         self::enforceTypeChecking($ret);
         return $ret;
      }
      public static function write(
            org__omg__CORBA__portable__OutputStream $os,
            foo__stfoo val)
      {
         ...
      }
   }
```

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

```
// generated PHP helper
abstract class IntSeqHelper
{
   public static function enforceTypeChecking(&$t)
   {
      if (! is_array($t)) {
         throw new CORBA__DATA_CONVERSION;
      }
      if (! check_array($t, array(-1), "long"))
      {
         throw new CORBA__DATA_CONVERSION;
      }
   }
   public static function insert(
         org__omg__CORBA__Any $a,
         $t)
   {
      self::enforceTypeChecking($t);
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
      self::enforceTypeChecking($ret);
      return $ret;
   }
   public static function type()
   {
```

```
        ...
    }
    public static function id()
    {
        ...
    }
    public static function read(
            org__omg__CORBA__portable__InputStream $is)
    {
        ...
        self::enforceTypeChecking($t);
        return $ret;
    }
    public static void write(
            org__omg__CORBA__portable__OutputStream $os, $val)
    {
        self::enforceTypeChecking($val);
        ...
    }
}
```

1.6 Mapping for Constant

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

1.6.1 Constants Within An Interface

Constants declared within an IDL interface are mapped to class/interface constants. Note that because the signature interface extends the operations interface for non-abstract IDL interfaces, the constant is available in all the mapped PHP interfaces.

1.6.1.1 Example

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

```
// generated PHP
interface Example__FaceOperations
{
    const aLongerOne = -321;
}
interface Example__Face implements Example__FaceOperations,
        org__omg__CORBA__Object,
        org__omg__CORBA__portable__IDLEntity
{}
// Helper class omitted for simplicity
```

1.6.2 Constants Not Within An Interface

Constants not declared within an IDL interface are mapped to public constants using the PHP `define` statement.

1.6.2.1 Example

```
// IDL
module Example {
    const long aLongOne = -123;
```

```
};

// generated PHP
define ("Example__aLongOne", -123);
```

1.7 Mapping for Enum

An IDL **enum** is mapped to a PHP class that implements IDLEntity with the same name as the **enum** type, which declares a value method, two static data member per label, an integer conversion method, a private constructor, a public initialization method and a readResolve method as follows:

```
// generated PHP

final class <enum_name>
     implements org__omg__CORBA__portable__IDLEntity
{
   // one pair of static declaration for each label in the enum
   public static $<label> = null;
   const _<label> = <value>;

   private $value;
   public static function initialize()
   {
      //one declaration for each label in the enum
      <enum_name>::$<label> =
           new <enum_name>(<enum_name>::_<label>);
   }
   private function __construct($value)
   {
      $this->value = $value;
   }

   // get the enum value
   public function value()
   {
      return $this->value;
   }

   // get enum with specified value
   public static function from_int($value)
   {
      ...
   }

   public static function readResolve()
   {
      ...
   }
}
<enum_name>::initialize();
```

One of the members is a **static** that has the same name as the IDL enum label. The other has an underscore (_) prepended and is intended to be used in switch statements.

The value method returns the integer value. Values are assigned sequentially starting with 0. Note that there is no conflict with the **value()** method in PHP even if there is a label named **value**.

There shall be only one instance for each enum label. Since there is only one instance, equality tests will work correctly. Note that it is necessary to supply a **readResolve** method to enforce uniqueness of enum elements, as otherwise serialization followed by deserialization will create a new element.

The PHP class for the enum has an additional method **from_int()**, which returns the enum with the specified value if the specified value corresponds to an element of the enum. If the specified value is out of range, a BAD_PARAM exception with a standard minor code of 25 is raised.

A helper class is also generated according to the normal rules, see Section 1.5, "Helpers," on page 1-13.

1.7.1 Example

```
// IDL
enum EnumType {first, second, third, fourth, fifth};

// generated PHP

final class EnumType
     implements org__omg__CORBA__portable__IDLEntity
{
   public static $first = null;
   const _first = 0;
   public static $second = null;
   const _second = 1;
   public static $third = null;
   const _third = 2;
   public static $fourth = null;
   const _fourth = 3;
   public static $fifth = null;
   const _fifth = 4;

   private $value;
   public static function initialize()
   {
      EnumType::$first = new EnumType(EnumType::_first);
      EnumType::$second = new EnumType(EnumType::_second);
      EnumType::$third = new EnumType(EnumType::_third);
      EnumType::$fourth = new EnumType(EnumType::_fourth);
      EnumType::$fifth = new EnumType(EnumType::_fifth);
   }

   public function value()
   {
      return $this->value;
   }

   public static function from_int($value)
   {
      switch ($value) {
         case EnumType::_first: return EnumType::$first;
         case EnumType::_second: return EnumType::$second;
         case EnumType::_third: return EnumType::$third;
         case EnumType::_fourth: return EnumType::$fourth;
         case EnumType::_fifth: return EnumType::$fifth;
         default:
            throw new org__omg__CORBA__BAD_PARAM(25);
      }
   }
   // constructor
   private function __construct($value)
   {
      $this->value = $value;
   }
   public function readResolve()
   {
      return EnumType::from_int( $this->value() );
   }
```

```
}
EnumType::initialize();

// generated PHP helper
abstract class EnumTypeHelper
{
   public static function enforceTypeChecking($t)
   {
      if (is_object($t)) {
         if (! $t instanceof EnumType) {
            throw new CORBA__DATA_CONVERSION;
         }
      } else {
         throw new CORBA__DATA_CONVERSION;
      }
   }
   public static function insert(org__omg__CORBA__Any $a,
      EnumType $t)
   {
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
      self::enforceTypeChecking($ret);
      return $ret;
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
   }
   public static function read(
      org__omg__CORBA__portable__InputStream $is)
   {
      ...
      self::enforceTypeChecking($ret);
      return $ret;
   }
   public static function write(
         org__omg__CORBA__portable__OutputStream $os,
         EnumType $val)
   {
      ...
   }
}
```

1.8 Mapping for Struct

An IDL **struct** is mapped to a final PHP class with the same name and which provides instance variables for the fields in IDL member ordering, a constructor for all values (with defaults), and which implements **IDLEntity**. All fields in the struct are initialized. Strings are initialized to "".

A helper class is also generated according to the normal rules, see Section 1.5, "Helpers," on page 1-13.

1.8.1 Example

```
// IDL
struct StructType {
   long field1;
```

```
   string field2;
};

// generated PHP
final class StructType
      implements org__omg__CORBA__portable__IDLEntity
{

   // instance variables
   public $field1 = 0;
   public $field2 = "";

   // constructors
   public function __construct($f1=0, $f2="")
   {
      $this->field1 = $f1;
      $this->field2 = $f2;
   }
}

abstract class StructTypeHelper
{
   public static function enforceTypeChecking($t)
   {
      if (is_object($t)) {
         if (! $t instanceof StructType) {
            throw new CORBA__DATA_CONVERSION;
         }
      } else {
         throw new CORBA__DATA_CONVERSION;
      }
   }
   public static function insert(org__omg__CORBA__Any $a,
      StructType $t)  {
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
      self::enforceTypeChecking($ret);
      return $ret;
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
   }
   public static function read(
         org__omg__CORBA__portable__InputStream $is)
   {
      ...
      self::enforceTypeChecking($ret);
      return $ret;
   }
   public static function write(
         org__omg__CORBA__portable__OutputStream $os,
      StructType $val)
   {
      ...
   }
}
```

1.9 Mapping for Union

An IDL **union** is mapped to a final PHP class with the same name, which implements **IDLEntity** and has

- a map between the discriminator value and the branch.
- three private attributes: discriminator value, the current union value and the initialized flag;
- a default constructor.
- methods for accessing the discriminator, named **getDiscriminator()** and **setDiscriminator()**.
- overloaded attributes thru __get and __set.

The branch accessor and modifier methods are overloaded by __get and __set methods. The __get method shall raise the CORBA__BAD_OPERATION system exception if the expected branch has not been set.

The **setDiscriminator** method should throw a BAD_PARAM exception with a standard OMG minor code of 34 when a value is passed for the discriminator that is not among the case labels.

It is illegal to specify a union with a default case label if the set of case labels completely covers the possible values for the discriminant. It is the responsibility of the PHP code generator (e.g., the IDL complier, or other tool) to detect this situation and refuse to generate illegal code.

If there is no explicit default case label, and the set of case labels does not completely cover the possible values of the discriminant, the default label should be mapped to the first case value.

The discriminator attribute should always have a default value of **null,** in which case, the default label will always be selected.

A helper class is also generated according to the normal rules, see Section 1.5, “Helpers,” on page 1-13.

1.9.1 Example

```
// IDL - EnumType from Section 1.7.1, “Example,” on page 1-14
union UnionType switch (EnumType) {
   case first: long win;
   case second: short place;
   case third:
   case fourth: octet show;
   default: boolean other;
};

// generated PHP
final class UnionType
     implements org__omg__CORBA__portable__IDLEntity
{
   static private $unionMap = array (
         EnumType::_first => “win”,
         EnumType::_second => “place”,
         EnumType::_third => “show”,
         EnumType::_fourth => “show”);
   private $discriminator=null;
   private $value;
   private $initialized = false;
   // constructor
   public function __construct()
```

```
   {
   }
   // discriminator accessor
   public function getDiscriminator()
   {
      return $this->discriminator;
   }
   public function setDiscriminator($value)
   {
      if (is_object($value)) { //Enum
         if (get_class($value)=="enumtype") {
            $value = $value->value();
         } else {
            throw new org__omg__CORBA__BAD_PARAM(34);
         }
      }
      if (! is_numeric($value)) {
         throw new org__omg__CORBA__BAD_PARAM(34);
      }
      if ($this->discriminator !== $value) {
         if (!array_key_exists($value, UnionType::$unionMap)) {
            throw new org__omg__CORBA__BAD_PARAM(34);
         }
         $this->initialized = false;
         $this->discriminator = $value;
      }
   }
   public function __get($propname)
   {
      $prop = "other"; //default label
      if (is_numeric($this->discriminator)) {
         if (array_key_exists($this->discriminator,
               UnionType::$unionMap)) {
            $prop = UnionType::$unionMap[$this->discriminator];
         }
      }
      if ($prop != $propname) {
         throw new org__omg__CORBA__BAD_OPERATION;
      }
      if (! $this->initialized) {
         throw new org__omg__CORBA__BAD_OPERATION;
      }
      return $this->value;
   }
   public function __set($propname, $propvalue)
   {
      $prop = "other"; //default label
      if (is_numeric($this->discriminator)) {
         if (array_key_exists($this->discriminator,
               UnionType::$unionMap)) {
            $prop = UnionType::$unionMap[$this->discriminator];
         }
      }
      if ($prop != $propname) {
         throw new  org__omg__CORBA__BAD_OPERATION;
      }
      $this->initialized = true;
      $this->value = $propvalue;
   }
}

abstract class UnionTypeHelper
{
   public static function enforceTypeChecking($t)
   {
      if (is_object($t)) {
         if (! $t instanceof UnionType) {
            throw new CORBA__DATA_CONVERSION;
```

```
            }
        } else {
            throw new CORBA__DATA_CONVERSION;
        }
    }
    public static function insert(
            org__omg__CORBA__Any $a, UnionType $t)
    {
        ...
    }
    public static function extract(org__omg__CORBA__Any $a)
    {
        ...
        self::enforceTypeChecking($ret);
        return $ret;
    }
    public static function type()
    {
        ...
    }
    public static function id()
    {
        ...
    }
    public static function read(
            org__omg__CORBA__portable__InputStream $is)
    {
        ...
        self::enforceTypeChecking($ret);
        return $ret;
    }
    public static function write(
            org__omg__CORBA__portable__OutputStream $os, UnionType
$val)
    {
        ...
    }
}
```

1.10 Mapping for Sequence

An IDL **sequence** is mapped to a PHP array with the same name. In the mapping, everywhere the sequence type is needed, an array of the mapped type of the sequence element is used. Bounds checking shall be done on bounded sequences when they are marshaled as parameters to IDL operations, and an IDL CORBA__MARSHAL is raised if necessary.

A helper class is also generated according to the normal rules, see Section 1.5, “Helpers,” on page 1-13.

1.10.1 Example

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

// generated PHP

abstract class UnBoundedDataHelper
{
    public static function enforceTypeChecking(&$t)
    {
        if (gettype($t) != "array") {
```

```
            throw new CORBA__DATA_CONVERSION;
         }
         $dimensions = array (
            <dimension> => array(-1, "long"),
            ...
         );

         if (! check_array($t, $dimensions)) {
            throw new CORBA__DATA_CONVERSION;
         }
      }
      public static function insert(org__omg__CORBA__Any $a, $t)
      {
         self::enforceTypeChecking($t);
         ...
      }
      public static function extract(org__omg__CORBA__Any $a)
      {
         ...
         self::enforceTypeChecking($ret);
         return $ret;
      }
      public static function type() {...}
      public static function id() {...}
      public static function read(
            org__omg__CORBA__portable__InputStream $istream)
      {
         ...
         self::enforceTypeChecking($ret);
         return $ret;
      }
      public static function write(
            org__omg__CORBA__portable__OutputStream $ostream, $val)
      {
         self::enforceTypeChecking($val);
         ...
      }
   }

   abstract class BoundedDataHelper
   {
      public static function enforceTypeChecking(&$t)
      {
         if (gettype($t) != "array") {
            throw new CORBA__DATA_CONVERSION;
         }
         $dimensions = array (
            <dimension> => array(42, "long"),
            ...
         );

         if (! check_array($t, $dimensions)) {
            throw new CORBA__DATA_CONVERSION;
         }
      }
      public static function insert(org__omg__CORBA__Any $a, $t)
      {
         self::enforceTypeChecking($t);
         ...
      }
      public static function extract(Any a)
      {
         ...
         self::enforceTypeChecking($ret);
         return $ret;
      }
      public static function type() {...}
      public static function id() {...}
```

```
   public static function read(
        org__omg__CORBA__portable__InputStream $istream)
   {
      ...
      self::enforceTypeChecking($ret);
      return $ret;
   }
   public static function write(
        org__omg__CORBA__portable__OutputStream $ostream, $val)
   {
      self::enforceTypeChecking($val);
      ...
   }
}
```

1.11 Mapping for Array

An IDL array is mapped the same way as an IDL bounded sequence. In the mapping, everywhere the array type is needed, an array of the mapped type of the array element is used. In PHP, the natural PHP subscripting operator is applied to the mapped array.

The bounds for the array are checked when the array is marshaled as an argument to an IDL operation and a CORBA__MARSHAL exception is raised if a bounds violation occurs. The length of the array can be made available in PHP, by bounding the array with an IDL constant, which will be mapped as per the rules for constants.

1.11.1 Example

```
// IDL
const long ArrayBound = 42;
typedef long larray[ArrayBound];

// generated PHP
const ArrayBound = 42;
abstract class larrayHelper
{
   public static function enforceTypeChecking(&$t)
   {
      if (gettype($t) != "array") {
         throw new CORBA__DATA_CONVERSION;
      }
      $dimensions = array (
         <dimension> => array( ArrayBound, "long"),
         ...
      );

      if (! check_array($t, $dimensions)) {
         throw new CORBA__DATA_CONVERSION;
      }
   }
   public static function insert(org__omg__CORBA__Any $a, $t)
   {
      self::enforceTypeChecking($t);
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
      self::enforceTypeChecking($ret);
      return $ret;
   }
   public static function type()
   {
```

```
        ...
    }
    public static function id()
    {
        ...
    }
    public static function read(
          org__omg__CORBA__portable__InputStream $istream)
    {
        ...
        self::enforceTypeChecking($ret);
        return $ret;
    }
    public static function write(
          org__omg__CORBA__portable__OutputStream $ostream, $val)
    {
        self::enforceTypeChecking($val);
        ...
    }
}
```

1.12 Mapping for Interface

1.12.1 Basics

A non abstract IDL **interface** is mapped to two public PHP interfaces: a *signature interface* and an *operations interface*. The signature interface, which extends **IDLEntity**, has the same name as the IDL interface name and is used as the signature type in method declarations when interfaces of the specified type are used in other interfaces. The operations interface has the same name as the IDL interface with the suffix **Operations** appended to the end and is used in the server-side mapping and as a mechanism for providing optimized calls for collocated client and servers.

A helper class is also generated according to the normal rules, see Section 1.5, “Helpers,” on page 1-13.

The PHP operations interface contains the mapped operation signatures. If an operation raises exceptions, then the corresponding PHP method must throw PHP exceptions corresponding to the listed IDL exceptions.

The PHP signature interface extends the operations interface, the (mapped) base **org__omg__CORBA__Object**, as well as **org__omg__portable__IDLEntity**. Methods can be invoked on the signature interface. Interface inheritance expressed in IDL is reflected in both the PHP signature interface and operations interface hierarchies.

The helper class holds a static narrow method that allows an **org__omg__CORBA__Object** to be narrowed to the object reference of a more specific type. The IDL exception CORBA__BAD_PARAM is thrown if the narrow fails because the object reference does not support the requested type. A different system exception is raised to indicate other kinds of errors. Trying to narrow a **null** will always succeed with a return value of **null**.

The helper class holds a static unchecked_narrow method that allows an org__omg__CORBA__Object to be narrowed to the object reference of a more specific type. No type-checking is performed to verify that the object actually supports the requested type. The IDL exception CORBA__BAD_OPERATION can be expected if unsupported operations are invoked on the new returned reference, but no failure is expected at the time of the unchecked_narrow.

There are no special “nil” object references. PHP **null** can be passed freely wherever an object reference is expected.

Attributes are mapped to a pair of PHP accessor and modifier methods. These methods have the same name as the IDL attribute prefixed by a "get_" for accessors or a "set_" for modifiers. There is no modifier method for IDL **readonly** attributes.

Attribute exceptions are mapped as follows:
1. If a readonly attribute raises exceptions, then the PHP read accessor method must throw PHP exceptions corresponding to the listed IDL exceptions.
2. If an attribute has a getRaises clause, the PHP accessor method must throw PHP exceptions corresponding to the IDL exceptions listed in the getRaises clause.
3. If an attribute has a setRaises clause, the PHP modifier method must throw PHP exceptions corresponding to the IDL exceptions listed in the setRaises clause.

Local Interfaces

A new interface in org__omg__CORBA called **LocalInterface** is defined as:
```
interface LocalInterface extends org__omg__CORBA__Object {}
```
A local interface **<typename>** is mapped to the following PHP classes:
```
interface <typename>
   extends <typename>Operations,
      org__omg__CORBA__LocalInterface,
      org__omg__CORBA__portable__IDLEntity
```
where **interface <typename**> and **<typename>Operations** are identical to the mapping for a non-local interface, except for the inheritance relationship of **interface <typename>**.

In order to support _is_a, it is necessary to have information about the repository_ids of all super-interfaces of the local interface. This requires generating a base class that is used for implementations of local interfaces. This base class must satisfy the following requirements:
1. The base class is a public abstract class named _<typename>LocalBase.
2. It must define the _ids() method, which is a method in org__omg__CORBA__LocalObject.
3. The list of strings returned from the _ids() method must start with the repository ID of the most derived interface.

An implementation of **<typename>** may then be specified as:
```
class <typename>Impl extends <typename>LocalBase
{
   //Whatever constructors this implementation needs
   ...
   // Implementation of methods defined in
   // <typename>Operations
   ...
}
```
and an instance would be created using the usual PHP language construct:
```
<typename>Impl $ti = new <typename>Impl(...);
```
A helper class is also generated according to the normal rules, see Section 1.5, "Helpers," on page 1-13.
ORB implementations shall detect attempts to marshal local objects and throw a CORBA__MARSHAL exception.

For example, consider the following IDL definition:

```
local interface Test {
   long ping( in long arg ) ;
} ;
```

This results in the following classes:
```
interface Test extends TestOperations,
     org__omg__CORBA__LocalInterface,
     org__omg__CORBA__IDLEntity {}
interface TestOperations
```

```
{
   function ping( $arg );
}
abstract class _TestLocalBase extends
      org__omg__CORBA__LocalObject implements Test {
   private $_type_ids = array( "IDL:Test:1.0" );
   public function _ids() {
      return $this->_type_ids;
   }
}
```

Abstract Interfaces

An IDL **abstract interface** is mapped to a single public PHP interface with the same name as the IDL interface. The mapping rules are similar to the rules for generating the PHP operations interface for a non-abstract IDL interface. However this interface also serves as the signature interface, and hence extends **org__omg__CORBA__portable__IDLEntity**. The mapped PHP interface has the same name as the IDL interface name and is also used as the signature type in method declarations when interfaces of the specified type are used in other interfaces. It contains the methods which are the mapped operations signatures.

A helper class is also generated according to the normal rules, see Section 1.5, “Helpers,” on page 1-13.
CORBA__AbstractBase is mapped to **php__lang__Object**.

1.12.1.1 Example

```
// IDL
module Example {
   interface Marker {
   };
   abstract interface Base {
      void baseOp();
   };
   interface Extended: Base, Marker {
      long method (in long arg) raises (e);
      attribute long assignable;
      readonly attribute long nonassignable;
   }
}

// generated PHP
interface Example__MarkerOperations
{}
interface Example__Base extends
      org__omg__CORBA__portable__IDLEntity
{
   function baseOp();
}
interface Example__ExtendedOperations extends
      Example__Base, Example__MarkerOperations
{
   function method($arg);
   function get_assignable()
   function set_assignable($i);
   function get_nonassignable();
}
interface Example__Marker extends Example__MarkerOperations,
      org__omg__CORBA__Object,
      org__omg__CORBA__portable__IDLEntity
{}
interface Example__Extended extends Example__ExtendedOperations,
      Example__Marker,
```

```
     org__omg__CORBA__portable__IDLEntity
{}
abstract class Example__ExtendedHelper {
   public static function enforceTypeChecking($t)
   {
      if (is_object($t)) {
         if (! $t instanceof Example__Extended) {
            throw new CORBA__DATA_CONVERSION;
         }
      } else {
         throw new CORBA__DATA_CONVERSION;
      }
   }
   public static function insert(org__omg__CORBA__Any $a,
         Example__Extended $t)
   {
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
      self::enforceTypeChecking($ret);
      return $ret;
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
   }
   public static function read(
         org__omg__CORBA__portable__InputStream $is)
   {
      ...
      self::enforceTypeChecking($ret);
      return $ret;
   }
   public static function write(
         org__omg__CORBA__portable__OutputStream $os,
         Example__Extended $val)
   {
      ...
   }
   public static function narrow(org__omg__CORBA__Object $obj)
   {
      ...
   }
}
abstract class Example__BaseHelper {
   public static function insert(org__omg__CORBA__Any $a,
         Example__Base $t)
   {
      ...
   }
   public static function extract(org__omg__CORBA__Any $a) {
      ...
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
   }
   public static function read(
```

```
        org__omg__CORBA__portable__InputStream $is)
    {
        ...
    }
    public static function write(
        org__omg__CORBA__portable__OutputStream $os,
        Example__Base $val)
    {
        ...
    }
    public static function narrow($obj)
    {
        if (! is_object($obj)) {
            throw CORBA__BAD_PARAM;
        }
        ...
    }
}
abstract class MarkerHelper{
    public static function insert(org__omg__CORBA__Any $a,
        Example__Marker $t)
    {
        ...
    }
    public static function extract(org__omg__CORBA__Any $a)
    {
        ...
    }
    public static function type()
    {
        ...
    }
    public static function id()
    {
        ...
    }
    public static function read(
        org__omg__CORBA__portable__InputStream $is)
    {
        ...
    }
    public static function write(
        org__omg__CORBA__portable__OutputStream $os,
        Example__Marker val)
    {
        ...
    }
    public static function narrow(org__omg__CORBA__Object $obj)
    {
        ...
    }
}
```

1.12.2 Parameter Passing Modes

IDL **in** parameters, which implement call-by-value semantics, are mapped to normal PHP parameters. The results of IDL operations are returned as the result of the corresponding PHP method.

IDL **out** and **inout** parameters, which implement call-by-result and call-byvalue/result semantics, are mapped to PHP pass-by-reference parameters(&).

- For IDL **in** parameters that are not valuetypes:
  - PHP objects passed for non-valuetype IDL **in** parameters are created and owned

by the caller. With the exception of value types, the callee must not modify **in** parameters or retain a reference to the **in** parameter beyond the duration of the call. Violation of these rules can result in unpredictable behavior.

- For IDL **in** parameters that are valuetypes:
  - PHP objects passed for valuetype IDL **in** parameters are created by the caller and a copy is passed to the callee. The callee may modify or retain a reference to the copy beyond the duration of the call.
- PHP objects returned as IDL **out** or return parameters are created and owned by the callee. Ownership of such objects transfers to the caller upon completion of the call. The callee must not retain a reference to such objects beyond the duration of the call. Violation of these rules can result in unpredictable behavior.
- IDL **inout** parameters have the above **in** semantics for the **in** value, and have the above **out** semantics for the **out** value.
- The above rules do not apply to PHP primitive types.

1.12.2.1 Example

```
// IDL
module Example {
   interface Modes {
      long operation(in long inArg, out long outArg, inout long inoutArg);
   };
};

// Generated PHP
interface Example__ModesOperations {
   function operation($inArg, &$outArg, &$inoutArg);
}
interface Example__Modes extends Example__ModesOperations,
      org__omg__CORBA__Object,
      org__omg__CORBA__portable__IDLEntity
{}
abstract class ModesHelper {
   public static function insert(org__omg__CORBA__Any $a,
      Example__Modes $t)
   {
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
   }
   public static function read(
        org__omg__CORBA__portable__InputStream $is)
   {
      ...
   }
   public static function write(
        org__omg__CORBA__portable__OutputStream $os,
        Example__Modes $val)
   {
      ...
   }
   public static function narrow($obj)
   {
      if (! is_object($obj)) {
```

```
            throw CORBA__BAD_PARAM;
         ...
      }
   }
```

In the above, the result comes back as an ordinary result and the actual **in** parameters only is an ordinary value. For the **out** and **inout** parameters, the only change is the pass-by-reference operator(&).

Before the invocation, the input value of the **inout** parameter must be set.

1.12.3 Context Arguments to Operations

If an operation in an IDL specification has a context specification, then an **org__omg__CORBA__Context** input parameter (see Section 1.19.7, "Context," on page 1-71) is appended following the operation-specific arguments, to the argument list for an invocation.

1.13 Mapping for Value Type

1.13.1 PHP Interfaces Used For Value Types

This section describes several PHP interfaces that are used (and required) as part of the PHP mapping for IDL value types.

1.13.1.1 ValueBase Interface

```
interface org__omg__CORBA__portable __ValueBase
   extends org__omg__CORBA__portable __IDLEntity {
   function _truncatable_ids();
}
abstract class org__omg__CORBA__ValueBaseHelper {
   public static function insert(
      org__omg__CORBA__Any $a,
      php__io__Serializable $t)
   {
      ...
   }
   public static function extract(
      org__omg__CORBA__Any $a)
   {
      ...
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
   }
   public static function read(
      org__omg__CORBA__portable__InputStream $is)
   {
      ...
   }
   public static function write(
      org__omg__CORBA__portable__OutputStream $os,
      php__io__Serializable $val)
   {
      ...
   }
}
```

All value types implement **ValueBase** either directly (see Section 1.14.2, "Boxed Primitive Types," on page 1-49), or indirectly by implementing either the **StreamableValue** or **CustomValue** interface (see below).

1.13.1.2 StreamableValue Interface

```
interface org__omg__CORBA__portable__StreamableValue
   extends org__omg__CORBA__portable__Streamable,
   org__omg__CORBA__portable__ValueBase {}
```

All non-boxed IDL valuetypes that are not custom marshaled implement the **StreamableValue** interface.

1.13.1.3 CustomMarshal Interface

```
interface org__omg__CORBA__CustomMarshal {
   function marshal(org__omg__CORBA__DataOutputStream $os);
   function unmarshal(org__omg__CORBA__DataInputStream $is);
}
```

Implementers of custom marshaled values implement the **CustomMarshal** interface to provide custom marshaling.

The stream APIs that are passed as arguments for the marshal and unmarshal methods are not sufficient to marshal all valuetypes. To support custom marshaling of valuetypes, the ORB shall actually pass an instance of CustomOutputStream and CustomInputStream to these methods respectively (see Section 1.21.5, "Custom Streaming APIs," on page 1-118 for information on Custom stream APIs).

1.13.1.4 CustomValue Interface

```
interface org__omg__CORBA__portable__CustomValue
   extends org__omg__CORBA__portable__ValueBase,
   org__omg__CORBA__CustomMarshal {}
```

All custom value types generated from IDL implement the **CustomValue** interface.

1.13.1.4.1 ValueFactory Interface

```
interface org__omg__CORBA__portable__ValueFactory {
   function read_value(
      org__omg__CORBA_2_3__portable__InputStream $is);
}
```

The **ValueFactory** interface is the native mapping for the IDL type **CORBA::ValueFactory**. The **read_value()** method is called by the ORB runtime while in the process of unmarshaling a value type. A user implements this method as part of implementing a type specific value factory. In the implementation, the user calls **$is->read_value(php__io__Serializable)** with an uninitialized valuetype to use for unmarshaling. The value returned by the stream is the same value passed in, with all the data unmarshaled.

1.13.2 Basics For Stateful Value Types

A concrete value type (i.e., one that is not declared as abstract) is mapped to an abstract PHP class with the same name, and a factory PHP interface with the suffix "**ValueFactory**" appended to the value type name. In addition, a helper class with the suffix "**Helper**" appended to the value type name is generated.

The value type's mapped PHP abstract class contains instance variables that correspond

to the fields in the state definition in the IDL declaration. The order and name of the PHP instance variables are the same as the corresponding IDL state fields. Fields that are identified as **public** in the IDL are mapped to **public** instance variables. Fields that are identified as **private** in the IDL are mapped to **protected** instance variables in the mapped PHP class.
The PHP class for the value type extends either **org__omg__CORBA__portable__CustomValue** or **org__omg__CORBA__portable__StreamableValue**, depending on whether it is declared as custom in IDL or not, respectively.

The generated PHP class shall provide an implementation of the **ValueBase** interface for this value type. For value types that are streamable (i.e., non-custom), the generated PHP class also provides an implementation for the **org__omg__CORBA__portable__Streamable** interface.

The value type's generated value factory interface extends **org__omg__CORBA__portable__ValueFactory** and contains one method corresponding to each factory declared in the IDL. The name of the method is the same as the name of the factory, and the factory arguments are mapped in the same way as in parameters are for IDL operations. If the factory raises exceptions, then the corresponding PHP method must throw PHP exceptions corresponding to the listed IDL exceptions.
The implementor provides a factory class with implementations for the methods in the generated value factory interface. When no factories are declared in IDL, then the value type's value factory is eliminated from the mapping and the implementor simply implements **org__omg__CORBA__portable__ValueFactory** to provide the method body for **read_value()**.
The mapped PHP class contains abstract method definitions that correspond to the operations and attributes defined on the value type in IDL. Attributes are mapped in the same way as in interfaces, to modifier and accessor methods. Exceptions are mapped as follows:

1. If an operation raises exceptions, then the corresponding PHP method must throw PHP exceptions corresponding to the listed IDL exceptions.
2. If a readonly attribute raises exceptions, then the PHP read accessor method must throw PHP exceptions corresponding to the listed IDL exceptions.
3. If an attribute has a getRaises clause, the PHP accessor method must throw PHP exceptions corresponding to the listed IDL exceptions.
4. If an attribute has a setRaises clause, the PHP modifier method must throw PHP exceptions corresponding to the listed IDL exceptions.

An implementor of the value type extends the generated PHP class to provide implementation for the operations and attributes declared in the IDL, including those for any derived or supported value types or interfaces.

1.13.2.1 Inheritance From Value Types

The inheritance scheme and specifics of the mapped class depend upon the inheritance and implementation characteristics of the value type and are described in the following subsections.

Value types that do not inherit from other values or interfaces:

For non custom values, the generated PHP class also implements the **StreamableValue** interface and provides appropriate implementation to marshal the state of the object. For custom values, the generated class extends **CustomValue** but does not provide an implementation for the **CustomMarshal** methods.

Inheritance from other stateful values
The generated PHP class extends the PHP class to which the inherited value type is mapped. If the valuetype is custom, but its base is not, then the generated PHP class also

implements the **CustomValue** interface.

Inheritance from abstract values
The generated PHP class implements the PHP interface to which the inherited abstract value is mapped (see Section 1.13.3, "Abstract Value Types," on page 1-41).

Supported interfaces
The PHP class implements the Operations PHP interface of all the interfaces, if any, that it supports. (Note that the operations interface for abstract interfaces does not have the "Operations" suffix, see "Abstract Interfaces" on page 1-31). It also implements the appropriate interface, either **StreamableValue** or **CustomValue**, as per the rules stated in "Value types that do not inherit from other values or interfaces:" on page 1-40. The implementation of the supported interfaces of the value type shall use the tie mechanism, to tie to the value type implementation.

1.13.3 Abstract Value Types

An abstract value type maps to a PHP interface that extends **ValueBase** and contains all the operations and attributes specified in the IDL, mapped using the normal rules for mapping operations and attributes.
Abstract value types cannot be implemented directly. They must only be inherited by other stateful value types or abstract value types.

1.13.4 CORBA::ValueBase

CORBA::ValueBase is mapped to **php__io__Serializable**.

The **get_value_def()** operation is not mapped to any of the classes associated with a value type in PHP. Instead it appears as an operation on the ORB pseudo object in PHP (see "function get_value_def($repId)" in Section 1.19.10, "ORB," on page 1-79).

1.13.5 Example A

```
// IDL
typedef sequence<unsigned long> WeightSeq;
module ExampleA {
   valuetype WeightedBinaryTree {
      private long weight;
      private WeightedBinaryTree left;
      private WeightedBinaryTree right;
      factory createWBT(in long w);
      WeightSeq preOrder();
      WeightSeq postOrder();
   };
};

// generated PHP
abstract class ExampleA__WeightedBinaryTree
   implements org__omg__CORBA__portable__StreamableValue
{
   // instance variables and IDL operations
   protected $weight;
   protected $left;
   protected $right;
   public abstract function preOrder();
   public abstract function postOrder();
   // from ValueBase
   public function _truncatable_ids()
   {
      ...
   }
```

```
   // from Streamable
   public function _read(
      org__omg__CORBA__portable__InputStream $is)
   {
      ...
   }
   public function _write(
      org__omg__CORBA__portable__OutputStream $os)
   {
      ...
   }
   public function _type()
   {
      ...
   }
}
interface ExampleA__WeightedBinaryTreeValueFactory
   extends org__omg__CORBA__portable__ValueFactory
{
   function createWBT($weight)
   {
      ...
   }
}
abstract class ExampleA__WeightedBinaryTreeHelper
{
   public static function insert(
      org__omg__CORBA__Any $a,
      ExampleA__WeightedBinaryTree $t)
   {
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
   }
   public static WeightedBinaryTree read(
      org__omg__CORBA__portable__InputStream $is)
   {
      ...
   }
   public static function write(
      org__omg__CORBA__portable__OutputStream $os,
      ExampleA__WeightedBinaryTree $val)
   {
      ...
   }
   // for factory
   public static function createWBT(
      org__omg__CORBA__ORB $orb, $weight)
   {
      ...
   }
}
```

1.13.6 Example B

```
// IDL
module ExampleB {
```

```
   interface Printer {
      typedef sequence<unsigned long> ULongSeq;
      void print(in ULongSeq data);
   };
   valuetype WeightedBinaryTree supports Printer {
      private long weight;
      public WeightedBinaryTree left;
      public WeightedBinaryTree right;
      ULongSeq preOrder();
      ULongSeq postOrder();
   };
};

// generated PHP
interface ExampleB__PrinterOperations {
   function print ($data);
}
interface ExampleB__Printer extends
   ExampleB__PrinterOperations,
   org__omg__CORBA__Object,
   org__omg__CORBA__portable__IDLEntity {
}
abstract class PrinterHelper {
   public static function insert(org__omg__CORBA__Any $a,
      ExampleB__Printer $t)
   {
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
   }
   public static function read(
      org__omg__CORBA__portable__InputStream $is)
   {
      ...
   }
   public static function write(
      org__omg__CORBA__portable__OutputStream $os,
      ExampleB__Printer $val)
   {
      ...
   }
   public static function narrow(org__omg__CORBA__Object $obj)
   {
      ...
   }
}

abstract class ExampleB__WeightedBinaryTree
   implements ExampleB__PrinterOperations,
   org__omg__CORBA__portable__StreamableValue
{
   // instance variables and IDL operations
   protected $weight;
   public $left;
   public $right;
   public function preOrder()
   {
```

```
      ...
   }
   public function postOrder()
   {
      ...
   }
   public function print($data)
   {
      ...
   }
   // from ValueBase
   public function _truncatable_ids();
   // from Streamable
   public function _read(
      org__omg__CORBA__portable__InputStream $is);
   public function _write(
      org__omg__CORBA__portable__OutputStream $os);
   public function _type();
}
abstract class ExampleB__WeightedBinaryTreeHelper {
   public static function insert(
      org__omg__CORBA__Any $a, ExampleB__WeightedBinaryTree $t)
   {
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
   }
   public static function read(
      org__omg__CORBA__portable__InputStream $is)
   {
      ...
   }
   public static function write(
      org__omg__CORBA__portable__OutputStream $os,
      ExampleB__WeightedBinaryTree $val)
   {
      ...
   }
}
```

1.13.7 Parameter Passing Modes

If the formal parameter in the signature of an operation is a value type, then the actual parameter is passed by value. If the formal parameter type of an operation is an interface, then the actual parameter is passed by reference (i.e., it must be transformed to the mapped PHP interface before being passed).
IDL value type **in** parameters are passed as the mapped PHP class as defined above.
IDL value type **out** and **inout** parameters are passed using the pass-by-reference operator (&).

1.13.7.1 Example

```
// IDL - extended the above Example B
module ExampleB {
   interface Target {
```

```
      WeightedBinaryTree operation(
         in WeightedBinaryTree inArg,
         out WeightedBinaryTree outArg,
         inout WeightedBinaryTree inoutArg);
      };
   };
};

// generated PHP code
interface ExampleB__TargetOperations
{
   function operation(
      ExampleB__WeightedBinaryTree $inArg,
      ExampleB__WeightedBinaryTree &$outArg,
      ExampleB__WeightedBinaryTree &$inoutArg);
}
interface ExampleB__Target extends
   ExampleB__TargetOperations,
   org__omg__CORBA__Object,
   org__omg__CORBA__portable__IDLEntity
{}
abstract class TargetHelper {
   public static function insert(org__omg__CORBA__Any $a,
      ExampleB__Target $t)
   {
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
   }
   public static function read(
      org__omg__CORBA__portable__InputStream $is)
   {
      ...
   }
   public static function write(
      org__omg__CORBA__portable__OutputStream $os,
      ExampleB__Target $val)
   {
      ...
   }
   public static function narrow(org__omg__CORBA__Object $obj)
   {
      ...
   }
}
```

1.13.8 Value Factory and Marshaling

Marshaling PHP value instances is straightforward, but unmarshaling value instances is somewhat problematic. In PHP there is no *a priori* relationship between the RepositoryID encoded in the stream and the class name of the actual PHP class that implements the value. However, in practice we would expect that there will be a one-to-one relationship between the RepositoryID and the fully scoped name of the value type. However the RepositoryID may have an arbitrary prefix prepended to it, or be

completely arbitrary.

The following algorithm will be followed by the ORB:

- Look up the value factory in the RepositoryID to value factory map.
- If this is not successful and an expected type clz was passed, and if clz implements IDLEntity but not ValueBase, then unmarshal the valuetype as a boxed IDL type by calling the read method of the Helper.
- If this is not successful and the RepositoryId is a standard repository id that starts with "IDL:", then attempt to generate the value factory class name to use by stripping off the "IDL:" header and ":<major>.<minor>" version information trailer, and replacing the "/"s that separate the module names with "__"s and appending "DefaultFactory."
  - If this is not successful and the first two components of the PHP class name are "omg__org", then reverse the order of these components to be "org__omg" and repeat the above step.
- If this is not successful and the RepositoryId is a standard repository id that starts with "IDL:", then attempt to generate the boxed value helper class name to use by stripping off the "IDL:" header and ":<major>.<minor>" version information trailer, and replacing the "/"s that separate the module names with "__"s and appending "Helper."
  - If this is not successful and the first two components of the PHP class name are "omg__org", then reverse the order of these components to be "org__omg" and repeat the above step.
- If this is not successful, then raise the MARSHAL exception.

The IDL native type **ValueFactory** is mapped in PHP to **org__omg__CORBA__portable__ValueFactory**.
A null is returned when **register_value_factory()** is called and no previous RepositoryId was registered.
As usual, it is a tools issue, as to how RepositoryIDs are registered with classes. It is our assumption that in the vast majority of times, the above default implicit registration policies will be adequate. A tool is free to arrange to have the ORB's **register_value_factory()** explicitly called if it wishes to explicitly register a particular Value Factory with some RepositoryID. For example, this could be done by an "installer" in a server, by pre-loading the ORB runtime, etc.

1.14 Value Box Types

The rules for mapping value box types are specified in this section. There are two general cases to consider: value boxes that are mapped to PHP primitive types, and those that are mapped to PHP classes.
Helper classes are generated, however they have a somewhat different structure and inheritance hierarchy than helpers generated for other value types.

1.14.1 Generic BoxedValueHelper Interface

Concrete helper classes for boxed values are generated. They all implement the following PHP interface, which serves as a base for boxed value helpers.

```
interface org__omg__CORBA__portable__BoxedValueHelper {
   function read_value(
      org__omg__CORBA__portable__InputStream $is);
   function write_value(
      org__omg__CORBA__portable__OutputStream $os,
      php__io__Serializable $value);
   function get_id();
}
```

1.14.2 Boxed Primitive Types

If the value box IDL type maps to a PHP primitive (e.g., **float**, **long**, **char**, **wchar, string**, **wstring**,**boolean**, **octet**, **sequence**, **array**), then the value box type is mapped to a PHP class whose name is the same as the IDL value type. The class has a public data member named **value**. The helper class is also generated.

```
// IDL
valuetype <box_name> <primitive_type>;

// generated PHP
class <box_name> implements
   org__omg__CORBA__portable__ValueBase
{
   public $value;
   public <box_name>( $initial)
   {
      $this->value = $initial;
   }
   private static $_ids = array( <box_name>Helper::id() );
   public function _truncatable_ids()
   {
      return self::$_ids;
   }
}

class <box_name>Helper
   implements org__omg__CORBA__portable__BoxedValueHelper
{
   public static function insert(
      org__omg__CORBA__Any $a, <box_name> $t)
   {
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
   }
   public static function read(
      org__omg__CORBA__portable__InputStream $is)
   {
      ...
   }
   public static function write(
      org__omg__CORBA__portable__OutputStream $os,
      <box_name> $val)
   {
      ...
   }
   public function read_value(
      org__omg__CORBA__portable__InputStream $is)
   {
      ...
   }
   public function write_value(
      org__omg__CORBA__portable__OutputStream $os,
      php__io__Serializable $value)
   {
```

```
      ...
   }
   public function get_id()
   {
      ...
   }
}
```

1.14.2.1 Primitive Type Example

```
// IDL
valuetype MyLong long;
interface foo {
   void bar_in(in MyLong number);
   void bar_inout(inout MyLong number);_
};

// Generated PHP
class MyLong implements
   org__omg__CORBA__portable__ValueBase
{
   public $value;
   public function MyLong($initial)
   {
      $this->value = $initial;
   }
   private static $_ids = array(IntHelper::id());
   public function _truncatable_ids ()
   {
      return $self::_ids;
   }
}

class MyLongHelper
   implements org__omg__CORBA__portable__BoxedValueHelper
{
   public static function insert(
      org__omg__CORBA__Any $a, MyLong $t)
   {
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
   }
   public static function read(
      org__omg__CORBA__portable__InputStream $is)
   {
      ...
   }
   public static function write(
      org__omg__CORBA__portable__OutputStream $os,
      MyLong $val)
   {
      ...
   }
   public functional read_value(
      org__omg__CORBA__portable__InputStream $is)
```

```
   {
      ...
   }
   public function write_value(
      org__omg__CORBA__portable__OutputStream $os,
      php__io__Serializable $value)
   {
      ...
   }
   public function get_id()
   {
      ...
   }
}
interface fooOperations {
   function bar_in(MyLong $number);
   function bar_inout(MyLong &$number);
}
interface foo extends
   fooOperations,
   org__omg__CORBA__Object,
   org__omg__CORBA__portable__IDLEntity
{
}

abstract class fooHelper {
   public static function insert(
      org__omg__CORBA__Any $a,
      foo $t)
   {
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
   }
   public static function read(
      org__omg__CORBA__portable__InputStream $is)
   {
      ...
   }
   public static function write(
      org__omg__CORBA__portable__OutputStream $os, foo $val)
   {
      ...
   }
   public static function narrow($obj)
   {
      if (! is_object($obj)) {
         throw CORBA__BAD_PARAM;
      }
      ...
   }
}
```

1.14.3 Complex Types

If the value box IDL type is more complex and maps to a PHP class (e.g., **enum**, **struct**,

any, **interface**), then the value box type is mapped to the PHP class that is appropriate for the IDL type. Helper class is also generated.

1.14.3.1 Complex Type Example

```
// IDL
valuetype MySequence sequence<long>;
interface foo {
   void bar_in(in MySequence seq);
   void bar_inout(inout MySequence seq);
};

public class MySequenceHelper
   implements org__omg__CORBA__portable__BoxedValueHelper {
   public static function insert(org__omg__CORBA__Any $a, $t)
   {
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
   }
   public static function read(
      org__omg__CORBA__portable__InputStream $is)
   {
      ...
   }
   public static function write(
      org__omg__CORBA__portable__OutputStream $os,
      $val)
   {
      ...
   }
   public function read_value(
      org__omg__CORBA__portable__InputStream $is)
   {
      ...
   }
   public function write_value(
      org__omg__CORBA__portable__OutputStream $os,
      php__io__Serializable $value)
   {
      ...
   }
   public function get_id()
   {
      ...
   }
}
interface fooOperations {
   function bar_in($seq);
   function bar_inout(&$seq);
}
interface foo extends fooOperations,
   org__omg__CORBA__Object,
   org__omg__CORBA__portable__IDLEntity
{
}
```

```
abstract class fooHelper {
   public static function insert(org__omg__CORBA__Any $a, foo $t)
   {
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
   }
   public static function read(
      org__omg__CORBA__portable__InputStream $is)
   {
      ...
   }
   public static function write(
      org__omg__CORBA__portable__OutputStream $os, foo $val)
   {
      ...
   }
   public static function narrow($obj)
   {
      if (! is_object($obj)) {
         throw new CORBA__BAD_PARAM;
      }
      ...
   }
}
```

1.15 Mapping for Exception

IDL exceptions are mapped very similarly to structs. They are mapped to a PHP class that provides instance variables for the fields of the exception and constructors.
CORBA system exceptions are unchecked exceptions. They inherit (indirectly) from `php__lang__RuntimeException`.
User defined exceptions are checked exceptions. They inherit (indirectly) from **php__lang__Exception** via **org__omg__CORBA__UserException** which itself extends **IDLEntity**.

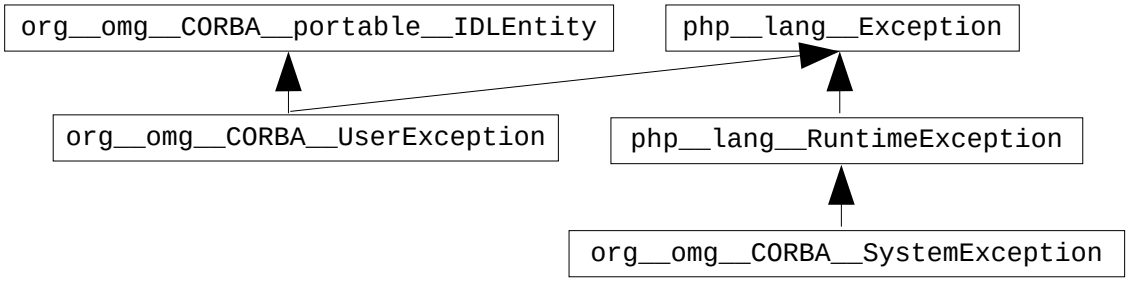

Figure 1-1 Inheritance of PHP Exception Classes

1.15.1 User Defined Exceptions

User defined exceptions are mapped to final PHP classes that extend **org__omg__CORBA__UserException** and have a "full" constructor (described below). They are otherwise mapped just like the IDL **struct** type, including the generation of Helper class.

The PHP generated exception class has a “full” constructor that has an initial string reason parameter which is concatenated to the id before calling the base **UserException** constructor.
If the exception is defined within a nested IDL scope (essentially within an interface), then its PHP class name is prefixed with a special scope. See Section 1.17, “Mapping for Certain Nested Types,” on page 1-65 for more details. Otherwise its PHP class name is prefixed with the name that corresponds to the exception’s enclosing IDL module.

The definition of the class is as follows:

```
// PHP
abstract class org__omg__CORBA__UserException
   extends php__lang__Exception
   implements org__omg__CORBA__portable__IDLEntity
{
   public function __construct($value=null) {
      parent::__construct($value);
   }
}
```

1.15.1.1 Example

```
// IDL
module Example {
   exception ex1 {long reason_code;};
};

// Generated PHP
final class ex1 extends org__omg__CORBA__UserException
{
   public $reason_code; // instance
   public function __construct($reason = null,
      $reason_code = null)
   { // default constructor
      $id = ex1Helper::id();
      if ($reason !== null) {
         $id .= " $reason";
      }
      parent::__construct($id);
      if ($reason_code !== null) {
         $this->reason_code = $reason_code;
      }
   }
}
```

1.15.1.2 Unknown User Exception

There is one standard user exception, the unknown user exception. Because the ORB does not know how to create user exceptions, it wraps the user exception as an UnknownUserException and passes it out to the DII layer. The exception is specified as follows:

```
final class org__omg__CORBA__UnknownUserException
   extends org__omg__CORBA__UserException {
   public $except;
   public function __construct(org__omg__CORBA__Any $a = null)
   {
      parent::__construct();
      $this->except = $a;
   }

}
```

In addition several exceptions that are PIDL are also mapped into user exceptions. See

Section 1.19.2, “Certain Exceptions,” on page 1-67 for more details.

1.15.2 System Exceptions

The standard IDL system exceptions are mapped to final PHP classes that extend **org__omg__CORBA__SystemException** and provide access to the IDL major and minor exception code, as well as a string describing the reason for the exception. Note there are no public constructors for **org__omg__CORBA__SystemException**; only classes that extend it can be instantiated.

A Helper class is provided for each concrete system exception. In addition, a Helper class is provided for **org__omg__CORBA__SystemException** which can be used to manipulate system exceptions when the concrete type is unknown. The Helper classes for system exceptions follow the normal rules defined in Section 1.5, “Helpers,” on page 1-13.

When a System Exception is marshaled, its GIOP Reply message shall include an associated ExceptionDetailMessage service context. The callee’s stack trace is often very valuable debugging information but may contain sensitive or unwanted information. The wstring within the service context will therefore contain additional information relating to the exception, for example the result of calling either printStackTrace (php__io__PrintWriter) or getMessage() on the exception. When unmarshaling a System Exception on the client side, the wstring from any ExceptionDetailMessage service context shall become the PHP error message in the unmarshaled exception object.

The PHP class name for each standard IDL exception is the same as its IDL name and is prefixed with **org__omg__CORBA** package. The constructor supplies default values for the three parameters: 0 for the minor code, COMPLETED_NO for the completion code, and “” for the reason string. The mapping from IDL name to PHP class name is listed in the table below.

Table 1-2 Mapping of IDL Standard Exceptions

IDL Exception	PHP Class Name
CORBA::UNKNOWN	`org__omg__CORBA__UNKNOWN`
CORBA::BAD_PARAM	`org__omg__CORBA__BAD_PARAM`
CORBA::NO_MEMORY	`org__omg__CORBA__NO_MEMORY`
CORBA::IMP_LIMIT	`org__omg__CORBA__IMP_LIMIT`
CORBA::COMM_FAILURE	`org__omg__CORBA__COMM_FAILURE`
CORBA::INV_OBJREF	`org__omg__CORBA__INV_OBJREF`
CORBA::NO_PERMISSION	`org__omg__CORBA__NO_PERMISSION`
CORBA::INTERNAL	`org__omg__CORBA__INTERNAL`
CORBA::MARSHAL	`org__omg__CORBA__MARSHAL`
CORBA::INITIALIZE	`org__omg__CORBA__INITIALIZE`
CORBA::NO_IMPLEMENT	`org__omg__CORBA__NO_IMPLEMENT`
CORBA::BAD_TYPECODE	`org__omg__CORBA__BAD_TYPECODE`
CORBA::BAD_OPERATION	`org__omg__CORBA__BAD_OPERATION`
CORBA::NO_RESOURCES	`org__omg__CORBA__NO_RESOURCES`
CORBA::NO_RESPONSE	`org__omg__CORBA__NO_RESPONSE`
CORBA::PERSIST_STORE	`org__omg__CORBA__PERSIST_STORE`
CORBA::BAD_INV_ORDER	`org__omg__CORBA__BAD_INV_ORDER`
CORBA::TRANSIENT	`org__omg__CORBA__TRANSIENT`

IDL Exception	PHP Class Name
CORBA__FREE_MEM	org__omg__CORBA__FREE_MEM
CORBA__INV_IDENT	org__omg__CORBA__INV_IDENT
CORBA__INV_FLAG	org__omg__CORBA__INV_FLAG
CORBA__INTF_REPOS	org__omg__CORBA__INTF_REPOS
CORBA__BAD_CONTEXT	org__omg__CORBA__BAD_CONTEXT
CORBA__OBJ_ADAPTER	org__omg__CORBA__OBJ_ADAPTER
CORBA__DATA_CONVERSION	org__omg__CORBA__DATA_CONVERSION
CORBA__OBJECT_NOT_EXIST	org__omg__CORBA__Object_NOT_EXIST
CORBA__TRANSACTION_REQUIRED	org__omg__CORBA__TRANSACTION_REQUIRED
CORBA__TRANSACTION_ROLLEDBACK	org__omg__CORBA__TRANSACTION_ROLLEDBACK
CORBA__INVALID_TRANSACTION	org__omg__CORBA__INVALID_TRANSACTION
CORBA__INV_POLICY	org__omg__CORBA__INV_POLICY
CORBA__CODESET_INCOMPATIBLE	org__omg__CORBA__CODESET_INCOMPATIBLE
CORBA__TRANSACTION_MODE	org__omg__CORBA__TRANSACTION_MODE
CORBA__TRANSACTION_UNAVAILABLE	org__omg__CORBA__TRANSACTION_UNAVAILABLE
CORBA__REBIND	org__omg__CORBA__REBIND
CORBA__TIMEOUT	org__omg__CORBA__TIMEOUT
CORBA__BAD_QOS	org__omg__CORBA__BAD_QOS

The definitions of the relevant classes are specified below.

```
final class org__omg__CORBA__CompletionStatus
   implements org__omg__CORBA__portable__IDLEntity {
   // Completion Status constants
   const _COMPLETED_YES = 0;
   const _COMPLETED_NO = 1;
   const _COMPLETED_MAYBE = 2;
   public static $COMPLETED_YES = null;
   public static $COMPLETED_NO = null;
   public static $COMPLETED_MAYBE = null;

   public function value() {...}
   public static final function from_int($i)
   {
      ...
   }
   private function __construct($_value)
   {
      ...
   }
}
abstract class org__omg__CORBA__SystemException
   extends php__lang__RuntimeException
{
   public $minor;
   public $completed;
   // constructor
   protected function __construct($reason, $minor,
      org__omg__CORBA__CompletionStatus $completed)
   {
      parent::__construct($reason);
      $this->minor = $minor;
      $this->completed = $completed;
   }
}
```

```
final class org__omg__CORBA__UNKNOWN
   extends org__omg__CORBA__SystemException
{
   public function __construct($reason='', $minor,
      org__omg__CORBA__CompletionStatus $completed=null)...
}
...
// there is a similar definition for each of the standard
// IDL system exceptions listed in the table above
```

1.16 Mapping for the Any Type

The IDL type **Any** maps to the PHP class **org__omg__CORBA__Any** which extends **IDLEntity**. This class has all the necessary methods to insert and extract instances of predefined types. If the extraction operations have a mismatched type, the CORBA::BAD_OPERATION exception is raised.
The **Any** class has an associated helper class. Its name is the name of the implementation class concatenated with **Helper**.
Insert and extract methods are defined in order to provide a high speed interface for use by portable stubs and skeletons. An insert and extract method are defined for each primitive IDL type, as well as for a generic streamable to handle the case of nonprimitive IDL types. Note that to preserve unsigned type information, unsigned methods are defined where appropriate.
The insert operations set the specified value and reset the **any**'s type if necessary.

The insert and extract methods for **Streamables** implement reference semantics. For the streamable IDL types, an **Any** is a container in which the data is inserted and held.
The **Any** does not copy or preserve the state of the streamable object that it holds when the insert method is invoked. The contents of the **Any** are not serialized until the **write_value()** method is invoked, or the **create_input_stream()** method is invoked. Invoking **create_output_stream()** and writing to the **Any**, or calling **read_value()**, will update the state of the last streamable object that was inserted into the **Any**, if one was previously inserted. Similarly, calling the **extract_streamable()** method multiple times will return the same contained streamable object.

The insert and extract methods for **Serializables** implement reference semantics. For a serializable type, an **Any** is a container in which the data is inserted and held. The **Any** does not copy or preserve the state of the serializable object that it holds when the insert method is invoked. The contents of the **Any** are not serialized until the **write_value()** method is invoked, or the **create_input_stream()** method is invoked. Invoking **create_output_stream()** and writing to the **Any**, or calling **read_value()**, will update the state of the last serializable object that was inserted into the **Any**, if one was previously inserted. Similarly, calling the **extract_Value()** method multiple times will return the same contained serializable object.

An object reference can be inserted into an "any" that has a matching tk_objref TypeCode. If the object reference inherits from an abstract interface, then it can also be inserted into an "any" that has a matching tk_abstract_interface TypeCode. A nil object reference can be inserted into any "any" that has a tk_objref or tk_abstract_interface TypeCode.

A valuetype can be inserted into an "any" that has a matching tk_value or tk_value_box TypeCode. If the valuetype supports an abstract interface, then it can also be inserted into an "any" that has a matching tk_abstract_interface TypeCode. A null valuetype can be inserted into any "any" that has a tk_value, tk_value_box, or tk_abstract_interface TypeCode.

The extract_Object operation can be used to extract an object reference (including a nil object reference) from an "any" that has a TypeCode of tk_objref. It can also be used to extract an object reference (including a nil object reference) from an "any" that has a

TypeCode of tk_abstract_interface and a boolean discriminator of true.

The extract_Value operation can be used to extract a valuetype (including a null valuetype) from an "any" that has a TypeCode of tk_value or tk_value_box. It can also be used to extract a valuetype (including a null valuetype) from an "any" that has a TypeCode of tk_abstract_interface and a boolean discriminator of false.

Setting the typecode via the **type()** accessor wipes out the value. An attempt to extract before the value is set will result in a CORBA__BAD_OPERATION exception being raised. This operation is provided primarily so that the type may be set properly for IDL **out** parameters.

```
abstract class org__omg__CORBA__Any
   implements org__omg__CORBA__portable__IDLEntity
{
   abstract public function equal(org__omg__CORBA__Any $a);
   // type code accessors
   abstract public function get_type();
   abstract public function set_type(
      org__omg__CORBA__TypeCode $t);
   // read and write values to/from streams
   // throw exception when typecode inconsistent with value
   abstract public function read_value(
      org__omg__CORBA__portable__InputStream $is,
      org__omg__CORBA__TypeCode $t);
   abstract public function write_value(
      org__omg__CORBA__portable__OutputStream $os);
   abstract public function create_output_stream();
   abstract public function create_input_stream();
   // insert and extract each primitive type
   abstract public function extract_short();
   abstract public function insert_short($s);
   abstract public function extract_long();
   abstract public function insert_long($i);
   abstract public function extract_longlong();
   abstract public function insert_longlong($l);
   abstract public function extract_ushort();
   abstract public function insert_ushort($s);
   abstract public function extract_ulong();
   abstract public function insert_ulong($i);
   abstract public function extract_ulonglong();
   abstract public function insert_ulonglong($l);
   abstract public function extract_float();
   abstract public function insert_float($f);
   abstract public function extract_double();
   abstract public function insert_double($d);
   abstract public function extract_boolean();
   abstract public function insert_boolean($b);
   abstract public function extract_char();
   abstract public function insert_char($c);
   abstract public function extract_wchar();
   abstract public function insert_wchar($c);
   abstract public function extract_octet();
   abstract public function insert_octet($b);
   abstract public function extract_any() ;
   abstract public function insert_any(org__omg__CORBA__Any $a);
   abstract public function extract_Object();
   abstract public function insert_Object(
      org__omg__CORBA__Object $obj);
   abstract public function extract_Value();
   abstract public function insert_Value(
      php__io__Serializable $v);
   abstract public function insert_Value(
      php__io__Serializable $v,
      org__omg__CORBA__TypeCode $t);
   // throw exception when typecode inconsistent with value
```

```
   abstract public function insert_Object(
      org__omg__CORBA__Object $obj,
      org__omg__CORBA__TypeCode $t);
   abstract public function extract_string() ;
   abstract public function insert_string($s);
   abstract public function extract_wstring() ;
   abstract public function insert_wstring($s);
   // insert and extract typecode
   abstract public function extract_TypeCode();
   abstract public function insert_TypeCode(
      org__omg__CORBA__TypeCode $t);

   // Deprecated - insert and extract Principal
   /**
   *@ deprecated
   */
   public function extract_Principal()
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   /**
   *@ deprecated
   */
   public function insert_Principal(org__omg__CORBA__Principal
$p)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   // insert and extract non-primitive IDL types
   // BAD_INV_ORDER if any doesn't hold a streamable
   public function extract_Streamable()
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function insert_Streamable(
      org__omg__CORBA__portable__Streamable $s)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   // insert and extract fixed
   public function extract_fixed()
   {
      throw org__omg__CORBA__NO_IMPLEMENT();
   }
   public function insert_fixed(php__math__BigDecimal $value)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function insert_fixed(php__math__BigDecimal $value,
      org__omg__CORBA__TypeCode $type) {
      throw new org__omg__CORBA__NO_IMPLEMENT();
}
```

Attempting to insert a **native** type into an **Any** using the **insert_Streamable** method results in a CORBA::MARSHAL exception being raised.

Note – This requires an extra test on every insert (in order to check for native). Say: A CORBA::MARSHAL exception is raised if an attempt is made to marshal an any that contains a native type. (A conforming implementation may choose to raise the exception "early" when the native is first inserted.)

1.17 Mapping for Certain Nested Types

IDL allows type declarations nested within interfaces. PHP does not allow classes to be nested within interfaces even classes nested within classes. Hence those IDL types that map to PHP classes and that are declared within the scope of an interface must be

prefixed with a special “scope” when mapped to PHP. For consistency, the “scope” package is also used for those IDL type declarations nested within valuetypes, structs, unions, and exceptions.
IDL types that contain these type declarations will generate a scope to contain the mapped PHP class declarations. The scope name is constructed by appending **Package** to the IDL type name.

1.17.1 Example

```
// IDL
module Example {
   interface Foo {
      exception e1 {};
   };
};

// generated PHP
package;
final class Example__FooPackage__e1
   extends org__omg__CORBA__UserException
{
   ...
}
```

1.18 Mapping for Typedef

PHP does not have a typedef construct. Helper classes are generated for all typedefs.

1.18.1 Simple IDL types

IDL types that are mapped to simple PHP types may not be subclassed in PHP. Hence any typedefs that are type declarations for simple types are mapped to the original (mapped type) everywhere the typedef type appears.
The IDL types covered by this rule are described in Section 1.4, “Mapping for Basic Types,” on page 1-5.

1.18.2 Complex IDL Types

Typedefs for types that are neither arrays nor sequences are “unwound” to their original type until a simple IDL type or user-defined IDL type (of the non typedef variety) is encountered.

1.18.2.1 Example

```
// IDL
struct EmpName {
string firstName;
string lastName;
};
typedef EmpName EmpRec;
typedef sequence <long> IntSeq;

// generated PHP
// regular struct mapping for EmpName
// regular helper class mapping for EmpRec
// unwind the sequence
final class EmpName
   extends org__omg__CORBA__portable__IDLEntity {
   ...
}
```

```
class EmpRecHelper {
...
}
```

1.19 Mapping Pseudo Objects to PHP

1.19.1 Introduction

Pseudo objects are constructs whose definition is usually specified in IDL, but whose mapping is language specific. A pseudo object is not (usually) a regular CORBA object. Often it is exposed to either clients and/or servers as a process, or a thread, in local programming language construct.
For each of the standard IDL pseudo-objects we either specify a specific PHP language construct or we specify it as a **pseudo interface**.

1.19.1.1 Pseudo Interface

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

- Not represented in the Interface Repository.
- No helper classes are generated.
- Mapped to a PHP **abstract class** that does not extend or inherit from any other classes or interfaces.

Note – The specific definition given for each piece of PIDL may override the general guidelines above. In such a case, the specific definition takes precedence.
All of the pseudo interfaces are mapped as if they were declared in:

```
module org {
module omg {
module CORBA {
...
```

That is, they are mapped with the **org__omg__CORBA** PHP prefix.

1.19.2 Certain Exceptions

The standard CORBA PIDL uses several exceptions, Bounds, BadKind, and InvalidName.
These exceptions don't exist in the interface repository. However, so that users can treat them as “normal exceptions” for programming purposes, they are otherwise mapped as normal user exceptions, including the generation of helper classes.
They are prefixed with the scope name within they are used. A Bounds and BadKind exception are prefixed with **TypeCodePackage,** for use by **TypeCode**. A Bounds exception is prefixed with the standard CORBA module, for use by **NVList**, **ExceptionList**, and **ContextList**. An InvalidName exception is prefixed with **ORBPackage,** for use by **ORB**.

```
// PHP
final class org__omg__CORBA__Bounds
   extends org__omg__CORBA__UserException
{
   public function __construct() {...}
}

final class org__omg__CORBA__TypeCodePackage__Bounds
   extends org__omg__CORBA__UserException
{
   public function __construct() {...}
```

```
}
final class org__omg__CORBA__TypeCodePackage__BadKind
   extends org__omg__CORBA__UserException
{
   public function __construct() {...}
}
final class org__omg__CORBA__ORBPackage__InvalidName
   extends org__omg__CORBA__UserException
{
   public function __construct() {...}
}
```

1.19.3 Environment

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

```
// PHP code
abstract class org__omg__CORBA__Environment {
   public abstract function set_exception(
      php__lang__Exception $except);
   public abstract function get_exception();
   public abstract function clear();
}
```

1.19.4 NamedValue

A **NamedValue** describes a name, value pair. It is used in the DII to describe arguments and return values, and in the context routines to pass property, value pairs.
In PHP it includes a name, a value (as an **Any**), and an integer representing a set of flags.

```
// IDL
typedef unsigned long Flags;
typedef string Identifier;
const Flags ARG_IN = 1;
const Flags ARG_OUT = 2;
const Flags ARG_INOUT = 3;
const Flags CTX_RESTRICT_SCOPE = 15;
pseudo interface NamedValue {
readonly attribute Identifier name;
readonly attribute any value;
readonly attribute Flags flags;
};
```

```
// PHP
interface org__omg__CORBA__ARG_IN
{
   const value = 1;
}
interface org__omg__CORBA__ARG_OUT
{
   const value = 2;
}
interface org__omg__CORBA__ARG_INOUT {
   const value = 3;
}
interface org__omg__CORBA__CTX_RESTRICT_SCOPE {
   const value = 15;
}
abstract class NamedValue {
   public abstract function name();
   public abstract function value();
   public abstract function flags();
```

```
}
```

1.19.5 NVList

An **NVList** is used in the DII to describe arguments, and in the context routines to describe context values.
In PHP it maintains a modifiable list of **NamedValue**s.

```
// IDL
pseudo interface NVList {
    readonly attribute unsigned long count;
    NamedValue add(in Flags flags);
    NamedValue add_item(in Identifier item_name, in Flags flags);
    NamedValue add_value(
        in Identifier item_name,
        in any val,
        in Flags flags);
    NamedValue item(in unsigned long index) raises (CORBA__Bounds);
    void remove(in unsigned long index) raises (CORBA__Bounds);
};
```

```
// PHP
abstract class org__omg__CORBA__NVList
{
   public abstract function count();
   public abstract function add($flags);
   public abstract function add_item(
      $item_name, $flags);
   public abstract function add_value(
      $item_name, org__omg__CORBA__Any $val, $flags);
   public abstract function item($index);
   public abstract function remove($index);
}
```

1.19.6 ExceptionList

An **ExceptionList** is used in the DII to describe the exceptions that can be raised by IDL operations.
It maintains a list of modifiable list of **TypeCode**s.

```
// IDL
pseudo interface ExceptionList {
    readonly attribute unsigned long count;
    void add(in TypeCode exc);
    TypeCode item (in unsigned long index) raises (CORBA__Bounds);
    void remove (in unsigned long index) raises (CORBA__Bounds);
};
```

```
// PHP
abstract class org__omg__CORBA__ExceptionList {
   public abstract function count();
   public abstract function add(org__omg__CORBA__TypeCode $exc);
   public abstract function item($index);
   public abstract funciton remove($index);
}
```

1.19.7 Context

A **Context** is used in the DII to specify a context in which context strings must be resolved before being sent along with the request invocation.

```
// IDL
```

```
pseudo interface Context {
    readonly attribute Identifier context_name;
    readonly attribute Context parent;
    Context create_child(in Identifier child_ctx_name);
    void set_one_value(in Identifier propname, in any propvalue);
    void set_values(in NVList values);
    void delete_values(in Identifier propname);
    NVList get_values(
    in Identifier start_scope,
    in Flags op_flags,
    in Identifier pattern);
};
pseudo interface ContextList {
    readonly attribute unsigned long count;
    void add(in string ctx);
    string item(in unsigned long index) raises (CORBA::Bounds);
    void remove(in unsigned long index) raises (CORBA::Bounds);
};

// PHP
abstract class Context {
   public abstract function context_name();
   public abstract function parent();
   public abstract function create_child($child_ctx_name);
   public abstract function set_one_value($propname,
      org__omg__CORBA__Any $propvalue);
   public abstract function set_values(
      org__omg__CORBA__NVList $values);
   public abstract function delete_values($propname);
   public abstract function get_values($start_scope,
      $op_flags,
      $pattern);
}
abstract class ContextList {
   public abstract function count();
   public abstract function add($ctx);
   public abstract function item($index);
   public abstract function remove($index);
}
```

1.19.8 Request

A **Request** is used in the DII to describe an invocation.

```
// IDL
pseudo interface Request {
    readonly attribute Object target;
    readonly attribute Identifier operation;
    readonly attribute NVList arguments;
    readonly attribute NamedValue result;
    readonly attribute Environment env;
    readonly attribute ExceptionList exceptions;
    readonly attribute ContextList contexts;
    attribute Context ctx;
    any add_in_arg();
    any add_named_in_arg(in string name);
    any add_inout_arg();
    any add_named_inout_arg(in string name);
    any add_out_arg();
    any add_named_out_arg(in string name);
    void set_return_type(in TypeCode tc);
    any return_value();
    void invoke();
    void send_oneway();
```

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

// PHP
abstract class org__omg__CORBA__Request {
   public abstract function target();
   public abstract function operation();
   public abstract function arguments();
   public abstract function result();
   public abstract function env();
   public abstract function exceptions();
   public abstract function contexts();
   public abstract function ctx();
   public abstract function ctx(org__omg__CORBA__Context $c);
   public abstract function add_in_arg();
   public abstract function add_named_in_arg($name);
   public abstract function add_inout_arg();
   public abstract function add_named_inout_arg($name);
   public abstract function add_out_arg();
   public abstract function add_named_out_arg($name);
   public abstract function set_return_type(
      org__omg__CORBA__TypeCode $tc);
   public abstract function return_value();
   public abstract function invoke();
   public abstract function send_oneway();
   public abstract function send_deferred();
   public abstract function get_response();
   public abstract boolean poll_response();
}
```

It is permissible to call the **return_value()** method before issuing the **Request** (e.g., before calling **invoke()**, **send_oneway()**, or **send_deferred()**).
Changes made to the **Any** that stores the result may be used by the implementation to improve performance. For example, one may insert a **Streamable** into the **Any** containing the return value before invoking the **Request**. Because **Any**s provide reference semantics, the result will be marshaled directly into the **Streamable** object avoiding additional marshaling if the **Any** were extracted after invocation.

1.19.9 TypeCode

The deprecated **parameter** and **param_count** methods are not mapped. The Typecode has a helper class.
The helper class has the same prefix as the implementation class for TypeCode. Its name is the name of the implementation class concatenated with **Helper**.

```
// IDL
module CORBA {
    enum TCKind {
        tk_null, tk_void,
        tk_short, tk_long, tk_ushort, tk_ulong,
        tk_float, tk_double, tk_boolean, tk_char,
        tk_octet, tk_any, tk_TypeCode, tk_Principal, tk_objref,
        tk_struct, tk_union, tk_enum, tk_string,
        tk_sequence, tk_array, tk_alias, tk_except,
        tk_longlong, tk_ulonglong, tk_longdouble,
        tk_wchar, tk_wstring, tk_fixed,
        tk_value, tk_value_box,
        tk_native,
        tk_abstract_interface,
        tk_local_interface
    };
    typedef short ValueModifier;
```

```
    const ValueModifier VM_NONE = 0;
    const ValueModifier VM_CUSTOM = 1;
    const ValueModifier VM_ABSTRACT = 2;
    const ValueModifier VM_TRUNCATABLE = 3;
    typedef short Visibility;
    const Visibility PRIVATE_MEMBER = 0;
    const Visibility PUBLIC_MEMBER = 1;
};

// PHP
class org__omg__CORBA__TCKind {
   const _tk_null = 0;
   public static $tk_null = null;
   const _tk_void = 1;
   public static $tk_void = null;
   const _tk_short = 2;
   public static $tk_short = null;
   const _tk_long = 3;
   public static $tk_long = null;
   const _tk_ushort = 4;
   public static $tk_ushort = null;
   const _tk_ulong = 5;
   public static $tk_ulong =  null;
   const _tk_float = 6;
   public static $tk_float = null;
   const _tk_double = 7;
   public static $tk_double = null;
   const _tk_boolean = 8;
   public static $tk_boolean = null;
   const _tk_char = 9;
   public static $tk_char = null;
   const _tk_octet = 10;
   public static $tk_octet = null;
   const _tk_any = 11;
   public static $tk_any = null;
   const _tk_TypeCode = 12;
   public static $tk_TypeCode = null;
   const _tk_Principal = 13;
   public static $tk_Principal = null;
   const _tk_objref = 14;
   public static $tk_objref = null;
   const _tk_stuct = 15;
   public static $tk_stuct = null;
   const _tk_union = 16;
   public static $_union = null;
   const _tk_enum = 17;
   public static $tk_enum = null;
   const _tk_string = 18;
   public static $tk_string = null;
   const _tk_sequence = 19;
   public static $tk_sequence = null;
   const _tk_array = 20;
   public static $tk_array = null;
   const _tk_alias = 21;
   public static $tk_alias = null;
   const _tk_except = 22;
   public static $tk_except = null;
   const _tk_longlong = 23;
   public static $tk_longlong = null;
   const _tk_ulonglong = 24;
   public static $tk_ulonglong = null;
   const _tk_longdouble = 25;
   public static $tk_longdouble = null;
   const _tk_wchar = 26;
   public static $tk_wchar = null;
   const _tk_wstring = 27;
   public static $tk_wstring = null;
```

```
    const _tk_fixed = 28;
    public static $tk_fixed = null;
    const _tk_value = 29;
    public static $tk_value = null;
    const _tk_value_box = 30;
    public static $tk_value_box = null;
    const _tk_native = 31;
    public static $tk_native = null;
    const _tk_abstract_interface = 32;
    public static $tk_abstract_interface = null;
    const _tk_local_interface = 33;
    public static $tk_local_interface = null;

    public function value() {...}
    public static function from_int($value) {...}
    protected function __construct($value) {...}
    public static function initialize() {
       self::$tk_null = new self(self::_tk_null);
       self::$tk_void = new self(self::_tk_void);
       self::$tk_short = new self(self::_tk_short);
       self::$tk_long = new self(self::_tk_long);
       self::$tk_ushort = new self(self::_tk_ushort);
       self::$tk_ulong = new self(self::_tk_ulong);
       self::$tk_float = new self(self::_tk_float);
       self::$tk_double = new self(self::_tk_double);
       self::$tk_boolean = new self(self::_tk_boolean);
       self::$tk_char = new self(self::_tk_char);
       self::$tk_octet = new self(self::_tk_octet);
       self::$tk_any = new self(self::_tk_any);
       self::$tk_TypeCode = new self(self::_tk_TypeCode);
       self::$tk_Principal = new self(self::_tk_Principal);
       self::$tk_objref = new self(self::_tk_objref);
       self::$tk_struct = new self(self::_tk_struct);
       self::$tk_union = new self(self::_tk_union);
       self::$tk_enum = new self(self::_tk_enum);
       self::$tk_string = new self(self::_tk_string);
       self::$tk_sequence = new self(self::_tk_sequence);
       self::$tk_array = new self(self::_tk_array);
       self::$tk_alias = new self(self::_tk_alias);
       self::$tk_except = new self(self::_tk_except);
       self::$tk_longlong = new self(self::_tk_longlong);
       self::$tk_ulonglong = new self(self::_tk_ulonglong);
       self::$tk_longdouble = new self(self::_tk_longdouble);
       self::$tk_wchar = new self(self::_tk_wchar);
       self::$tk_wstring = new self(self::_tk_wstring);
       self::$tk_fixed = new self(self::_tk_fixed);
       self::$tk_value = new self(self::_tk_value);
       self::$tk_value_box = new self(self::_tk_value_box);
       self::$tk_native = new self(self::_tk_native);
       self::$tk_abstract_interface =
          new self(self::_tk_abstract_interface);
       self::$tk_local_interface =
          new self(self::_tk_local_interface);
       self::$tk_char = new self(self::_tk_char);
    }
}
org__omg__CORBA__TCKind::__initialize();

interface org__omg__CORBA__VM_NONE {
   const value = 0;
}
interface org__omg__CORBA__VM_CUSTOM {
   const value = 1;
}
interface org__omg__CORBA__VM_ABSTRACT {
   const value = 2;
}
interface org__omg__CORBA__VM_TRUNCATABLE {
```

```
   const value = 3;
}
interface org__omg__CORBA__PRIVATE_MEMBER {
   const value = 0;
}
interface org__omg__CORBA__PUBLIC_MEMBER {
   const value = 1;
}

//IDL
pseudo interface TypeCode {
   exception Bounds {};
   exception BadKind {};
   // for all TypeCode kinds
   boolean equal(in TypeCode tc);
   boolean equivalent(in TypeCode tc);
   TypeCode get_compact_typecode();
   TCKind kind();
   // for objref, struct, union, enum, alias, value, valuebox,
   // native, abstract_interface, and except
   RepositoryID id() raises (BadKind);
   Identifier name() raises (BadKind);
   // for struct, union, enum, value, and except
   unsigned long member_count() raises (BadKind);
   Identifier member_name(in unsigned long index)
      raises (BadKind, Bounds);
   // for struct, union, value, and except
   TypeCode member_type(in unsigned long index)
      raises (BadKind, Bounds);
   // for union
   any member_label(in unsigned long index) raises (BadKind, Bounds);
   TypeCode discriminator_type() raises (BadKind);
   long default_index() raises (BadKind);
   // for string, sequence, and array
   unsigned long length() raises (BadKind);
   // for sequence, array, value, value_box and alias
   TypeCode content_type() raises (BadKind);
   // for fixed
   unsigned short fixed_digits() raises(BadKind);
   short fixed_Scale() raised (BadKind);
   // for value
   Visibility member_visibility(in unsigned long index)
      raises (BadKind, Bounds);
   ValueModifier type_modifier() raises (BadKind);
   TypeCode concrete_base_type() raises (BadKind);
}

// PHP
abstract class org__omg__CORBA__TypeCode implements
   org__omg__CORBA__portable__IDLEntity
{
   // for all TypeCode kinds
   public abstract function equal(org__omg__CORBA__TypeCode $tc);
   public abstract function equivalent(
      org__omg__CORBA__TypeCode $tc);
   public abstract function get_compact_typecode();
   public abstract TCKind kind();
   // for objref, struct, union, enum, alias,
   // value, value_box, native,
   // abstract_interface, and except
   public abstract function id();
   public abstract function name();
   // for struct, union, enum, value, and except
   public abstract function member_count();
   public abstract function member_name($index);
   // for struct, union, value, and except
```

```
    public abstract function member_type($index);
    // for union
    public abstract function member_label($index);
    public abstract function discriminator_type();
    public abstract function default_index();
    // for string, sequence, and array
    public abstract function length();
    // for sequence, array, value, value_box and alias
    public abstract function content_type();
    // for fixed
    public abstract function fixed_digits();
    public abstract function fixed_Scale();
    // for value
    public abstract function member_visibility($index);
    public abstract function type_modifer();
    public abstract function concrete_base_type();
}
```

1.19.10 ORB

The **ORB** defines operations that are implemented by the ORB core and are in general not dependent upon a particular object or object adapter.
In addition to the operations specifically defined on the ORB in the core, additional methods needed specifically for PHP are also defined.
The **UnionMemberSeq**, **EnumMemberSeq**, **StructMemberSeq, ValueMemberSeq** typedefs are real IDL and bring in the Interface Repository (see *The Common Object Request Broker: Architecture and Specification*, *Interface Repository* chapter). The **ServiceInformation** struct is real IDL and is defined in *The Common Object Request Broker: Architecture and Specification*, *ORB Interface* chapter. Rather than tediously listing interfaces, and other assorted types, suffice is to say that these constructs are all mapped following the rules for IDL set forth in this specification.

```
// IDL
pseudo interface ORB { // PIDL
    typedef string ObjectId;
    typedef sequence <ObjectId> ObjectIdList;
    ORBId id() ;
    exception InconsistentTypeCode {};
    exception InvalidName {};
    string object_to_string (
        in Object obj
    );
    Object string_to_object (
        in string str
    );
    // Dynamic Invocation related operations
    void create_list (
        in long count,
        out NVList new_list
    );
    void create_operation_list (
        in OperationDef oper,
        out NVList new_list
    );
    void get_default_context (
        out Context ctx
    );
    void send_multiple_requests_oneway(in RequestSeq req);
    void send_multiple_requests_deferred(in RequestSeq req);
    boolean poll_next_response();
    void get_next_response(out Request req);
    // Service information operations
    boolean get_service_information (
        in ServiceType service_type,
```

```
    out ServiceInformation service_information
);
ObjectIdList list_initial_services ();
void register_initial_reference(
    in ObjectId id,
    in Object obj
) raises (InvalidName) ;
// Initial reference operation
Object resolve_initial_references (
    in ObjectId identifier
) raises (InvalidName);
// Type code creation operations
TypeCode create_struct_tc (
    in RepositoryId id,
    in Identifier name,
    in StructMemberSeq members
);
TypeCode create_union_tc (
    in RepositoryId id,
    in Identifier name,
    in TypeCode discriminator_type,
    in UnionMemberSeq members
);
TypeCode create_enum_tc (
    in RepositoryId id,
    in Identifier name,
    in EnumMemberSeq members
);
TypeCode create_alias_tc (
    in RepositoryId id,
    in Identifier name,
    in TypeCode original_type
);
TypeCode create_exception_tc (
    in RepositoryId id,
    in Identifier name,
    in StructMemberSeq members
);
TypeCode create_interface_tc (
    in RepositoryId id,
    in Identifier name
);
TypeCode create_string_tc (
    in unsigned long bound
);
TypeCode create_wstring_tc (
    in unsigned long bound
);
TypeCode create_fixed_tc (
    in unsigned short digits,
    in short scale
);
TypeCode create_sequence_tc (
    in unsigned long bound,
    in TypeCode element type
);
TypeCode create_recursive_sequence_tc ( // deprecated
    in unsigned long bound,
    in unsigned long offset
);
TypeCode create_array_tc (
    in unsigned long length,
    in TypeCode element_type
);
TypeCode create_value_tc (
```

```
    in RepositoryId id,
    in Identifier name,
    in ValueModifier type_modifier,
    in TypeCode concrete_base,
    in ValueMemberSeq members
);
TypeCode create_value_box_tc (
    in RepositoryId id,
    in Identifier name,
    in TypeCode boxed_type
);
TypeCode create_native_tc (
    in RepositoryId id,
    in Identifier name
);
TypeCode create_recursive_tc (
    in RepositoryId id
);
TypeCode create_abstract_interface_tc (
    in RepositoryId id,
    in Identifier name
);
TypeCode create_local_interface_tc (
    in RepositoryId id,
    in Identifier name
);
// Thread related operations
boolean work_pending();
void perform_work();
void run();
void shutdown(
    in boolean wait_for_completion
);
void destroy();
// Policy related operations
Policy create_policy(
    in PolicyType type,
    in any val
) raises (PolicyError);
// Dynamic Any related operations deprecated and removed
// from primary list of ORB operations
// Value factory operations
ValueFactory register_value_factory(
    in RepositoryId id,
    in ValueFactory factory
);
void unregister_value_factory(in RepositoryId id);
ValueFactory lookup_value_factory(in RepositoryId id);
// Additional operations that only appear in the PHP mapping
TypeCode get_primitive_tc(in TCKind tcKind);
ExceptionList create_exception_list();
ContextList create_context_list();
Environment create_environment();
Current get_current();
Any create_any();
OutputStream create_output_stream();
void connect(Object obj);
void disconnect(Object obj);
Object get_value_def(in String repid);
void set_delegate(Object wrapper);
// additional methods for ORB initialization go here, but only
// appear in the mapped PHP
// (see Section 1.21.9, "ORB Initialization)
// public static function init($args, $props);
// protected abstract function set_parameters($args, $props);
```

};

All types defined in this chapter are either part of the CORBA or the CORBA_2_3 module. When referenced in OMG IDL, the type names must be prefixed by "**CORBA::**" or "**CORBA_2_3::**".

```
// PHP
abstract class org__omg__CORBA__ORB {
   public function id()
   {
      throw new org__omg__CORBA__NO_IMPLEMENT() ;
   }
   public abstract function string_to_object($str);
   public abstract function object_to_string(
      org__omg__CORBA__Object $obj);
   // Dynamic Invocation related operations
   public abstract function create_list($count);
   /**
   *@deprecated Deprecated by CORBA 2.3.
   */
   public abstract function create_operation_list(
      org__omg__CORBA__OperationDef $oper);
   // oper must really be an OperationDef

   public abstract function create_named_value(
      $name, org__omg__CORBA__Any $value, $flags);
   public abstract function create_exception_list();
   public abstract function create_context_list();
   public abstract function get_default_context();
   public abstract function create_environment();
   public abstract function send_multiple_requests_oneway($req);
   public abstract function send_multiple_requests_deferred
($req);
   public abstract function poll_next_response();
   public abstract function get_next_response();
   // Service information operations
   public function get_service_information(
      $service_type,
      org__omg__CORBA__ServiceInformation &$service_info)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public abstract function list_initial_services();
   public function register_initial_reference(
      $object_name,
      org__omg__CORBA__Object $object)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT() ;
   }
   // Initial reference operation
   public abstract function resolve_initial_references(
      $object_name);
   // typecode creation
   public abstract function create_struct_tc($id,
$name,$members);
   public abstract function create_union_tc($id, $name,
      org__omg__CORBA__TypeCode $discriminator_type,$members);
   public abstract function create_enum_tc($id, $name, $members);
   public abstract function create_alias_tc($id, $name,
      org__omg__CORBA__TypeCode $original_type);
   public abstract function create_exception_tc(
      $id, $name,$members);
   public abstract function create_interface_tc($id, $name);
   public abstract function create_string_tc($bound);
   public abstract function create_wstring_tc($bound);
   public abstract create_fixed_tc($digits, $scale)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
```

```
}
public abstract function create_sequence_tc($bound,
   org__omg__CORBA__TypeCode $element_type);
/**
*@deprecated Deprecated by CORBA 2.3.
*/
public abstract function create_recursive_sequence_tc(
   $bound, $offset);
public abstract function create_array_tc(
   $length, org__omg__CORBA__TypeCode $element_type);
public function create_value_tc($id, $name, $type_modifier,
   org__omg__CORBA__TypeCode $concrete_base, $members)
{
   throw new org__omg__CORBA__NO_IMPLEMENT();
}
public function create_value_box_tc(
   $id, $name, org__omg__CORBA__TypeCode $boxed_type)
{
   throw new org__omg__CORBA__NO_IMPLEMENT();
}
public function create_native_tc($id, $name)
{
   throw new org__omg__CORBA__NO_IMPLEMENT();
}
public function create_recursive_tc($id)
{
   throw new org__omg__CORBA__NO_IMPLEMENT();
}
public function create_abstract_interface_tc($id, $name)
{
   throw new org__omg__CORBA__NO_IMPLEMENT();
}
public function create_local_interface_tc($id, $name)
{
   throw org__omg__CORBA__NO_IMPLEMENT();
}
/**
*@deprecated Deprecated by CORBA 2.2.
*/
public function get_current() {
   throw new org__omg__CORBA__NO_IMPLEMENT();
}
/**
*@deprecated Deprecated by Portable Object Adapter,
see OMG document orbos/98-01-06 for details.
*/
public function connect( org__omg__CORBA__Object $obj)
{
   throw new org__omg__CORBA__NO_IMPLEMENT();
}
/**
*@deprecated Deprecated by Portable Object Adapter,
*see OMG document orbos/98-01-06 for details.
*/
public function disconnect( org__omg__CORBA__Object $obj)
{
   throw new org__omg__CORBA__NO_IMPLEMENT();
}
// Thread related operations
public function work_pending() {
   throw new org__omg__CORBA__NO_IMPLEMENT();
}
public function perform_work() {
   throw new org__omg__CORBA__NO_IMPLEMENT();
}
public function run() {
   throw new org__omg__CORBA__NO_IMPLEMENT();
}
```

```
    public function shutdown($wait_for_completion)
    {
      throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function destroy() {
      throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    // Policy related operations
    public function create_policy($policy_type,
      org__omg__CORBA__Any $val)
    {
      throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    // additional methods for IDL/PHP mapping
    public abstract function get_primitive_tc(
      org__omg__CORBA__TCKind $tcKind);
    public abstract function create_any();
    public abstract function create_output_stream();
    // additional static methods for ORB initialization
    public static function init($args = null, $props = null);
    protected abstract function set_parameters($arg, $props);
    public function get_value_def( $repid )
    {
      throw new org__omg__CORBA__NO_IMPLEMENT() ;
    }
    public function register_value_factory(
      $id, org__omg__CORBA__portable__ValueFactory $factory)
    {
      throw new org__omg__CORBA__NO_IMPLEMENT() ;
    }
    public function unregister_value_factory( $id )
    {
      throw new org__omg__CORBA__NO_IMPLEMENT() ;
    }
    public function lookup_value_factory($id)
    {
      throw new org__omg__CORBA__NO_IMPLEMENT() ;
    }

    public function set_delegate(org__omg__CORBA__Object $wrapper)
    {
      throw new org__omg__CORBA__NO_IMPLEMENT() ;
    }
  }
  abstract class org__omg__CORBA_2_3__ORB
    extends org__omg__CORBA__ORB
  {
    // always return a ValueDef or throw BAD_PARAM if
    // repid not of a value
    public function get_value_def($repid) {
      throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    // Value factory operations
    public function register_value_factory($id,
      org__omg__CORBA__portable__ValueFactory $factory)
    {
      throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function unregister_value_factory($id)
    {
      throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function lookup_value_factory($id)
    {
      throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function set_delegate(org__omg__CORBA__Object $wrapper)
    {
```

```
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
}

abstract class org__omg__CORBA_2_5__ORB
   extends org__omg__CORBA_2_3__ORB
{
   public function id()
   {
      throw new org__omg__CORBA__NO_IMPLEMENT() ;
   }
   public function register_initial_reference($object_name,
      org__omg__CORBA__Object $object)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT() ;
   }
   public function create_local_interface_tc($id,$name)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT() ;
   }
}
```

1.19.10.1 set_delegate

The **set_delegate()** method supports the PHP ORB portability interfaces by providing a method for classes that support ORB portability through delegation to set their delegate. This is typically required in cases where instances of such classes were created by the application programmer rather than the ORB runtime. The wrapper parameter is the instance of the object on which the ORB must set the delegate. The mechanism to set the delegate is specific to the class of the wrapper instance. The **set_delegate()** method supports setting delegates on instances of the following PHP classes:
org__omg__PortableServer__Servant
If the wrapper parameter is not an instance of a class for which the ORB can set the delegate, the CORBA__BAD_PARAM exception is thrown.

1.19.10.2 get_value_def

The **get_value_def()** method is declared to return an **org__omg__CORBA__Object**. However, it is intended to only be used for value types.
The actual implementation:

- raises the BAD_PARAM system exception if the specified **repid** parameter does not identify an IDL type that is a value type.
- returns a **ValueDef** if the specified **repid** parameter identifies an IDL type that is a value type.

1.19.11 CORBA::Object

The IDL **Object** type is mapped to the **org__omg__CORBA__Object** and **org__omg__CORBA__ObjectHelper** classes as shown below.
The PHP interface for each user defined IDL **interface** extends **org__omg__CORBA__Object**, so that any object reference can be passed anywhere a **org__omg__CORBA__Object** is expected.

The **Policy**, **DomainManager**, and **SetOverrideType** types are real IDL and are defined in *The Common Object Request Broker: Architecture and Specification, ORB Interface* chapter. Rather than tediously list the mapping here, suffice it to say that these constructs are all mapped following the rules for IDL set forth in this specification.

The **_get_interface()** and **_get_interface_def()** operations both return an **InterfaceDef** object that defines the runtime type of the **CORBA__Object** on which they are invoked.

The deprecated **_get_interface()** method returns the **InterfaceDef** directly as an **InterfaceDef**. Because this operation is deprecated, new applications should not use it, and it may be removed in a future release of this specification. The **_get_interface_def()** operation returns the **InterfaceDef** as a **CORBA__Object**, and the invoker of **_get_interface_def()** must narrow the **CORBA__Object** to an **InterfaceDef** in order to use it.

```
// PHP
interface org__omg__CORBA__Object {
   function _is_a($identifier);
   function _is_equivalent(org__omg__CORBA__Object $that);
   function _non_existent();
   function _hash($maximum);
   function _duplicate();
   function _release();
   /**
   *@deprecated Deprecated by CORBA 2.3.
   */
   function _get_interface();
   function _get_interface_def();
   function _request($s);
   function _create_request(
      org__omg__CORBA__Context $ctx,
      $operation,
      org__omg__CORBA__NVList $arg_list,
      org__omg__CORBA__NamedValue $result);
   function _create_request(
      org__omg__CORBA__Context $ctx,
      $operation,
      org__omg__CORBA__NVList $arg_list,
      org__omg__CORBA__NamedValue $result,
      org__omg__CORBA__ExceptionList $exclist,
      org__omg__CORBA__ContextList $ctxlist);
   function _get_policy($policy_type);
   function _get_domain_managers();
   function _set_policy_override(
      $policies, org__omg__CORBA__SetOverrideType $set_add);
}
abstract class org__omg__CORBA__ObjectHelper {
   public static function insert(
      org__omg__CORBA__Any $a,
      org__omg__CORBA__Object $t)
   {
      ...
   }
   public static function extract(org__omg__CORBA__Any $a)
   {
      ...
   }
   public static function type()
   {
      ...
   }
   public static function id()
   {
      ...
   }
   public static function read(
      org__omg__CORBA__portable__InputStream $is)
   {
      ...
   }
   public static function write(
      org__omg__CORBA__portable__OutputStream $os,
      org__omg__CORBA__Object $val)
   {
      ...
```

```
    }
}
```

1.19.12 Principal

Principal was deprecated in CORBA 2.2. No support is implemented.

1.20 Server-Side Mapping

1.20.1 Introduction

This section discusses how object implementations written in PHP create and register objects with the ORB runtime.

1.20.2 Implementing Interfaces

To define an implementation in PHP, a developer must write an implementation class. Instances of the implementation class implement IDL interfaces. The implementation class must define public methods corresponding to the operations and attributes of the IDL interface supported by the object implementation, as defined by the mapping specification for IDL interfaces. Providing these methods are sufficient to satisfy all abstract methods defined by a particular interface's skeleton class.

The mapping specifies two alternative relationships between the application-supplied implementation class and the generated class or classes for the interface. Specifically, the mapping requires support for both *inheritance-based* relationships and *delegation-based* relationships. Conforming ORB implementations shall provide both of these alternatives. Conforming applications may use either or both of these alternatives.

1.20.2.1 Mapping for Local Interface

In the PHP language mapping, the **LocalObject** class is used as a base class for implementations of a local interface. It is a class that implements all the operations in the **org__omg__CORBA__Object** interface.

```
// PHP
import org__omg__CORBA__portable__*;
abstract class org__omg__CORBA__LocalObject
   implements org__omg__CORBA__Object
{
   public function __construct()
   {}
   public function _is_equivalent(org__omg__CORBA__Object $that)
   {
      return $this->equals($that);
   }
   public function _non_existent()
   {
      return false;
   }
   public function _hash($maximum)
   {
      return $this->hashCode();
   }
   public function _ids() {
      throw new NO_IMPLEMENT() ;
   }
   public function _is_a($repositoryId)
   {
      $ids = $this->_ids();
```

```
      $l = count($ids);
      for ($i=0; $i <$l; ++$i) {
         if ($repositoryId == $ids[$i]) {
            return true;
         }
      }
      return false ;
   }
   public function _duplicate()
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function _release()
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function _request($operation)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function _create_request(
      org__omg__CORBA__Context $ctx,
      $operation,
      org__omg__CORBA__NVList $arg_list,
      org__omg__CORBA__NamedValue $result)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function _create_request(
      org__omg__CORBA__Context $ctx,
      $operation,
      org__omg__CORBA__NVList $arg_list,
      org__omg__CORBA__NamedValue $result,
      org__omg__CORBA__ExceptionList $exceptions,
      org__omg__CORBA__ContextList $contexts)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   /**
   *@deprecated Deprecated by CORBA 2.4
   */
   public function _get_interface()
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function _get_interface_def()
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function _orb()
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function _get_policy($policy_type)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function _get_domain_managers()
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function _set_policy_override(
      $policies,
      org__omg__CORBA__SetOverrideType $set_add)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function _is_local()
```

```
    {
        throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function _servant_preinvoke($operation, $expectedType)
    {
        throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function _servant_postinvoke(
        org__omg__CORBA__portable__ServantObject $servant)
    {
        throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function _request($operation, $responseExpected)
    {
        throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function _invoke(
        org__omg__CORBA__portable__OutputStream $output)
    {
        throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function _releaseReply(
        org__omg__CORBA__portable__InputStream $input)
    {
        throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function validate_connection()
    {
        throw new org__omg__CORBA__NO_IMPLEMENT();
    }
}
```

1.20.2.2 Mapping of PortableServer::Servant

The **PortableServer** module for the Portable Object Adapter (POA) defines the native **Servant** type. In PHP, the **Servant** type is mapped to the PHP **org__omg__PortableServer__Servant** class. The implementation of the **Servant** shall delegate all operations to the **org__omg__PortableServer__portable__Delegate** class defined in Section 1.21.8, “Servant Delegate”, on page 1-127.

The **Servant** class is defined as follows:

```
//PHP
abstract class org__omg__PortableServer__Servant {
    // Convenience methods for application programmer
    public final function _this_object()
    {
        return $this->_get_delegate()->this_object($this);
    }
    public final function _this_object(org__omg__CORBA__ORB $orb)
    {
        $orb->set_delegate($this);
        return $this->_this_object();
    }
    public final function _orb()
    {
        return $this->_get_delegate()->orb($this);
    }
    public final function _poa()
    {
        return $this->_get_delegate()->poa($this);
    }
    public final function _object_id()
    {
        return $this->_get_delegate()->object_id($this);
    }
```

```
   // Methods which may be overridden by the
   // application programmer
   public function _default_POA()
   {
      return $this->_get_delegate()->default_POA($this);
   }
   public function _is_a($repository_id)
   {
      return $this->_get_delegate()->is_a($this, $repository_id);
   }
   public function _non_existent()
   {
      return $this->_get_delegate()->non_existent($this);
   }
   /**
   *@deprecated Deprecated by CORBA 2.4
   */
   public function _get_interface()
   {
      return $this->_get_delegate()->get_interface($this);
   }
   public function _get_interface_def()
   {
      return $this->_get_delegate()->get_interface_def($this);
   }
   // methods for which the skeleton or application
   // programmer must provide an an implementation
   public abstract function _all_interfaces(
      org__omg__PortableServer__POA $poa, $objectId);
   // private implementation methods
   private $_delegate = null;
   public final function _get_delegate()
   {
      if ($this->_delegate === null) {
         throw new org__omg__CORBA__BAD_INV_ORDER(
      "The Servant has not been associated with an ORBinstance");
      }
      return $this->_delegate;
   }
   public final _set_delegate(
      org__omg__PortableServer__portable__Delegate $delegate)
   {
      $this->_delegate = $delegate;
   }
}
```

The **Servant** class is a PHP abstract class that serves as the base class for all POA servant implementations. It provides a number of methods that may be invoked by the application programmer, as well as methods that are invoked by the POA itself and may be overridden by the user to control aspects of servant behavior.

With the exception of the **_all_interfaces()** and **_this_object(ORB orb)** methods, all methods defined on the Servant class may only be invoked after the Servant has been associated with an ORB instance. Attempting to invoke the methods on a Servant that has not been associated with an ORB instance results in a CORBA::BAD_INV_ORDER exception being raised.

A Servant may be associated with an ORB instance via one of the following means:

- Through a call to **_this_object(ORB orb)** passing an ORB instance as parameter. The Servant will become associated with the specified ORB instance.
- By explicitly activating a Servant with a POA by calling either **POA::activate_object** or **POA::activate_object_with_id**. Activating a Servant in this fashion will associate the Servant with the ORB instance, which contains the POA on which the Servant has been activated.
- By returning a Servant instance from a ServantManager. The Servant

returned from **PortableServer__ServantActivator::incarnate()** or **PortableServer__ServantLocator::preinvoke()** will be associated with the ORB instance that contains the POA on which the ServantManager is installed.

- By installing the Servant as a default servant on a POA. The Servant will become associated with the ORB instance which contains the POA for which the Servant is acting as a default servant.
- By explicitly setting it by a call to **org__omg__CORBA__ORB::set_delegate()**. It is not possible to associate a Servant with more than one ORB instance at a time. Attempting to associate a Servant with more than one ORB instance will result in undefined behavior.

_this_object

The **_this_object()** methods have the following purposes:

- Within the context of a request invocation on the target object represented by the servant, it allows the servant to obtain the object reference for the target CORBA Object it is incarnating for that request. This is true even if the servant incarnates multiple CORBA objects. In this context, **_this_object()** can be called regardless of the policies the dispatching POA was created with.
- Outside the context of a request invocation on the target object represented by the servant, it allows a servant to be implicitly activated if its POA allows implicit activation. This requires the POA to have been created with the **IMPLICIT_ACTIVATION** policy. If the POA was not created with the **IMPLICIT_ACTIVATION** policy, the CORBA::OBJ_ADAPTER exception is thrown. The POA to be used for implicit activation is determined by invoking the servant's **_default_POA()** method.
- Outside the context of a request invocation on the target object represented by the servant, it will return the object reference for a servant that has already been activated, as long as the servant is not incarnating multiple CORBA objects. This requires the servant's POA to have been created with the **UNIQUE_ID** and **RETAIN** policies. If the POA was created with the **MULITPLE_ID** or **NON_RETAIN** policies, the CORBA::OBJ_ADAPTER exception is thrown. The POA used in this operation is determined by invoking the servant's **_default_POA()** method.
- The **_this_object(ORB orb)** method first associates the Servant with the specified ORB instance and then invokes **_this_object()** as normal.

_orb

The **_orb()** method is a convenience method that returns the instance of the ORB currently associated with the Servant.

_poa and ***_object_id***

The methods **_poa()** and **_object_id()** are equivalent to calling the methods **PortableServer__Current:get_POA** and **PortableServer__Current__get_object_id**. If the **PortableServer__Current** object throws a PortableServer::Current::NoContext exception, then **_poa()** and **_object_id()** throws a CORBA::OBJ_ADAPTER system exception instead.

These methods are provided as a convenience to the user to allow easy execution of these common methods.

_default_POA

The method **_default_POA()** returns a default POA to be used for the servant outside the context of POA invocations. The default behavior of this function is to return the root POA from the ORB instance associated with the servant. Subclasses may override this method to return a different POA. It is illegal to return a null value.

_all_interfaces

The **_all_interfaces()** method is used by the ORB to obtain complete type information from the servant. The ORB uses this information to generate IORs and respond to **_is_a()** requests from clients. The method takes a POA instance and an **ObjectId** as an argument and returns a sequence of repository ids representing the type of information

for that **oid**. The repository id at the zero index represents the most derived interface. The last id, for the generic CORBA Object (i.e.,"IDL:omg.org/CORBA/Object:1.0"), is implied and not present. An implementor of this method must return complete type information for the specified **oid** for the ORB to behave correctly.

_non_existent
Servant provides a default implementation of **_non_existent()** that can be overridden by derived servants if the default behavior is not adequate.

_get_interface
The **_get_interface_def()** operation returns an **InterfaceDef** object as a **CORBA::Object** that defines the runtime type of the **CORBA::Object** that is implemented by the **Servant**. The invoker of **_get_interface_def** must narrow the result to an **InterfaceDef** in order to use it.
Servant provides a default implementation of **_get_interface_def()** that can be overridden by derived servants if the default behavior is not adequate. As defined in the CORBA 2.3.1 specification, section 11.3.1, the default behavior of **_get_interface_def()** is to use the most derived interface of a static servant or the most derived interface retrieved from a dynamic servant to obtain the **InterfaceDef**. This behavior shall be supported by the **Delegate** that implements the **Servant**.

_is_a
Servant provides a default implementation of **_is_a()** that can be overridden by derived servants if the default behavior is not adequate. The default implementation checks to see if the specified **repid** is present on the list returned by **_all_interfaces()** (see "_all_interfaces" on page 1-99) or is the repository id for the generic CORBA Object. If so, then **_is_a()** returns **true**; otherwise, it returns **false**.

1.20.2.3 Mapping of Dynamic Skeleton Interface

This section contains the following information:
- Mapping of the Dynamic Skeleton Interface's **ServerRequest** to PHP.
- Mapping of the Portable Object Adapter's Dynamic Implementation Routine to PHP.

Mapping of ServerRequest
The **ServerRequest** interface maps to the following PHP class:

```
// PHP
abstract class ServerRequest {
   /**
   * @deprecated use operation()
   */
   public function op_name()
   {
      return operation();
   }
   public function operation() {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public abstract function ctx();
   /**
   * @deprecated use arguments()
   */
   public function params(org__omg__CORBA__NVList $parms)
   {
      $this->arguments($parms);
   }
   public function arguments(org__omg__CORBA__NVList $nv)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   /**
```

```
   * @deprecated use set_result()
   */
   public function result(org__omg__CORBA__Any $a)
   {
      $this->set_result($a);
   }
   public function set_result(org__omg__CORBA__Any $val)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   /**
   * @deprecated use set_exception()
   */
   public function except(org__omg__CORBA__Any $a)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function set_exception(org__omg__CORBA__Any $val)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
}
```

Note that several methods have been deprecated in CORBA 2.3 in favor of the current methods as defined in CORBA 2.2 and used in the current C++ mapping.
Implementations using the POA should use the new routines.

Mapping of POA Dynamic Implementation Routine
In PHP, POA-based DSI servants inherit from the standard **DynamicImplementation** class. This class inherits from the **org__omg__PortableServer__Servant** class. The **DynamicImplementation** class is defined as follows:

```
// PHP
abstract class org__omg__PortableServer__DynamicImplementation
   extends org__omg__PortableServer__Servant
{
   public abstract function invoke(
      org__omg__CORBA__ServerRequest $request);
}
```

The **invoke()** method receives requests issued to any CORBA object incarnated by the DSI servant and performs the processing necessary to execute the request.
The ORB user must also provide an implementation to the **_all_interfaces()** method declared by the **Servant** class.

1.20.2.4 Skeleton Portability

The PHP language mapping defines a binary portability layer (see Section 1.21, “PHP ORB Portability Interfaces,” on page 1-109) in order to provide binary compatibility of servant implementations on ORBs from different vendors. The server-side mapping supports this by defining two models of code generations for skeletons: a stream-based model and a DSI-based model. The example in the rest of the server-side mapping section uses the DSI-based model.

1.20.2.5 Skeleton Operations

All skeleton classes provide a **_this()** method. The method provides equivalent behavior to calling **_this_object()** but returns the most derived PHP interface type associated with the servant.
It should be noted that because of the way the inheritance hierarchy is set up, the Object returned by **_this()** is an instance of a stub as defined in Section 1.21.6, “Portability Stub and Skeleton Interfaces,” on page 1-119.

1.20.2.6 Inheritance-Based Interface Implementation

Implementation classes can be derived from a generated base class based on the OMG IDL interface definition. The generated base classes are known as *skeleton classes* and the derived classes are known as *implementation classes*. Each skeleton class implements the generated interface operations associated with the IDL interface definition. The implementation class shall provide implementations for each method defined in the interface operations class. It is important to note that the skeleton class does not extend the interface class associated with the IDL interface, but only implements the interface operations class.

For each IDL interface **<interface_name>**the mapping defines a PHP class as follows:

```
// PHP
abstract class <interface_name>POA
   extends org__omg__PortableServer__DynamicImplementation
   implements <interface_name>Operations
{
   public function _this( org__omg__CORBA__ORB $orb = null )
   {
      if ($orb === null) {
         return <interface_name>Helper::narrow(
            $this->_this_object());
      } else {
         return <interface_name>Helper::narrow(
            $this->_this_object($orb));
      }
   }
   public function _all_interfaces(
      org__omg__PortableServer__POA $poa, $objectId)
   {
      ...
   }
   public function invoke(org__omg__CORBA__ServerRequest
$request)
   {
      ...
   }
}
```

The implementation of **_all_interfaces()** and **invoke()** are provided by the compiler. The **_all_interfaces()** method must return the full type hierarchy as known at compile time.

For example, given the following IDL:

```
// IDL
interface A {
   short op1();
   void op2(in long val);
}
```

A skeleton class for interface **A** would be generated as follows:

```
// PHP
abstract class APOA
   extends org__omg__PortableServer__DynamicImplementation
   implements Aoperations
{
   public function _this( org__omg__CORBA__ORB $orb ) {
      if ($orb === null) {
         return Ahelper::narrow($this->_this_object());
      } else {
         return Ahelper::narrow($this->_this_object($orb));
      }
```

```
   }
   public function _all_interfaces(
      org__omg__PortableServer__POA $poa, $objectId)
   {
      ...
   }
   public function invoke(org__omg__CORBA__ServerRequest
$request)
   {
      ...
   }
}
```

The user subclasses the **APOA** class to provide implementations for the methods on **AOperations**.

1.20.2.7 Delegation-Based Interface Implementation

Because PHP does not allow multiple implementation inheritance, inheritance-based implementation is not always the best solution. Delegation can be used to help solve this problem. This section describes a delegation approach to implementation which is type safe.

For each IDL interface **<interface_name>** the mapping defines a PHP tie class as follows:

```
// PHP
class <interface_name>POATie
   extends <interface_name>POA
{
   private $_delegate;
   private $_poa;
   public function __construct(
      <interface_name>Operations $delegate,
      org__omg__PortableServer__POA $poa = null)
   {
      $this->_delegate = $delegate;
      $this->_poa = $poa;
   }
   public function _get_delegate()
   {
      return _delegate;
   }
   public function _set_delegate(
      <interface_name>Operations $delegate)
   {
      $this->_delegate = $delegate;
   }
   public function _default_POA()
   {
      if ($this->_poa !== null) {
         return _poa;
      } else {
         return parent::_default_POA();
      }
   }
   // for each method <method> defined in
   // <interface_name>Operations
   // The return statement is present for methods with
   // return values.
   public <method> {
      [return] $this->_delegate-><method>;
   }
}
```

Using the example above, a tie class for interface **A** would be generated as follows:

```
// PHP
public class APOATie extends APOA
{
   private $_delegate;
   private $_poa;

   public function __construct(AOperations $delegate,
      org__omg__PortableServer__POA $poa)
   {
      $this->_delegate = $delegate;
      $this->_poa = $poa;
   }
   public function _get_delegate()
   {
      return $this->_delegate;
   }
   public void _set_delegate(AOperations $delegate)
   {
      $this->_delegate = $delegate;
   }
   public function _default_POA()
   {
      if ($this->_poa !== null) {
         return $this->_poa;
      } else {
         return parent::_default_POA();
      }
   }
   public function op1()
   {
      return $this->_delegate->op1();
   }
   public function op2($val)
   {
      $this->_delegate->op2($val);
   }
   ...
}
```

To implement an interface using the delegation approach, a developer must write an implementation class, which implements the operations class associated with the interface they wish to implement. The developer then instantiates an instance of the implementation class and uses this instance in the constructor of the tie class associated with the interface. The tie class can then be used as servant in POA operations.
It is important to note that the implementation class has no access to the object reference associated with the tie object. One way for the delegate to access this information is for the delegate to keep a reference to the tie object. For a delegate that is tied to multiple tie objects, this approach will not work. Instead, the delegate can determine its current object reference by calling **PortableServer__Current::get_object_id()** and passing the return value to **PortableServer__POA::id_to_reference()**. The result may then be narrowed to the appropriate interface if required.

Ties for Local Interfaces
Ties for local interfaces are defined in a manner very similar to the normal POA-based Ties. The major difference is that no POA is present in the Tie, since local interfaces do not use the POA. Ties are useful for local interfaces for the same reason as they are for normal interfaces: PHP does not support multiple inheritance.
For each local interface <interface_name> the mapping defines a PHP local tie class as follows:

```
// PHP
public class <interface_name>LocalTie
   extends <interface_name>LocalBase
```

```
{
   private $_impl;
   public function __construct (
      <interface_name>Operations $delegate)
   {
      $this->_impl = $delegate;
   }
   public function _get_delegate()
   {
      return $this->_impl;
   }
   public function _set_delegate (
      <interface_name>Operations $delegate)
   {
      $this->_impl = $delegate;
   }
   // For each <method> defined in
   // <interface_name>Operations (the return statement
   // is present for methods with non-void
   // <method_return_type>):
   public function <method>( <params> )
   {
      [return] $this->_impl-><method>( <param_names> );
   }
}
```

Assume that we have the interface Alocal defined as follows:

```
local interface A {
   short op1() ;
   void op2( in long val ) ;
} ;
```

Then the LocalTie class for ALocal is as follows:

```
class ALocalTie extends AlocalBase
{
   public function __construct( AOperations $delegate )
   {
      $this->_impl = $delegate;
   }
   public function _get_delegate()
   {
      return $this->_impl;
   }
   public function _set_delegate (AOperations $delegate )
   {
      $this->_impl = $delegate;
   }
   public function op1 ()
   {
      return $this->_impl->op1();
   }
   public function op2 ($val)
   {
      $this->_impl->op2($val);
   }
   private $_impl;
}
```

To implement a local interface using the Tie approach, a developer must write an implementation class, which implements the operations class associated with the interface they wish to implement. The developer then instantiates an instance of the implementation class and uses this instance in the constructor of the LocalTie class associated with the interface. The Tie class then provides an implementation of the local interface.

1.20.3 Mapping for PortableServer::ServantManager

1.20.3.1 Mapping for Cookie

The native type **PortableServer::ServantLocator::Cookie** is mapped to **php__lang__Object**.
For the PHP mapping of the **PortableServer::ServantLocator::preinvoke()** operation, a **Cookie** object set to null will be passed by reference. The user may then set the value to any PHP object. The same **Cookie** object will then be passed to the **PortableServer::ServantLocator::postinvoke()** operation.

1.20.3.2 ServantManagers and AdapterActivators

Portable servants that implement the **PortableServer::AdapterActivator**, the **PortableServer::ServantActivator**, or **PortableServer::ServantLocator** interfaces are implemented just like any other servant. They may use either the inheritance-based approach or the delegation-based approach.

1.21 PHP ORB Portability Interfaces

1.21.1 Introduction

The APIs specified here provide the minimal set of functionality to allow portable stubs and skeletons to be used with a PHP ORB. The interoperability requirements for PHP go beyond that of other languages. Because PHP classes are often downloaded and come from sources that are independent of the ORB in which they will be used, it is essential to define the interfaces that the stubs and skeletons use. Otherwise, use of a stub (or skeleton) will require: either that it have been generated by a tool that was provided by the ORB vendor (or is compatible with the ORB being used), or that the entire ORB runtime be downloaded with the stub or skeleton. Both of these scenarios are unacceptable.
Two such styles of interfaces are defined, one based on the DII/DSI, the other based on a streaming approach. Conforming ORB PHP runtimes shall support both styles. A conforming vendor tool may choose between the two styles of stubs/skeletons to generate, but shall support the generation of at least one style.

1.21.1.1 Design Goals

The design balances several goals:

- **Size** - Stubs and skeletons must have a small bytecode footprint in order to make downloading fast in a browser environment and to minimize memory requirements when bundled with a PHP runtime, particularly in specialized environments such as settop boxes.
- **Performance** - Obviously, the runtime performance of the generated stub code must be excellent. In particular, care must be taken to minimize temporary PHP object creation during invocations in order to avoid PHP runtime garbage collection overhead.

A very simple delegation scheme is specified here. Basically, it allows ORB vendors maximum flexibility for their ORB interfaces, as long as they implement the interface APIs. Of course vendors are free to add proprietary extensions to their ORB runtimes.

Stubs and skeletons that require proprietary extensions will not necessarily be portable or interoperable and may require download of the corresponding runtime.

1.21.2 Overall Architecture

The stub and skeleton portability architecture supports the use of both the DII/DSI, and a streaming API as its portability layer. The mapping of the DII and DSI PIDL have operations that support the efficient implementation of portable stubs and skeletons.
The major components to the architecture are:

- Portable Streamable - provides standard APIs to read and write IDL datatypes.
- Portable Streams - provide standard APIs to the ORB's marshaling engine.
- Portable Stubs and Skeletons - provides standard APIs that are used to connect stubs and skeletons with the ORB.
- Portable Delegate - provides the vendor specific implementation of CORBA object.
- Portable Servant Delegate - provides the vendor specific implementation of **PortableServer::Servant**.
- ORB Initialization - provides standard way to initialize the ORB.

1.21.2.1 Portability Package

The APIs needed to implement portability are found in the **org__omg__CORBA__portable** and **org__omg__PortableServer__portable** prefix.
The portability package contains interfaces and classes that are designed for and intended to be used by ORB implementor. It exposes the publicly defined APIs that are used to connect stubs and skeletons to the ORB.

1.21.3 Streamable APIs

The Streamable Interface API provides the support for the reading and writing of complex data types. It is implemented by static methods on the Helper classes.

```
interface org__omg__CORBA__portable__Streamable {
   function _read(org__omg__CORBA__portable__InputStream $is);
   function _write(org__omg__CORBA__portable__OutputStream $os);
   function _type();
}
```

1.21.4 Streaming APIs

The streaming APIs are PHP interfaces that provide for the reading and writing of all of the mapped IDL types to and from streams. Their implementations are used inside the ORB to marshal parameters and to insert and extract complex datatypes into and from **Any**s.
The streaming APIs are found in the **org__omg__CORBA__portable** and **org__omg__CORBA_2_3__portable** prefix. The ORB object is used as a factory to create an output stream. An input stream may be created from an output stream.

```
interface org__omg__CORBA__ORB
{
   function create_output_stream();
}

abstract class org__omg__CORBA__portable__InputStream
   extends php__io__InputStream
{
   public function read() {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function orb() {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public abstract function read_boolean();
   public abstract function read_char();
```

```
    public abstract function read_wchar();
    public abstract function read_octet();
    public abstract function read_short();
    public abstract function read_ushort();
    public abstract function read_long();
    public abstract function read_ulong();
    public abstract function read_longlong();
    public abstract function read_ulonglong();
    public abstract function read_float();
    public abstract function read_double();
    public abstract function read_string();
    public abstract function read_wstring();
    public abstract function read_boolean_array(
       $value, $offset, $length);
    public abstract function read_char_array(
       $value, $offset, $length);
    public abstract function read_wchar_array(
       $value, $offset, $length);
    public abstract function read_octet_array(
       $value, $offset, $length);
    public abstract function read_short_array(
       $value, $offset, $length);
    public abstract function read_ushort_array(
       $value, $offset, $length);
    public abstract function read_long_array(
       $value, $offset, $length);
    public abstract function read_ulong_array(
       $value, $offset, $length);
    public abstract function read_longlong_array(
       $value, $offset, $length);
    public abstract function read_ulonglong_array(
       $value, $offset, $length);
    public abstract function read_float_array(
       $value, $offset, $length);
    public abstract function read_double_array(
       $value, $offset, $length);
    public abstract function read_Object($clz = '');
    public abstract function read_TypeCode();
    public abstract function read_any();
    public function read_Context() {
       throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function read_fixed($digits, $scale) {
       throw new org__omg__CORBA__NO_IMPLEMENT();
    }
}

abstract class org__omg__CORBA__portable__OutputStream
    extends php__io__OutputStream
{
    public function write($b) {
       throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function orb() {
       throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public abstract function create_input_stream();
    public abstract function write_boolean ($value);
    public abstract function write_char ($value);
    public abstract function write_wchar ($value);
    public abstract function write_octet ($value);
    public abstract function write_short ($value);
    public abstract function write_ushort ($value);
    public abstract function write_long ($value);
    public abstract function write_ulong ($value);
    public abstract function write_long long ($value);
    public abstract function write_ulonglong ($value);
    public abstract function write_float ($value);
```

```
   public abstract function write_double ($value);
   public abstract function write_string ($value);
   public abstract function write_wstring ($value);
   public abstract function write_boolean_array(
      $value, $offset, $length);
   public abstract function write_char_array (
      $value, $offset, $length);
   public abstract function write_wchar_array (
      $value, $offset, $length);
   public abstract function write_octet_array (
      $value, $offset, $length);
   public abstract function write_short_array (
      $value, $offset, $length);
   public abstract function write_ushort_array (
      $value, $offset, $length);
   public abstract function write_long_array (
      $value, $offset, $length);
   public abstract function write_ulong_array (
      $value, $offset, $length);
   public abstract function write_longlong_array (
      $value, $offset, $length);
   public abstract function write_ulonglong_array (
      $value, $offset, $length);
   public abstract function write_float_array (
      $value, $offset, $length);
   public abstract function write_double_array (
      $value, $offset, $length);
   public abstract function write_Object(
      org__omg__CORBA__Object $value);
   public abstract function write_TypeCode(
      org__omg__CORBA__TypeCode $value);
   public abstract function write_any (
      org__omg__CORBA__Any $value);
   public function write_Context(
      org__omg__CORBA__Context $ctx,
      org__omg__CORBA__ContextLists $contexts)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function write_fixed(
      php__math__BigDecimal $value, $digits, $scale)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
}

abstract class org__omg__CORBA_2_3__portable__InputStream
   extends org__omg__CORBA__portable__InputStream
{
   public function read_value($repid_clz_helper=0)
   {
      if ($repid_clz_helper === 0) {
         return read_value_pure();
      } elseif (is_object($repid_clz_helper)) {
         if ($repid_clz_helper instanceof php__io__serializable) {
            return read_value_serializable($repid_clz_helper);
         } else {
            return read_value_helper($repid_clz_helper);
         }
      } elseif (is_string($repid_clz_helper) {
         if (substr($repid_clz_helper, 0, 4) == 'IDL:')) {
            read_value_repid($repid_clz_helper);
         } else {
            read_value_clz($repid_clz_helper);
         }
      } else {
         throw new org__omg__CORBA__BAD_PARAM();
      }
```

```
    }
    public function read_value_pure()
    {
       throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function read_value_repid($rep_id)
    {
       throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function read_value_clz($clz)
    {
       throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function read_value_helper(
       org__omg__CORBA__portable__BoxedValueHelper $factory)
    {
       throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function read_value_serializable(
       php__io__Serializable $value)
    {
       throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function read_abstract_interface()
    {
       throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    public function read_abstract_interface($clz)
    {
       throw new org__omg__CORBA__NO_IMPLEMENT();
    }
}

abstract class org__omg__CORBA_2_3__portable__OutputStream
    extends org__omg__CORBA__portable__OutputStream
{
    public function write_value(
       php__io__Serializable $value,
       $repid_clz_helper=0)
    {
       if ($repid_clz_helper === 0) {
          write_value_pure($value);
       } elseif (is_object($repid_clz_helper)) {
          write_value_helper($value, $repid_clz_helper);
       } elseif (is_string($repid_clz_helper) {
          if (substr($repid_clz_helper, 0, 4) == 'IDL:')) {
             write_value_repid($value, $repid_clz_helper);
          } else {
             write_value_clz($value, $repid_clz_helper);
          }
       } else {
          throw new org__omg__CORBA__BAD_PARAM();
       }
    }
    private function write_value_pure($value)
    {
       throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    private function write_value_repid(
       php__io__Serializable $value, $rep_id)
    {
       throw new org__omg__CORBA__NO_IMPLEMENT();
    }
    private function write_value_clz(
       php__io__Serializable $value, $clz)
    {
       throw new org__omg__CORBA__NO_IMPLEMENT();
    }
```

```
   private function write_value_helper(
      php__io__Serializable $value,
      org__omg__CORBA__portable__BoxedValueHelper factory)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public void write_abstract_interface($obj) {
      if (! is_object($obj)) {
         throw new org__omg__CORBA__BAD_PARAM();
      }
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
}
```

1.21.4.1 InputStream Method Semantics

read_Context

The **read_Context()** method reads a **Context** from the stream. The **Context** is read from the stream as a sequence of strings as specified in *The Common Object Request Broker Architecture (CORBA)*, GIOP chapter.

read_Object

For read_Object, the clz argument is one of the following:

- the **Class** object for the stub class which corresponds to the type that is statically expected. Typically, the ORB runtime will allocate and return a stub object of this stub class.
- the **Class** object for the IDL interface type that is statically expected. The ORB runtime must allocate and return a stub object that conforms to this interface.

read_abstract_interface

For **read_abstract_interface**, the ORB runtime will return either a value object or a suitable stub object. The specified clz argument is one of the following:

- the **Class** object for the stub class which corresponds to the type that is statically expected.
- the **Class** object for the IDL interface type that is statically expected. If a stub object is returned, it must conform to this interface.

The **read_abstract_interface()** and **read_abstract_interface(clz)** actual implementations may throw the **org__omg__CORBA__portable__IndirectionException** exception.

read_value

The **read_value()** methods unmarshal a value type from the input stream. The specified **clz** is the declared type of the value to be unmarshaled. The specified **rep_id** identifies the type of the value type to be unmarshaled. The specified **factory** is the instance of the helper to be used for unmarshaling the boxed value.

The specified **value** is an uninitialized value that is added to the orb's indirection table before calling **Streamable::_read()** or **CustomMarshal::unmarshal()** to unmarshal the value.

The **read_value()** and **read_value(clz)** actual implementations may throw the **org__omg__CORBA__portable__IndirectionException** exception.

read_fixed

The short parameters indicate the digits and scale of the fixed-point decimal. These parameters are needed because when the **fixed** is read off the stream, the **BigDecimal** can't be reconstructed to the original digits and scale without explicitly passing in the digits and scale.

1.21.4.2 OutputStream Method Semantics

create_input_stream
The **create_input_stream()** method returns a new input stream from the output stream. The method implements copy semantics, so that the current contents of the output stream is copied to the input stream. Anything subsequently written to the output stream is not visible to the newly created input stream.

write_Context
The **write_context()** method writes the specified **Context** to the stream. The **Context** is marshaled as a sequence of strings as specified in *The Common Object Request Broker Architecture (CORBA)*, GIOP chapter. Only those Context values specified in the **contexts** parameter are actually written.

write_value
The **write_value()** methods marshal a value type to the output stream. The first parameter is the actual value to write. The specified **clz** is the declared type of the value to be marshaled. The specified **rep_id** identifies the type of the value type to be marshaled. The specific **factory** is the instance of the helper to be used for marshaling the boxed value.

write_Object
The implementation of **write_Object()** shall check if the object passed in is of type **org__omg__CORBA__LocalObject** and if so an **org__omg__CORBA__MARSHAL** exception shall be thrown.

write_fixed
The short parameters indicate the digits and scale of the fixed-point decimal. These parameters are needed because the information of the maximum number of digits and scale is lost when the IDL fixed is mapped to a **BigDecimal** for **write_fixed**.

1.21.5 Custom Streaming APIs

The Custom streaming APIs provide for the reading and writing of all the different valuetypes that may be custom marshaled. The semantics of the methods in these interfaces are identical to their counterparts (with identical signatures) in the CORBA_2_3 portable streaming APIs.

```
interface CustomOutputStream extends
   org__omg__CORBA__DataOutputStream
{
   public function write_value(
      php__io__Serializable $value, $repid_clz_helper=0);
   public function write_value_pure(php__io__Serializable
$value);
   public function write_value_repid(php__io__Serializable
$value,
      $repId);
   public function write_value_helper(
      php__io__Serializable $value,
      org__omg__CORBA__portable__BoxedValueHelper $helper);
   public function write_abstract_interface($obj);
}
interface CustomInputStream extends
   org__omg__CORBA__DataInputStream
{
   public function read_value($repid_clz_helper=0);
   public function read_value_pure();
   public function read_value_repid($repId);
   public function read_value_clz($clz);
   public function read_abstract_interface();
   public function read_abstract_interface($clz);
```

```
    public function read_value_helper(
       org__omg__CORBA__portable__BoxedValueHelper $helper);
    // boxed valuetypes
}
```

1.21.6 Portability Stub and Skeleton Interfaces

1.21.6.1 Stub/Skeleton Architecture

The mapping defines a single stub that may be used for both local and remote invocation. Local invocation provides higher performance for collocated calls on Servants located in the same process as the client. Remote invocation is used to invoke operations on objects that are located in an address space separate from the client.

Note that neither stubs nor skeletons are required for interfaces that are defined as local.

While a stub is using local invocation it provides complete location transparency. To provide the correct semantics, compliant programs comply with the parameter passing semantics defined in Section 1.12.2, "Parameter Passing Modes," on page 1-34. When using local invocation the stub copies all valuetypes passed to them, either as in parameters, or as data within in parameters, and passes the resulting copies to the Servant in place of the originals. The valuetypes are copied using the same deep copy semantics as would result from GIOP marshaling and unmarshaling.
The following sections describe the characteristics of the stubs and skeletons. The examples are based on the following IDL:

```
// Example IDL
module Example {
    exception AnException {};
    interface AnInterface {
        long length(in string s) raises (AnException);
    };
};
```

Stub Design
All stubs inherit from a common base class **org__omg__CORBA__portable__ObjectImpl**. The class is responsible for delegating shared functionality such as **is_a()** to the vendor specific implementation.
This model provides for a variety of vendor dependent implementation choices, while reducing the client-side and server "code bloat".
The stub is named ***_<interface_name>*Stub** where ***<interface_name>*** is the IDL interface name this stub is implementing and implements the signature interface ***<interface_name>***.

The stub class supports either the DII or the streaming style APIs.

Stubs are not required to define any constructors. However, if they do define special constructors, a constructor with no arguments must also be defined.

Skeleton Design
Skeletons may be either stream-based or DSI-based.

Stream-based skeletons directly extend the **org__omg__PortableServer__Servant** class (Section 1.20.2.2, "Mapping of PortableServer::Servant," on page 1-96) and implement the **InvokeHandler** interface (Section 1.21.6.4, "Invoke Handler," on page 1-133) as well as the operations interface associated with the IDL interface the skeleton implements.

DSI-based skeletons directly extend the **org__omg__PortableServer__DynamicImplementation** class (Section 1.20.2.3,

"Mapping of Dynamic Skeleton Interface," on page 1-100) and implement the operations interface associated with the IDL interface the skeleton implements.

Stream-based Stub Example

```
class Example___AnInterfaceStub
   extends org__omg__CORBA__portable__ObjectImpl
   implements Example__AnInterface
{
   public static function _ids ()
   {
      return self::$__ids;
   }
   private static $__ids = array('IDL:Example/AnInterface:1.0');
   const _opsClass = 'Example__AnInterfaceOperations';
   public function length($s)
   {
      while(true) {
         if(!$this->_is_local()) {
            $_output = null;
            $_input = null;
            try {
               $_output = $this->_request('length', true);
               $_output->write_string($s);
               $_input = $this->_invoke($_output);
               $aux = $_input->read_long();
               $this->_releaseReply($_input); //finally
               return $aux;
            } catch (
               org__omg__CORBA__portable__RemarshalException
               $_exception){
               $this->_releaseReply($_input); //finally
               continue;
            } catch (
               org__omg__CORBA__portable__ApplicationException
               $_exception){
               $_exception_id = $_exception->getId();
               if ($_exception_id ==
                  Example__AnExceptionHelper::id())) {
                  $_input = $_exception->getInputStream();
                  $aux = Example__AnExceptionHelper::read($_input);
                  $this->_releaseReply($_input); //finally
                  throw $aux;
               }
               $this->_releaseReply($_input); //finally
               throw new org__omg__CORBA__UNKNOWN(
                  "Unexpected User Exception: $_exception_id");
            }
            $this->_releaseReply($_input); //finally
         } else {
            // co-located call optimization
            $_so = $this->_servant_preinvoke(
               'length', self::_opsClass );
            if ($_so === null) {
               continue;
            }
            $_self = $_so->servant;
            try {
               $result = $_self->length($s);
               if ($_so instanceof
                     org__omg__CORBA__portable__ServantObjectExt) {
                  $_so->normalCompletion();
               }
               $this->_servant_postinvoke( $_so ); //finally
               return $result;
            } catch (Example__AnException $exc) {
               if ($_so instanceof
                     org__omg__CORBA__portable__ServantObjectExt) {
```

```
                $_so->exceptionalCompletion( $exc ) ;
              }
              $this->_servant_postinvoke( $_so ); //finally
              throw $exc ;
            } catch (php__lang__RuntimeException $re) {
              if ($_so instanceof
                   org__omg__CORBA__portable__ServantObjectExt) {
                $_so->exceptionalCompletion( $re );
              }
              $this->_servant_postinvoke( $_so ); //finally
              throw $re;
            } catch (php__lang__Error $err) {
              if ($_so instanceof
                   org__omg__CORBA__portable__ServantObjectExt) {
                $_so->exceptionalCompletion( $err ) ;
              }
              $this->_servant_postinvoke( $_so ); //finally
              throw $err;
            }
            $this->_servant_postinvoke( $_so ); //finally
          }
        }
      }
    }
```

Stream-based Skeleton Example

```
abstract class Example__AnInterfacePOA
  extends org__omg__PortableServer__Servant
  implements org__omg__CORBA__portable__InvokeHandler,
    Example__AnInterfaceOperations
{
  public function _this( org__omg__CORBA__ORB $orb = null)
  {
    if ($orb === null) {
      return Example__AnInterfaceHelper::narrow(
        parent::_this_object());
    } else {
      return Example__AnInterfaceHelper::narrow(
        parent::_this_object($orb));
    }
  }
  public function _all_interfaces(
    org__omg__PortableServer__POA $poa, $objectId)
  {
    return self::$__ids;
  }
  private static $__ids = array('IDL:Example/AnInterface:1.0');
  public function _invoke($opName,
    org__omg__CORBA__portable__InputStream $is,
    org__omg__CORBA__portable__ResponseHandler $handler)
  {
    $_output = null;
    if ($opName == 'length') {
      try {
        $s = $is->read_string();
        $_result = $this->length($s);
        $_output = $handler->createReply();
        $_output->write_long($_result);
      } catch (Example__AnException $_exception) {
        $_output = $handler->createExceptionReply();
        Example__AnExceptionHelper::write(
          $_output, $_exception);
      }
      return $_output;
    } else {
      throw new org__omg__CORBA__BAD_OPERATION();
    }
  }
```

```
}
class Example__AnInterfacePOATie
   extends Example__AnInterfacePOA
{
   private $_delegate;
   private $_poa;
   public function __construct(
      Example__AnInterfaceOperations $delegate,
      org__omg__PortableServer__POA $poa = null)
   {
      $this->_delegate = $delegate;
      $this->_poa = $poa;
   }
   public function _get_delegate()
   {
      return this._delegate;
   }
   public function _set_delegate(
      Example__AnInterfaceOperations $delegate)
   {
      $this->_delegate = $delegate;
   }
   public function _default_POA() {
      if($this->_poa !== null) {
         return $this->_poa;
      } else {
         return parent::_default_POA();
      }
   }
   public function length ($s)
   {
      return $this->_delegate->length($s);
   }
}
```

Dynamic (DII-based) Stub Example

```
public class Example___AnInterfaceStub
   extends org__omg__CORBA__portable__ObjectImpl
   implements Example__AnInterface
{
   public static function _ids() {
      return self::$__ids;
   }
   private static $__ids = array('IDL:Example/AnInterface:1.0');
   public function length ($s) {
      $_request = $this->_request('length');
      $_request->set_return_type(
         $this->_orb()->get_primitive_tc(
            org__omg__CORBA__TCKind::tk_long));
      $_s = $_request->add_in_arg();
      $_s->insert_string($s);
      $_request->exceptions()->add(
         Example__AnExceptionHelper::type());
      $_request->invoke();
      $_exception = $_request->env()->exception();
      if($_exception !== null) {
         if($_exception instanceof
            org__omg__CORBA__UnknownUserException) {
            $_userException = $_exception;
            if($_userException->except->type()->equals(
               Example__AnExceptionHelper::type())) {
               throw Example__AnExceptionHelper::extract(
                  $_userException->except);
            } else {
               throw new org__omg__CORBA__UNKNOWN();
            }
         }
         throw $_exception;
```

```
      }
      $_result = $_request->return_value()->extract_long();
      return $_result;
   }
}
```

Dynamic (DSI-based) Skeleton Example

```
abstract class Example__AnInterfacePOA
   extends org__omg__PortableServer__DynamicImplementation
   implements Example__AnInterfaceOperations
{
   public function _this()
   {
      return Example__AnInterfaceHelper::narrow(
         parent::_this_object());
   }
   public function _this(org__omg__CORBA__ORB $orb)
   {
      return Example__AnInterfaceHelper::narrow(
         parent::_this_object($orb));
   }
   public static function _all_interfaces(
      org__omg__PortableServer__POA $poa,
      $objectId)
   {
      return self::$__ids;
   }
   private static $__ids = array('IDL:Example/AnInterface:1.0');
   public function invoke(
      org__omg__CORBA__ServerRequest $_request)
   {
      $_method = $_request->operation();
      if($_method == 'length') {
         try {
            $_params = $this->_orb()->create_list(1);
            $_s = $this->_orb()->create_any();
            $_s->type($this->_orb()->get_primitive_tc(
               org__omg__CORBA__TCKind::tk_string));
            $_params->add_value(
               's', $_s, org__omg__CORBA__ARG_IN::$value);
            $_request->arguments($_params);
            $s = $_s->extract_string();
            $_result = $this->length($s);
            $_resultAny = $this->_orb()->create_any();
            $_resultAny->insert_long($_result);
            $_request->set_result($_resultAny);
         } catch (Example__AnException $_exception) {
            $_exceptionAny = $this->_orb()->create_any();
            Example__AnExceptionHelper::insert(
               $_exceptionAny, $_exception);
            $_request->set_exception($_exceptionAny);
         }
         return;
      } else {
         throw new org__omg__CORBA__BAD_OPERATION();
      }
   }
}
```

1.21.6.2 Stub and Skeleton Class Hierarchy

The required class hierarchy is shown in Figure 1-2 on page 1-127. The hierarchy is shown for a sample IDL interface **Foo**. Classes that are PHP interfaces are indicated with the word *interface* before the class name. Classes with the prefix **org__omg** are defined by the PHP mapping. Classes with a slash in the upper left-hand corner indicate classes that are generated by the IDL compiler or other tools. Classes beginning with

User indicate user defined classes, which implement interfaces.
The following diagram shows the hierarchy used for DSI-based skeletons. For streambased skeletons, the **org__omg__PortableServer__DynamicImplementation** class is omitted from the hierarchy, and **FooPOA** extends **org__omg__PortableServer__Servant** and implements **org__omg__CORBA__portable__InvokeHandler**.

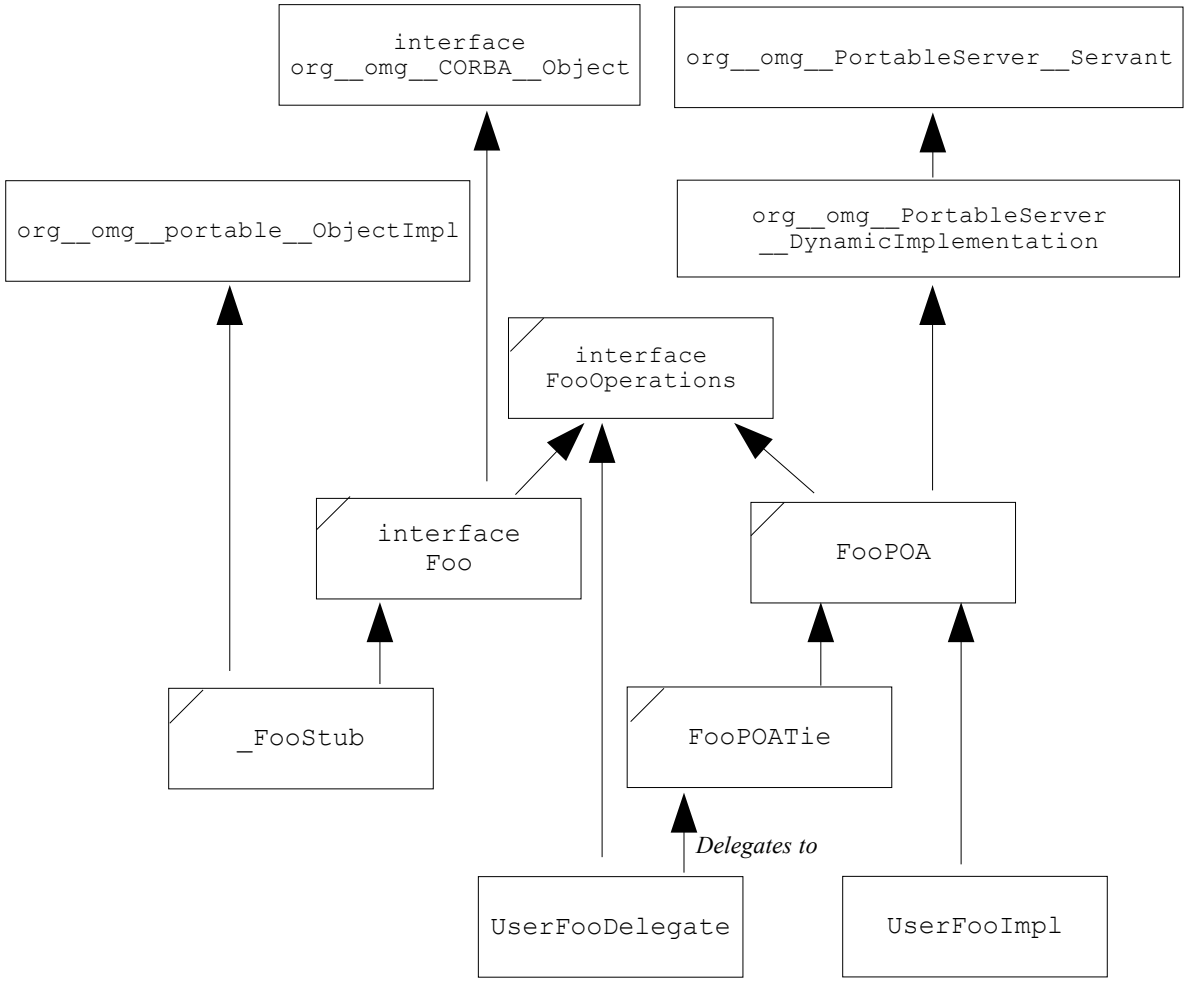

Figure 1-2 Class hierarchy for portable PHP stubs and skeletons

1.21.6.3 Portable ObjectImpl

The ObjectImpl class is the base class for stubs. It provides the basic delegation mechanism.

```
class org__omg__CORBA__portable__ServantObject {
   public $servant;
}
abstract class org__omg__CORBA__portable__ServantObjectExt
   extends org__omg__CORBA__portable__ServantObject
{
   abstract public function normalCompletion() ;
   abstract public function exceptionalCompletion(
      php__lang__Throwable $thr);
}

abstract class org__omg__CORBA__portable__ObjectImpl
   implements org__omg__CORBA__Object
{
   private $__delegate;
   public function _get_delegate()
```

```
{
   if ($this->__delegate === null) {
      throw new org__omg__CORBA__BAD_OPERATION();
   }
   return $this->__delegate;
}
public function _set_delegate(
   org__omg__CORBA__portable__Delegate $delegate)
{
   $this->__delegate = $delegate;
}
public abstract function _ids()
{...}

public function _get_interface_def()
{
   return $this->_get_delegate()->get_interface_def($this);
}
public function _duplicate()
{
   return $this->_get_delegate()->duplicate($this);
}
public function _release()
{
   $this->_get_delegate()->release($this);
}
public function _is_a($repository_id)
{
   return $this->_get_delegate()->is_a($this, $repository_id);
}
public function _is_equivalent(org__omg__CORBA__Object $rhs)
{
   return $this->_get_delegate()->is_equivalent($this, $rhs);
}
public function _non_existent()
{
   return $this->_get_delegate()->non_existent($this);
}
public function _hash($maximum)
{
   return $this->_get_delegate()->hash($this, $maximum);
}
public function _request($operation)
{
   return $this->_get_delegate()->request($this, $operation);
}
public function _request($operation, $responseExpected)
{
   return $this->_get_delegate()->request(
      $this, $operation, $responseExpected);
}
public function _invoke(
   org__omg__CORBA__portable__OutputStream $os)
{
   return $this->_get_delegate()->invoke($this, $os);
}
public function _releaseReply(
   org__omg__CORBA__portable__InputStream $is)
{
   return $this->_get_delegate()->releaseReply($this, $is);
}
public function _create_request(
   org__omg__CORBA__Context $ctx, $operation,
   org__omg__CORBA__NVList $arg_list,
   org__omg__CORBA__NamedValue $result)
{
   return $this->_get_delegate()->create_request(
      $this, $ctx, $operation, $arg_list, $result);
```

```
    }
    public function _create_request2(
       org__omg__CORBA__Context $ctx, $operation,
       org__omg__CORBA__NVList $arg_list,
       org__omg__CORBA__NamedValue $result,
       org__omg__CORBA__ExceptionList $exceptions,
       org__omg__CORBA__ContextList $contexts)
    {
       return $this->_get_delegate()->create_request2(
          $this, $ctx, $operation, $arg_list, $result,
          $exceptions, $contexts);
    }
    public function _get_policy($policy_type) {
       return $this->_get_delegate()->get_policy(
          $this, $policy_type);
    }
    public function _get_domain_managers() {
       return $this->_get_delegate()->get_domain_managers($this);
    }
    public function _set_policy_override(
       $policies, org__omg__CORBA__SetOverrideType $set_add)
    {
       return $this->_get_delegate()->set_policy_override(
          $this, $policies, $set_add);
    }
    public function _orb()
    {
       return $this->_get_delegate()->orb($this);
    }
    public function _is_local() {
       return $this->_get_delegate()->is_local($this);
    }
    public function _servant_preinvoke($operation, $expectedType)
    {
       return $this->_get_delegate()->servant_preinvoke(
          $this, $operation, $expectedType);
    }
    public function _servant_postinvoke(
       org__omg__CORBA__portable__ServantObject $servant)
    {
       $this->_get_delegate()->servant_postinvoke($this, $servant);
    }
    public function __toString()
    {
       if ( $this->__delegate !== null ) {
          return $this->__delegate->__toString($this);
       } else {
          return get_class($this) . ": no delegate set";
       }
    }
    public function hashCode()
    {
       if ( $this->__delegate !== null ) {
          return $this->__delegate->hashCode($this);
       } else {
          return php__lang__System::identityHashCode($this);
       }
    }
    public function equals($obj) {
       if ( $this->__delegate !== null ) {
          return $this->__delegate->equals($this, $obj);
       } else {
          return ($this===$obj);
       }
    }
}

abstract class org__omg__CORBA_2_3__portable__ObjectImpl
```

```
   extends org__omg__CORBA__portable__ObjectImpl
{
   /** Returns the codebase for this object reference.
   * @return the codebase as a space delimited list of url
   * strings or null if none
   */
   public function _get_codebase() {
      $delegate = $this->_get_delegate();
      if ($delegate instanceof
            org__omg__CORBA_2_3__portable__Delegate) {
         return $delegate->get_codebase($this);
      }
      return null;
   }
}
```

_ids

The method **_ids()** returns an array of repository ids that an object implements. The string at the zero index represents the most derived interface. The last id, for the generic CORBA object (i.e., "IDL:omg.org/CORBA/Object:1.0") is implied and not present.

Streaming Stub APIs

The method **_request()** is called by a stub to obtain an **OutputStream** for marshaling arguments. The stub must supply the operation name, and indicate if a response is expected (i.e., is this a one way call).

The method **_invoke()** is called to invoke an operation. The stub provides an OutputStream that was previously returned from a **_request()** call. The method **_invoke ()** returns an **InputStream** that contains the marshaled reply. The **_invoke()** method may throw only one of the following: an **ApplicationException**, a **RemarshalException**, or a CORBA system exception as described below:

- An **ApplicationException** is thrown to indicate the target has raised a CORBA user exception during the invocation. The stub may access the **InputStream** of the **ApplicationException** to unmarshal the exception data.
- A **RemarshalException** is thrown if the stub was redirected to a different target object and remarshaling is necessary, this is normally due to a GIOP object forward or locate forward message. In this case, the stub then attempts to reinvoke the request on behalf of the client after verifying the target is still remote by invoking **_is_local()** (see "Local Invocation APIs" on page 1-132). If **_is_local()** returns **True**, then an attempt to reinvoke the request using the Local Invocation APIs shall be made.
- If the CORBA system exception org__omg__CORBA__portable__UnknownException is thrown, then the stub does one of the following:
  - Translates it to **org__omg__CORBA__UNKNOWN**.
  - Translates it to the nested exception that the **UnknownException** contains.
  - Passes it on directly to the user.
- If the CORBA system exception being thrown is not **org__omg__CORBA__portable__UnknownException**, then the stub passes the exception directly to the user.

The method **_releaseReply()** may optionally be called by a stub to release a reply stream back to the ORB when unmarshaling has completed. The stub passes the InputStream returned by **_invoke()** or **ApplicationException.getInputStream()**. A null value may also be passed to **_releaseReply()**, in which case the method is a noop. This method may be used by the ORB to assist in buffer management.

Local Invocation APIs

Local invocation is supported by the following methods and classes.

The **_is_local()** method is provided so that stubs may determine if a particular object is implemented by a local servant and hence local invocation APIs may be used.

The **_is_local()** method returns true if the servant incarnating the object is located in the same process as the stub and they both share the same ORB instance. The **_is_local()** method returns **false** otherwise. The default behavior of **_is_local()** is to return **false**.

The **_servant_preinvoke()** method is invoked by a local stub to obtain a PHP reference to the servant that should be used for this request. The method takes a string containing the operation name and a class name representing the expected type of the servant as parameters and returns a ServantObject object.

Note – ORB vendors may subclass the ServantObject object to return an additional request state that may be required by their implementations.

The operation name corresponds to the operation name as it would be encoded in a GIOP request. The expected type is the Class object associated with the operations class of the stub's interface (e.g., a stub for an interface **Foo**, would pass the class name for the **FooOperations** interface). The method returns a **null** value if the servant is not local or the servant has ceased to be local as a result of the call (i.e., due to a ForwardRequest from a POA ServantManager). The method throws an **org__omg__CORBA__BAD_PARAM** exception if the servant is not of the expected type. If a **ServantObject** object is returned, then the servant field has been set to an object of the expected type.

Note – The object may or may not be the actual servant instance.

The local stub may invoke the operation directly. The **ServantRequest** object is valid for only one invocation, and cannot be used for more than one invocation.
If the ServantObject returned by the _servant_preinvoke() call is an instance of ServantObjectExt, the local invocation code must also satisfy the following conditions:

1. If the invocation on the servant completes without throwing an exception, then the stub code must call `$servant->normalCompletion()` after the invocation completes.
2. If the invocation on the servant throws exception `$exc`, then the stub code must call `$servant->exceptionalCompletion($exc)` after the invocation completes.
3. In either case, the servant completion call must occur before the `_servant_postinvoke()` call.

Note that an older stub may fail to satisfy these conditions. In this case, any request interceptors that run during local invocations will be unable to correctly report the completion of the request in the PortableInterceptor__RequestInfo reply_status field.

The **_servant_postinvoke()** method is invoked after the operation has been invoked on the local servant. The local stub must pass the instance of the **ServerObject** object returned from the **_servant_preinvoke()** method as an argument. This method must be called if **_servant_preinvoke()** returned a nonnull value, even if an exception was thrown by the servant's method. For this reason, the call to **_servant_postinvoke()** should be placed in a PHP **finally** clause, if it's available. Otherwise the call should be copied before every **return** or **thow** clause inside the **try-catch** block.

1.21.6.4 Invoke Handler

The **org__omg__CORBA__portable__InvokeHandler** interface provides a dispatching mechanism for an incoming call. It is invoked by the ORB to dispatch a request to a servant.

```
interface org__omg__CORBA__portable__InvokeHandler {
   function _invoke($method,
   org__omg__CORBA__portable__InputStream $is,
   org__omg__CORBA__portable__ResponseHandler $handler);
```

```
}
```

The **_invoke()** method receives requests issued to any servant that implements the **InvokeHandler** interface. The **InputStream** contains the marshaled arguments.
The specified **ResponseHandler** will be used by the servant to construct a proper reply.
The only exceptions that may be thrown by this method are CORBA SystemExceptions.
The returned **OutputStream** is created by the **ResponseHandler** and contains the marshaled reply.
A servant shall not retain a reference to the **ResponseHandler** beyond the lifetime of the method invocation.
Servant behavior is defined as follows:

- Determine correct method, and unmarshal parameters from **InputStream**.
- Invoke method implementation.
- If no user exception, create a normal **Reply** using the **ResponseHandler**.
- If user exception occurred, create an exception reply using **ResponseHandler**.
- Marshal reply into **OutputStream** returned by the **ResponseHandler**.
- Return the **OutputStream** to the ORB.

1.21.6.5 Response Handler

The **org__omg__CORBA__portable__ResponseHandler** interface is supplied by an ORB to a servant at invocation time and allows the servant to later retrieve an **OutputStream** for returning the invocation results.

```
interface org__omg__CORBA__portable__ResponseHandler
{
   /**
   * Called by servant during a method invocation.
   * The servant should call
   * this method to create a reply marshal buffer if
   * no exception occurred.
   *
   * Returns an OutputStream suitable for
   * marshalling reply.
   */
   function createReply();
   /**
   * Called by servant during a method invocation.
   * The servant should call
   * this method to create a reply marshal buffer if
   * a user exception occurred.
   *
   * Returns an OutputStream suitable for marshalling
   * the exception ID and the user exception body.
   */
   function createExceptionReply();
}
```

1.21.6.6 Application Exception

The **org__omg__CORBA__portable__ApplicationException** class is used for reporting application level exceptions between ORBs and stubs.
The method **getId()** returns the CORBA repository ID of the exception without removing it from the exception's input stream.

```
class org__omg__CORBA__portable__ApplicationException
     extends php__lang__Exception
{
   public function __construct(
      $id, org__omg__CORBA__portable__InputStream $is)
   {...}
   public function getId()
```

```
   {...}
   public function getInputStream()
   {...}
}
```

The constructor takes the CORBA repository ID of the exception and an input stream from which the exception data can be read as its parameters.

1.21.6.7 Remarshal Exception

The **org__omg__CORBA__portable__RemarshalException** class is used for reporting locate forward exceptions and object forward GIOP messages back to the ORB. In this case the ORB must remarshal the request before trying again. See "Stub Design" on page 1-120 for more information.

```
final class org__omg__CORBA__portable__RemarshalException
   extends php__lang__Exception
{
   public function __construct()
   {
      parent::__construct();
   }
}
```

1.21.6.8 UnknownException

The **org__omg__CORBA__portable__UnknownException** is used for reporting unknown exceptions between ties and ORBs and between ORBs and stubs. It provides a PHP representation of an UNKNOWN system exception that has an **UnknownExceptionInfo** service context.

```
class org__omg__CORBA__portable__UnknownException
   extends org__omg__CORBA__SystemException
{
   public $originalEx;
   public function __construct(
      php__lang__Throwable $ex,
      $message = '',
      $minor_code = 0,
      org__omg__CORBA__portable__CompletionStatus $status = null)
   {
      if ($status === null) {
         $status = org__omg__CORBA__portable__CompletionStatus.
COMPLETED_MAYBE);
      }
      parent::__construct($message, $minor_code, $status);
      $this->originalEx = $ex;
   }
}
```

1.21.7 Delegate Stub

The delegate class provides the ORB vendor specific implementation of CORBA object.

```
// PHP
abstract class org__omg__CORBA__portable__Delegate {
   public abstract function get_interface_def(
      org__omg__CORBA__Object $self);
   public abstract function duplicate(
      org__omg__CORBA__Object $self);
   public abstract function release(
      org__omg__CORBA__Object $self);
   public abstract function is_a(
```

```
      org__omg__CORBA__Object $self,
      $repository_id);
   public abstract function non_existent(
      org__omg__CORBA__Object $self);
   public abstract function is_equivalent(
      org__omg__CORBA__Object $self,
      org__omg__CORBA__Object $rhs);
   public abstract function hash(
      org__omg__CORBA__Object $self, $max);
   public abstract function create_request(
      org__omg__CORBA__Object $self,
      org__omg__CORBA__Context $ctx,
      $operation,
      org__omg__CORBA__NVList $arg_list,
      org__omg__CORBA__NamedValue $result);
   public abstract function create_request2(
      org__omg__CORBA__Object $self,
      org__omg__CORBA__Context $ctx,
      $operation,
      org__omg__CORBA__NVList $arg_list,
      org__omg__CORBA__NamedValue $result,
      org__omg__CORBA__ExceptionList $excepts,
      org__omg__CORBA__ContextList $contexts);
   public abstract function request(
      org__omg__CORBA__Object $self, $operation,
      $responseExpected = true);
   public function invoke(
      org__omg__CORBA__Object $self,
      org__omg__CORBA__portable__OutputStream $os)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function releaseReply(
      org__omg__CORBA__Object $self,
      org__omg__CORBA__portable__InputStream $is)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function get_policy(
      org__omg__CORBA__Object $self,
      $policy_type)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function get_domain_managers(
      org__omg__CORBA__Object $self) {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function set_policy_override(
      org__omg__CORBA__Object $self,
      $policies,
      org__omg__CORBA__SetOverrideType $set_add)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function orb(org__omg__CORBA__Object $self)
   {
      throw new org__omg__CORBA__NO_IMPLEMENT();
   }
   public function is_local(org__omg__CORBA__Object $self)
   {
      return false;
   }
   public function servant_preinvoke(
      org__omg__CORBA__Object $self,
      $operation, $expectedType)
   {
      return null;
```

```
   }
   public function servant_postinvoke(
      org__omg__CORBA__Object $self,
      org__omg__CORBA__portable__ServantObject $servant)
   {
   }
   public function __toString(org__omg__CORBA__Object $self)
   {
      return get_class($self) . ":" . $this->__toString();
   }
   public function hashCode(org__omg__CORBA__Object $self)
   {
      return php__lang__System::identityHashCode($self);
   }
   public function equals(
      org__omg__CORBA__Object $self,
      $obj)
   {
      return ($self==$obj);
   }
}

abstract class org__omg__CORBA_2_3__portable__Delegate
   extends org__omg__CORBA__portable__Delegate
{
   /** Returns the codebase for this object reference.
   * @param self the object reference for which to return
   * the codebase
   * @return the codebase as a space delimited list of url
   * strings or null if none
   */
   public function get_codebase(org__omg__CORBA__Object $self)
   {
      return null;
   }
}
```

1.21.8 Servant Delegate

The Delegate interface provides the ORB vendor specific implementation of **PortableServer::Servant**.

```
interface org__omg__PortableServer__portable__Delegate
{
   function orb(org__omg__PortableServer__Servant $self);
   function this_object(org__omg__PortableServer__Servant $self);
   function poa(org__omg__PortableServer__Servant $self);
   function object_id(org__omg__PortableServer__Servant $self);
   function default_POA(org__omg__PortableServer__Servant $self);
   function is_a(org__omg__PortableServer__Servant $self,
      $repository_id);
   function non_existent(org__omg__PortableServer__Servant
$self);
   function get_interface_def(
      org__omg__PortableServer__Servant $self);
}
```

1.21.9 ORB Initialization

The ORB class represents an implementation of a CORBA ORB. Vendor specific ORB implementations can extend this class to add new features.

There are several cases to consider when creating the ORB instance. An important factor is whether an applet in a browser or an stand-alone PHP application is being used. In any event, when creating an ORB instance, the class names of the ORB implementation are located using the following search order:

- check in Applet parameter, if any
- check in properties parameter, if any
- check in the System properties
- check in orb.properties file, if it exists (Section 1.21.9.2, “orb.properties file,” on page 1-140)
- fall back on a hardcoded default behavior

1.21.9.1 Standard Properties

The OMG standard properties are defined in the following table.

Table 1-3 Standard ORB properties

Property Name	Property Value
org__omg__CORBA__ORBClass	Class name of an ORB implementation.
org__omg__CORBA__ORBSingletonClass	Class name of the singleton ORB implementation.

1.21.9.2 orb.properties file

The **orb.properties** file is an optional file. The search order for the file is:
1. The user’s home directory, given by the user.home system property.
2. The **<php-home>/lib** directory, where **<php-home>** is the value of the System property **php.home**.

It consists of lines of the form **<property-name>=<property-value>**.
See Table 1-3 for a list of the property names and values that are recognized by ORB::init. Any property names not in this list shall be ignored by **ORB::init()**. The file may also contain blank lines and comment lines (starting with #), which are ignored.

1.21.9.3 ORB Initialization Methods

There are three forms of initialization as shown below. In addition the actual ORB implementation (subclassed from **ORB**) must implement the **set_parameters()** methods so that the initialization parameters will be passed into the ORB from the initialization methods.

```
// PHP
abstract class org__omg__CORBA__ORB {
   // Application init
   public static function init($args = null, $props = null)
   {
      if (is_array($args) && is_array($props)) {
         // call to: set_parameters($args, $props);
      }
      ...
   }

   // Implemented by subclassed ORB implementations
   // and called by init methods to pass in their params
   abstract protected function set_parameters($args, $props);
}
```

Default initialization
The default initialization method returns the singleton ORB. If called multiple times it will always return the same PHP object.

The primary use of the no-argument version of **ORB::init()** is to provide a factory for **TypeCode**s for use by Helper classes implementing the **type()** method, and to create **Any** instances that are used to describe union labels as part of creating a union **TypeCode**.

The following list of ORB methods are the only methods that may be called on the singleton ORB. An attempt to invoke any other ORB method shall raise the system exception NO_IMPLEMENT.

- **`create_xxx_tc(), where xxx is one the defined typecode types`**
- `get_primitive_tc()`
- `create_any()`

Application initialization
The application initialization method should be used from a stand-alone PHP application. It is passed an array of strings that are the command arguments and a list of PHP properties. Either the argument array or the properties may be **null**.
It returns a new fully functional ORB PHP object each time it is called.

1.22 PHP Mapping for CORBA Messaging

1.22.1 Introduction

The CORBA Messaging specification creates new requirements for the PHP mapping.The PHP mapping must define what code needs to be generated for sendc and sendp operations consistent with the current standards for ordinary static invocations, and the standard APIs that must be supported for portable AMI stubs. The work is still in progress, so this chapter is largely a placeholder in the current version of this specification.

1.22.2 Mapping of Native Types

Messaging::UserExceptionBase is mapped to org__omg__CORBA__UserException.

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O