.NET CLS and CORBA IDL Interoperability

IDL ⇆ CLS Mapping Specification
IIOP.NET

Imput.	Report	Version	Date	Author(s)	Status	Visa
		1.02	09.01.04	DUL, PRR		

I. Table of Content

I. Table of Content 3

II. Record of Changes 6

III. References 6

IV. Abbreviations 6

V. Preface 7

1 Overview 8
1.1 Mapping Definitions and Restrictions 8
1.2 Type System's Definitions 8
1.2.1 CLS 8
1.2.2 OMG IDL 8
1.3 Document Structure 8
1.4 Future Work 8

2 Mapping of CLS Types 10
2.1 The IDL Subset of .NET CLS 10
2.1.1 Conforming Types 11
2.1.2 Non-conforming Types 11
2.1.3 Non-mappable Types 11
2.2 Mapping CLS Names to IDL Names 12
2.2.1 Mapping CLS Namespace Names to Module Names 12
2.2.2 CLS Names that Clash with IDL Keywords 12
2.2.3 CLS Names with Leading Underscores 12
2.2.4 CLS Names with Illegal IDL Identifier Characters 12
2.2.5 Names for Inner Classes 12
2.2.6 Overloaded Method Names 12
2.2.7 Names Differing only in Case 13
2.3 Mapping for Namespaces 13
2.3.1 Example 13
2.4 Mapping for Primitive Types 13
2.5 Events 14
2.6 Delegates 14
2.7 Constants 14
2.8 Conforming Reference Types (MarshalByRefObject and Subclasses) 14
2.8.1 Special Case for MarshalByRefObject 14
2.8.2 Inherited Interfaces 14
2.8.3 Inherited Base Class 14
2.8.4 Constructors 14
2.8.5 Fields 14
2.8.6 Properties 15
2.8.7 Methods 15
2.8.8 Repository ID 15
2.8.9 Special Cases 15
2.8.10 Example 15
2.9 Conforming Value-Types 16
2.9.1 Inherited Base Class 16
2.9.2 Inherited Interfaces 17

2.9.3 Methods ... 17
2.9.4 Constructors ... 17
2.9.5 Fields ... 17
2.9.6 Properties ... 17
2.9.7 Repository ID ... 17
2.9.8 Special Cases ... 17
2.9.9 Example ... 18
2.10 System.Exception and Subclasses ... 19
2.10.1 GenericUserException ... 19
2.11 Enum ... 19
2.11.1 Example ... 19
2.12 Arrays ... 20
2.12.1 One-dimensional Arrays ... 20
2.12.2 Multi-dimensional Arrays ... 20
2.12.3 Example ... 20
2.13 Non-conforming Types ... 21
2.13.1 System.Object ... 21
2.13.2 System.ValueType ... 21
2.13.3 Interfaces ... 21
2.13.4 Non-conforming Value Types ... 22
2.14 CLS Attributes ... 22
2.15 Preventing Redefinition of Types ... 23
2.15.1 Example ... 23

3 Mapping of OMG IDL Types ... 24
3.1 Names ... 24
3.1.1 Names that Clash with C# and VB.NET Identifiers ... 24
3.2 Mapping of Module ... 24
3.2.1 Example ... 24
3.3 Basic Types ... 24
3.3.1 Unsigned Integral Types ... 25
3.3.2 char, wchar, string, and wstring ... 25
3.4 IDLEntity Interface ... 26
3.5 Enum ... 26
3.5.1 Example ... 26
3.6 Struct ... 26
3.7 Union ... 27
3.7.1 Example ... 28
3.8 Sequence ... 29
3.9 Array ... 30
3.10 Interface ... 30
3.10.1 Inheritance Hierarchy Mapping ... 31
3.10.2 IDL-Attributes ... 31
3.10.3 Methods ... 31
3.10.4 #pragma ID ... 31
3.10.5 Example ... 32
3.10.6 CORBA::Object ... 32
3.10.7 CORBA::AbstractBase ... 32
3.11 Value Types ... 33
3.11.1 #pragma ID ... 33
3.11.2 Inheritance ... 33

3.11.3 Methods 33
3.11.4 IDL-Attributes 33
3.11.5 Concrete Value Types 33
3.11.6 Abstract Value Types 34
3.11.7 Boxed Value Types 35
3.11.8 CORBA::ValueBase 37
3.12 Exceptions 37
3.12.1 Mapping of CORBA Exceptions 37
3.12.2 Mapping of user defined Exceptions 39
3.13 Any 39
3.14 Nested Types 40
3.15 TypeDef 40
3.16 Constants 40
3.17 Used Attributes 41

4 Marshalling / Unmarshalling 43
4.1 Marshalling of Parameters During Methods Calls 43
4.2 Marshalling / Unmarshalling of Value-Types 44

II. Record of Changes

Filename	Version	Date	Description / Author
CLSIDLMappingSpec.DOC	1.02	09.01.04	Added IDL bounded sequences support / DUL
CLSIDLMappingSpec.DOC	1.01	29.09.03	Added IDL union and IDL constants support / DUL
CLSIDLMappingSpec.DOC	1.00	10.06.03	First release / PRR
CLSIDLMappingSpec.DOC	0.93	1.06.03	Revision after internal review / PRR
CLSIDLMappingSpec.DOC	0.92	15.05.03	Revision / PRR
CLSIDLMappingSpec.DOC	0.91	22.04.03	Revision / PRR
CLSIDLMappingSpec.DOC	0.9	16.03.03	First draft / DUL

III. References

[1] OMG, **The Common Object Request Broker: Architecture and Specification**; version 2.3.1; www.omg.org

[2] OMG; **Java to IDL Language Mapping Specification**; version 1.2 www.omg.org

[3] OMG; **IDL to Java Language Mapping Specification**; version 1.2 www.omg.org

[4] ECMA, **ECMA-335 Common Language Infrastructure (CLI)** http://www.ecma-international.org/

IV. Abbreviations

CLS	(.NET) Common Language Specification
CLR	(.NET) Common Language Runtime
CORBA	Common Object Request Broker Architecture
ECMA	European Computer Manufacturers Association
IDL	(CORBA) Interface Definition Language
IIOP	(CORBA) Internet Inter-ORB Protocol
OMG	Object Management Group
ORB	(CORBA) Object Request Broker
RMI	(Java) Remote Method Invocation

V. Preface

About ELCA

ELCA Informatique SA was founded in 1968. It is head-quartered in **Lausanne** with branch offices in Zürich, Geneva, Bern, London and Ho Chi Minh City (Vietnam).

With over 350 top-level engineers who mainly graduated from the Federal Institutes of Technology and other famous Universities, ELCA is one of the leading Swiss information technology services companies. ELCA has been ISO 9001 certified since 1993.

ELCA’s vital statistics:

- 21 nationalities
- 20 languages
- 40 Ph.D.s
- 11 university disciplines
- Software used in 34 countries

ELCA's areas of expertise are :

- Business Information systems
- Data Warehouse
- Decision Support System
- Architectures and distribute systems
- Content and electronic document management
- CRM & Business Intelligence
- Enabling e-business
- Internet / Intranet / Web factory
- Industrial systems
- Project support and supervision

Acknowledgements

This work was realized in collaboration with Prof. J. Gutknecht (Programming Languages and Runtime Systems Research Group), ETH Zürich, and is based on Dominic Ullmann’s diploma thesis.

Patrik Reali and Dominic Ullmann would like to thank Alain Borlet-Hote, Christian Gasser, Philipp Oser, and Bernhard Rytz for reviewing this document and for their many and valuable comments.

1 Overview

This document describes the mapping between the OMG IDL type system [1] and the .NET CLS type system [2].

1.1 Mapping Definitions and Restrictions

This specification specifies a mapping for most of the IDL entities, respectively CLS types. We distinguish between pure mappings and extended mappings.

Pure mappings rely on types having the same domain (i.e. set of values) in both type systems, such that no runtime checks are needed in the conversion, and a value for a given type is guaranteed to exist in the mapped type.

Extended mappings are defined for some commonly used types (e.g. strings), where the source and the destination type have slightly different domains, and thus some values may not be mappable. These mappings require extra care and must be checked at runtime for correctness.

Finally, a few types have no mapping, because no appropriate type exists in the destination type system (see 2.1.3).

1.2 Type System's Definitions

1.2.1 CLS

The ECMA 335 standard version 1 [2, chap. 7-11] defines the Common Language Specification (CLS) and the CLS type system.

1.2.2 OMG IDL

The OMG CORBA and OMG IDL specification (version 2.3.1) [1, chap. 3] defines the OMG IDL type system. We use version 2.3.1, because this is same version as used by the Java RMI/IIOP protocol.

1.3 Document Structure

In chapter 2, the mapping from CLS to IDL is specified.

In chapter 3, the mapping from IDL to CLS is specified.

Chapter 4 covers the marshalling and unmarshalling of types. This chapter is added to show, that the specified mapping is adequate to support serialisation and deserialisation of parameters.

1.4 Future Work

This mapping is not exaustive. A few mapping have not been investigated yet, and will be clarified in a future release of the mapping. For the moment, these types and constructs are considered "non-mappable":

- CLS System.Decimal
- CLS Class fields

- IDL fixed, IDL long double
- IDL valuetype initializers
- IDL truncatable valuetypes
- IDL arrays
- IDL type codes

Furthermore, this specification shall be upgraded to the latest CORBA specification.

2 Mapping of CLS Types

This chapter describes the mapping of .NET CLS to OMG IDL (version 2.3.1).

2.1 The IDL Subset of .NET CLS

This section defines the subset of the CLS that is mapped to the IDL and is thus available for use with the IIOP Protocol.

When dealing with a type system conversion, we must distinguish between converting type definitions (mapping) and converting instances (marshalling). The table below gives an overview of the relationships between convertions, types, and instances.

Type t		*Instance of t*
Mappable	conforming	marshallable
	non-conforming	non-instantiable
		non-marshallable
non-mappable		

Obviously, types can be parted in mappable and non-mappable, and instances in marshallable and non-marshallable. Less intuitive is, that those sets are different, i.e. that not all instances of a mappable type are also marshallable. The reason for this lies in the characteristics of the object-oriented type systems, which require an instance type to be compatible (and not strictly equal) to the declared type.

The type system mapping takes place at two distinct moments, which reflect this distinction: at design and compilation time, the type definitions are mapped; at runtime the object instances are marshalled.

The CLS types are divided into conforming, non-conforming, and non-mappable types. Instances of conforming types can be mapped to the IDL, whereas instances of non-mappable types cannot.

Non-conforming types make a third category in-between: these types can be mapped whereas their instances cannot be instantiated or cannot be marshalled (but subclasses thereof may be marshalled).

Example: an interface defines a set of methods to be implemented by a class. Interfaces are mostly mappable, but always non-instantiable. It must be resolved at run-time, whether the object implementing the interface is marshallable or not.

Example: System.Object is non-conforming type: it is mappable to the IDL, but instances thereof are not marshallable; however there exists types compatible to System.Object that are conforming types: whether a parameter of type System.Object can be marshalled or not, depends on the actual type of the object, which must be conforming.

2.1.1 Conforming Types

Conforming types have a mapping to OMG IDL, are instantiable, and instances of these types can be serialized and deserialized.

The CLS conforming types are

- the primitive types `System.Boolean, System.Byte, System.Char, System.Double, System.Int16, System.Int32, System.Int64, System.Single, System.String, System.Void` (see section 2.4)
- the conforming reference types`:` `MarshalByRefObject` and subclasses thereof (see section 2.5)
- the conforming value types: all subclasses of `MarshalByValueComponent`, and all serializable structures and classes (see section 2.9).
- the CLS exceptions (see section 2.10)
- the `Int32`-based enumerations with no associated constant values (see section 2.11)
- the arrays of conforming types (see section 2.12)

2.1.2 Non-conforming Types

Non-conforming types have a mapping to OMG IDL; variables declared with a non-conforming type can be serialized and deserialized only when the dynamic type is a conforming type. Non-conforming types usage is restricted to method signatures, field declarations and inheritance relationships.

The following CLS types are non-conforming:

- `System.Object` (section 2.13.1)
- `System.ValueType` (section 2.13.2)
- All interface types (section 2.13.3)
- Non-conforming value types (section 2.13.4)
  All mappable CLS types, which do not belong to another category are considered as non-conforming value types.

2.1.3 Non-mappable Types

Non-mappable CLS types cannot be mapped to the IDL. The following types are non-mappable:

- `System.IntPtr, System.Decimal`
- Enumeration with `int64` based type, marked as `flags`, or with associated constant values. (section 2.11).

IntPtr is not mappable, because pointers cannot be mapped across different domains; Decimal will be addressed in the future; IDL enumerations can only be based on int32 types.

2.2 Mapping CLS Names to IDL Names

In general, CLS names are mapped to the equivalent name in OMG IDL. However, there are some exceptions (listed below) when the CLS name is not a legal identifier in OMG IDL.

2.2.1 Mapping CLS Namespace Names to Module Names

Each namespace is mapped to an IDL module; nested namespaces are mapped to nestes IDL modules (see Section 2.3).

A CLS namespace `a.b.c` is mapped to an OMG IDL module `::a::b::c` (a leading "`::`" is added and each separation dot is replaced by two separation semicolons).

2.2.2 CLS Names that Clash with IDL Keywords

CLS Names that clash with an IDL keyword are mapped to OMG IDL by adding a leading underscore. The CLS name `oneway` is mapped to the OMG IDL identifier `_oneway` (an escaped identifier).

2.2.3 CLS Names with Leading Underscores

For CLS Names that have leading underscores, the leading underscore is replaced with "`N_`". Thus, `_fred` is mapped to `N_fred`. CLS Names beginning with "`N_`" cannot be mapped.

2.2.4 CLS Names with Illegal IDL Identifier Characters

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

Any illegal character is replaced by "U" followed by 4 hexadecimal characters (in upper case) representing the character's Unicode value.

2.2.5 Names for Inner Classes

The name of an inner class is mapped to a composite name formed by the name for the outer class, two underscores, and the name of the inner class.

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

2.2.6 Overloaded Method Names

If a CLS method is not overloaded, then the same method name is used in OMG IDL as in CLS.

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

For overloaded methods, the mangled OMG IDL name is formed by taking the CLS method name, appending "__" followed by the fully qualified OMG IDL types of each of the arguments separated by two underscores; leading "`::`" and underscores are removed, embedded "`::`" and spaces are replaced with "__".

2.2.6.1 Example

C#	IDL

`Class C : MarshalByRefObject {`
`    Void Meth(int a);`
`    char Meth(float x);`
`}`
	`interface C {`
	`  void Meth__long(long a);`
	`  wchar Meth__float(float x);`
	`};`

Both methods in the IDL raise `::Ch::Elca::Iiop::GenericUserException;` in the example the raise clause is omitted for clarity.

2.2.7 Names Differing only in Case

OMG IDL doesn't support case sensitive names. Names differing only in case are not supported.

2.3 Mapping for Namespaces

Each namespace is mapped to an IDL module; nested namespaces are mapped to nestes IDL modules.

2.3.1 Example

CLS (C#)	IDL

`namespace x.y {`
`   // …`
`}`
	`module x {`
	`  module y {`
	`     // …`
	`  };`
	`};`

2.4 Mapping for Primitive Types

The mapping for CLS primitive types is as follows:

CLS (class library name)	**OMG IDL**
`System.Boolean`	`Boolean`
`System.Char`	`wchar`
`System.String`	`::CORBA::WStringValue`
`System.Single`	`float`
`System.Double`	`double`
`System.Int16`	`short`
`System.Int32`	`long`
`System.Int64`	`long long`
`System.Byte`	`octet`
`System.Void`	`void`
`System.IntPtr`	not mappable

`System.String` is mapped to the boxed value type `::CORBA::WstringValue`, because in the domain of the IDL primitive type wstring, null is missing. Furthermore the semantic of `System.String` is better represented by an IDL valuetype, than by a primitive type.

2.5 Events

Events are not mapped to the OMG IDL, because there is no corresponding type.

2.6 Delegates

Delegates are not mapped to the OMG IDL, because there is no corresponding type.

2.7 Constants

Constants are not mapped to the OMG IDL, they will be treated in a future version of the mapping.

2.8 Conforming Reference Types (MarshalByRefObject and Subclasses)

Object instances compatible to `MarshalByRefObject` can be invoked remotely in the .NET Remoting framework. These objects are remotely represented with references to the original object, which resides in a .NET managed environment.

A conforming reference type is mapped to an OMG IDL `public interface` in an OMG IDL module corresponding to the type's namespace; the type's name is mapped according to 2.2.

2.8.1 Special Case for MarshalByRefObject

As a special case, any explicit use of MarshalByRefObject is mapped to the OMG IDL type `CORBA::Object`.

All conforming classes inherit from `MarshalByRefObject`. This inheritance is represented in the mapping as the implicit inheritance of IDL interface types from `CORBA::Object`.

2.8.2 Inherited Interfaces

Each inherited interface in the CLS class is represented by an equivalent inherited abstract interface in the OMG IDL (see Section 2.13.3).

2.8.3 Inherited Base Class

Each inherited class other than `MarshalByRefObject` is represented by an equivalent inherited interface in the OMG IDL (see Section 2.8).

2.8.4 Constructors

Constructors are not mapped to IDL.

2.8.5 Fields

Fields are not mapped to IDL, they will be treated in a future version of the mapping.

2.8.6 Properties

CLS properties are mapped to OMG IDL attributes. The property name is mapped according to Section 2.2, the property type is mapped to the corresponding IDL type.

2.8.6.1 Read-only Properties

Properties having only the `get` accessor are read-only properties. The mapped OMG IDL attribute is tagged with the `readonly` attribute.

2.8.6.2 Write-only Properties

Properties having only the `set` accessor are write-only properties. These properties cannot be mapped to the OMG IDL.

2.8.7 Methods

Public instance methods are mapped to OMG IDL methods in the corresponding interface where:

1. the OMG IDL method name is generated as described in Section 2.2.
2. the CLS return type is mapped to the corresponding OMG IDL return type.
3. each parameter is mapped to an OMG IDL parameter with the corresponding OMG IDL type; the CLS parameter modifiers are mapped as follows:

CLS parameter modifier	*IDL parameter modifier*
ref	*in out*
out	*out*
default	*In*

4. the raise clause throws the generic exception `Ch::Elca::Iiop::GenericUserException.`

Only the public instance methods defined in the current class are mapped; method inherited from other classes or interfaces are defined in the type that defines them.

Non-public instance methods and static methods are not remotely accessible, thus they are not mapped.

2.8.8 Repository ID

A repository id is assigned to the mapped OMG IDL interface with a `#pragma ID` directive.

The repository ID is in OMG IDL format, as specified in [1], *Interface Repository* chapter, Section 10.6.1, page10-39.

2.8.9 Special Cases

Types implementing the marker interface IDLEntity are not mapped to the OMG IDL, because they are CLS conversions of types already defined in OMG IDL.

2.8.10 Example

```
// C#
namespace a.b.c {
  public interface OPInterface1 {
      public void method1(int a, int b);
  }

  public class Class1 : MarshalByRefObject, OPInterface1 {
      public void method1(int a, int b) {
          // some impl
      }
      public MarshalByRefObject method2() {
          // some impl
      }
  }
}

// IDL
module a {
module b {
module c {
  abstract interface OPInterface1 {
      void method1(in long a, in long b) raises
      (::Ch::Elca::Iiop::GenericUserException);
  };
  #pragma ID OPInterface1 "IDL:a/b/c/OPInterface1:1.0"
  interface Class1 : OPInterface1 {
      Object method2()raises
      (::Ch::Elca::Iiop::GenericUserException);
  };
  #pragma ID Class1 "IDL:a/b/c/Class1:1.0"

};
};
};
```

2.9 Conforming Value-Types

Conforming value types are user-defined conforming types, whose instances are passed by value.

Value types may be passed as arguments or results of remote methods, or as fields within instances of other value types.

Conforming value types are

- concrete CLS classes and structs marked with the attribute `Serializable`
- concrete CLS subclasses of `MarshalByValueComponent`.

Conforming value types are mapped to an OMG IDL `valuetype` with the corresponding OMG IDL name (see Section 2.2) in the OMG IDL module corresponding to the CLS type's namespace.

2.9.1 Inherited Base Class

Each inherited class other than `MarshalByValueComponent`, `System.Object`, or `System.ValueType` is represented in the OMG IDL by an equivalent inherited abstract or concrete valuetype.

2.9.2 Inherited Interfaces

Each inherited interface in the CLS class is represented by an equivalent inherited abstract interface in the OMG IDL (see Section 2.13.3).

2.9.3 Methods

Only non-private methods are mapped to the OMG IDL. The rules for conforming reference types apply (see Section 2.8.7) but for the raise clause, which is left empty.

2.9.4 Constructors

Constructors are not mapped to OMG IDL. Remote objects are accessed through interfaces, which do no support constructors.

2.9.5 Fields

Non-private fields are mapped to public IDL state members, private one's are mapped to private members. The type of the state member is the mapped type of the CLS field.

The ordering of the fields in the IDL defines the order, in which the fields are serialized, thus the IDL states must be declared in the same order as in the CLS type.

Exceptions

A field is not mapped, if

- the field type is not mappable
- the field has the `NonSerializable` attribute

2.9.6 Properties

Properties are mapped according to the rules in Section 2.8.6.

2.9.7 Repository ID

Repository ID are mapped according to the rules in Section 2.8.8.

2.9.8 Special Cases

For the following conforming value types, a special mapping is provided:

- `System.Type`

Furthermore, types implementing the marker interface IDLEntity are not mapped to the OMG IDL, because they are CLS conversions of types already defined in OMG IDL.

2.9.8.1 System.Type

System.Type contains the metadata information about a given CLS type; this information is useless for CORBA, because it relies on a different type system; the corresponding information is the repository-ID (or an IDL type-code). Therefore `System.Type` is mapped to the following IDL-Type:

```
module System {
  valuetype CORBATypeDesc {
      public ::CORBA::WStringValue repositoryID;
  };
  #pragma ID CORBATypeDesc "IDL:System/CORBATypeDesc:1.0"
};
```

Only classes and structs can be represented using this mapping.

Note: in an future mapping, IDL type-codes should be used instead, because they can describe every type.

2.9.9 Example

```
// .NET (C#)
namespace a{
  [Serializable]
  public class TestBase {
      public TestBase(int x) {
          m_x = x;
      }
      public int m_x;
      public void inc() {
          m_x++;
      }
  }

  [Serializable]
  public class TestValue : TestBase, IComparable {
      public int CompareTo(Object obj) {
          // …
      }

      public void add(int y) {
          // …
      }
  }

}

// IDL
module System {
  abstract interface IComparable {
      long CompareTo(any obj) raises
      (::Ch::Elca::Iiop::GenericUserException);
  };
  #pragma ID IComparable "IDL:System/IComparable:1.0"

};
module a {
  valuetype TestBase {
      public long m_x;
      void inc();
  };
  #pragma ID TestBase "IDL:a/TestBase:1.0"

  valuetype TestValue : TestBase supports IComparable {
      void add(long y);
  };
  #pragma ID TestValue "IDL:a/TestValue:1.0"

};
```

2.10 System.Exception and Subclasses

Exceptions are problematic, because CLS methods do not have a raise clause to specify which exceptions are thrown. Therefore every exception occurring during a remote method invocation must be mapped to one of the standard CORBA-Exceptions or to a `GenericUserException`.

2.10.1 GenericUserException

GenericUserException is defined as follows:

```
Module Ch::Elca::Iiop {
  exception GenericUserException {
    ::CORBA::WStringValue name;
    ::CORBA::WstringValue message;
    ::CORBA::WStringValue throwingMethod;
  }
}
```

2.11 Enum

Enumerations based on a type other than `System.Int16` and `System.Int32` (the default base type), with assigned constant values, or with the `Flags` (bitfields) attributes are not mappable to the OMG IDL.

CLS Enums are mapped to the OMG IDL `enum` type. The type name is mapped according to Section 2.2.

The named constant names are mapped to an OMG IDL name and prefixed with the IDL enumeration name followed by an underscore.

2.11.1 Example

```
// .NET (C#)
enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};

//IDL
enum Days {Days_Sat, Days_Sun, Days_Mon, Days_Tue, Days_Wed,
Days_Thu, Days_Fri};
```

2.12 Arrays

2.12.1 One-dimensional Arrays

CLS one-dimensional array types are mapped to an IDL boxed value type containing a sequence, the sequence type is the mapped array element type.

Jagged arrays (arrays of references to other array) are one-dimensional arrays of references.

The name of the boxed value-type is the concatenation of “`seq<n>_`” and the mapped array-element type, where <n> is the number of dimensions of the array type.

The array dimension is defined as:

- 1, for arrays whose element is not an array.
- 1+d, for arrays whole element is an array of dimension d.

For primitive types, the boxed sequence is declared in the module `::org::omg:boxedArray`; otherwise the module is `::org::omg:boxedArray` concatenated with module name of the element type. Thus, an array with base type ::a::b::c is defined in the module `::org::omg::boxedArray::a::b`.

The repository ID is assigned according to Section 2.8.8.

Note: At runtime, the size of the actual array instance is contained in the instance of the sequence.

2.12.2 Multi-dimensional Arrays

Multi-dimensional array are treated as jagged array.

Runtime checks must be performed, when a muli-dimensional array is returned from another ORB, to ensure that the array is legal.

2.12.3 Example

```
// C#
int[][]
a.b.C[]
```

```
// IDL
module org {
module omg {
module BoxedArray {
  module System {
      valuetype seq1_long sequence<long>;
      #pragma ID seq1_long
      "IDL:org/omg/BoxedArray/System/seq1_long:1.0"
      valuetype seq2_long
      sequence<::org::omg::BoxedArray::System::seq1_long>;
      #pragma ID seq2_long
      "IDL:org/omg/BoxedArray/System/seq2_long:1.0"
  };
  module a {
  module b {
      valuetype seq1_C sequence<::a::b::C>;
      #pragma ID seq1_C
      "IDL:org/omg/BoxedArray/a/b/seq1_C:1.0"
  };
  };
};
};
};
```

2.13 Non-conforming Types

Non-conforming types can be mapped to the OMG IDL, but only instances which are also compatible to a conforming type can be serialized.

Non-conforming types can only appear in an inheritance relationship, in a method signature, in a field declaration, and in a property declaration.

2.13.1 System.Object

`System.Object` is mapped to OMG IDL `any`.

A `System.Object` in a method signature means that any conforming type can be passed (an object reference or a value).

As an exception, an inheritance relationship with `System.Object` is not mapped to IDL.

2.13.2 System.ValueType

`System.ValueType` is only allowed in inheritance relationships, to qualify a class as value type. The inheritance relationship with `System.ValueType` is not mapped to IDL.

2.13.3 Interfaces

A CLS interface is mapped to an abstract OMG IDL interface having a name mapped according to Section 2.2.

The interface's methods are mapped according to Section 2.8.6, the properties according to Section 2.8.6.

2.13.3.1 Example

```
// C#
public interface DNDoubleInterface {
  double addDoubleVal(double val1, double val2);
  void incDoubleVal(ref double val);
}

// IDL
abstract interface DNDoubleInterface {
  void incDoubleVal(inout double val) raises
  (::Ch::Elca::Iiop::GenericUserException);
  double addDoubleVal(in double val1, in double val2) raises
  (::Ch::Elca::Iiop::GenericUserException);
};
#pragma ID DNDoubleInterface
"IDL:testDN/DNDoubleInterface:1.0"
```

2.13.4 Non-conforming Value Types

Non conforming value types are mapped to an abstract IDL value type with the corresponding OMG IDL name. The repository id is assigned according to Section 2.8.8.

The inheritance relationships are reflected in IDL

Type Members

Only public methods are mapped to OMG IDL. These methods are mapped according to Section 2.8.7.

2.13.4.1 Example

In this example NonConfValue is not a conforming value type, because it is abstract. NonConfValue2 is not conforming, because the Serializable attribute is missing.

```
// C#
[Serializable]
public abstract class NonConfValue {
  public int b;
}

public class NonConfValue2 {
  public int a;
  public void printA() {
      Console.WriteLine("a: " + a);
  }
}

// IDL
abstract valuetype NonConfValue {
};

#pragma ID NonConfValue "IDL:testDN/NonConfValue:1.0"

abstract valuetype NonConfValue2 {
  void printA();
};

#pragma ID NonConfValue2 "IDL:testDN/NonConfValue2:1.0"
```

2.14 CLS Attributes

Attributes are not mapped to OMG IDL.

2.15 Preventing Redefinition of Types

IDL-files support preprocessor directives for inclusion of external definition files. To prevent type redefinition due to multiple file inclusion, all type definitions are protected by an `#ifndef #endif` block.

The identifier used for the block starts with two underscores and is followed by the fully scoped IDL-name of the type with every "`::`" replaced by "_". The identifier ends with two underscores.

Forward declarations are not protected.

2.15.1 Example

```
abstract interface DNDoubleInterface; /*forward declaration*/

#ifndef __testDN_DNDoubleInterface__
#define __testDN_DNDoubleInterface__
module testDN {

  abstract interface DNDoubleInterface {
      void incDoubleVal(inout double val) raises
      (::Ch::Elca::Iiop::GenericUserException);
      double addDoubleVal(in double val1, in double val2)
      raises
      (::Ch::Elca::Iiop::GenericUserException);
  };

  #pragma ID DNDoubleInterface
  "IDL:testDN/DNDoubleInterface:1.0"

};
#endif
```

3 Mapping of OMG IDL Types

This section describes the mapping of the OMG-IDL type system to the CLS type system.

3.1 Names

OMG IDL names are mapped to the equivalent name in CLS. However, there are some exceptions (listed below) when the OMG IDL name is not a legal identifier in CLS.

3.1.1 Names that Clash with C# and VB.NET Identifiers

The CLS defines no keywords. Of the many source languages used to create CLS definitions, we consider only C# and VB.NET, due to their wide acceptance.

If the mapped identifier clashes with a C# or VB.NET identifier, it must be prefixed with an underscore.

3.2 Mapping of Module

Each OMG IDL module is mapped to a CLS namespace; nested modules are mapped to nestes namespaces.

An OMG IDL module `::a::b::c` is mapped to a CLS namespace `a.b.c`.

3.2.1 Example

OMG IDL	CLS(using C# notation)

`module x {`
`  module y {`
`     // …`
`  };`
`};`
	`namespace x.y {`
	`  // …`
	`}`

3.3 Basic Types

The following table shows the basic mapping. In the cases, where the source domain is different (usually smaller) than the destination domain, the third column gives the exception to raise.

The potential mismatch can occur when the domain of the CLS type is different than IDL. The value must be effectively checked at runtime when it is marshaled as an in parameter (or on input for an inout). For example, `System.Char` is a superset of IDL `char`.

Hint: Users should be careful when using unsigned types in IDL. Because these types are mapped to signed types, a user is reponsible for ensuring that large unsigned IDL type values are handled correctly as negative integers in CLS.

IDL-Type	*CLS Type (class library name)*	*Exceptions*
`boolean`	`System.Boolean`	

char	System.Char	CORBA::DATA_CONVERSION
wchar	System.Char	CORBA::DATA_CONVERSION
octet	System.Byte	
string	System.String	CORBA::DATA_CONVERSION CORBA::MARSHAL
wstring	System.String	CORBA::DATA_CONVERSION CORBA::MARSHAL
short	System.Int16	
unsigned short	System.Int16	
long	System.Int32	
unsigned long	System.Int32	
long long	System.Int64	
unsigned long long	System.Int64	
float	System.Single	
double	System.Double	
long double	not present in CLS	
fixed	not present in CLS	
void	System.Void	

The `long double` (IEEE double-extended 80bits floating point) and `fixed` IDL types cannot be mapped, because there is no CLS type with similar range and properties.

3.3.1 Unsigned Integral Types

The CLS has only signed integral types (the CLR has unsigned integrals). Therefore, unsigned integral types must be mapped to signed ones.

3.3.2 char, wchar, string, and wstring

The mapping of `char` to `System.Char` is an extended mapping, because the value range of `char` is a subset of the range of `System.Char`. This is a problem when values are converted from the CLS (e.g. in the case of return types or varargs), because the source value could lie outside of the destination type's legal range.

Mapping a `string` to a `System.String` suffers from exactly the same problems as the mapping from `char` to `System.Char`.

To handle `char` (respectively `string`)conversions correctly, each field and parameter of type `System.char` (respectively `System.String)` has the boolean attribute `WideCharAttribute`. With this additional information, the marshaller can serialize and deserialize `string/wstring` correctly. Runtime checks prevent passing of illegal values.

Furthermore, the attribute `StringValueAttribute` is used to recognize `System.String` parameters or fields defined in OMG IDL. For these types, the value `null` is not legal.

```
  [AttributeUsage(AttributeTargets.Parameter |
AttributeTargets.ReturnValue | AttributeTargets.Field |
AttributeTargets.Property)]
  public sealed class WideCharAttribute :
                          Attribute, IDLAttribute {
```

```
        public WideCharAttribute(bool isAllowed);
        public bool IsAllowed {get;}
    }

    [AttributeUsage(AttributeTargets.Parameter |
AttributeTargets.ReturnValue | AttributeTargets.Field |
AttributeTargets.Property)]
    public sealed class StringValueAttribute :
                         Attribute, IDLAttribute {
    }
```

3.4 IDLEntity Interface

All generated CLS conform classes and structures must inherit from `IDLEntity`.This marker empty interface `IDLEntity` is used in CLS to recognize all types generated from IDL, and avoid their accidental remapping to the IDL.

3.5 Enum

The OMG IDL `enum` type is mapped to a CLS `enum` type with implicit underlying datatype `System.Int32`; the enum name and the enum element names are mapped according to Section 3.1.

An CLS attribute `IdlEnumAttribute` is used to specify, that a CLS enum is mapped from an IDL enum.

```
    [AttributeUsage(AttributeTargets.Enum)]
    public class IdlEnumAttribute : Attribute, IDLAttribute {
        public IdlEnumAttribute();
    }
```

3.5.1 Example

```
// IDL
enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};

//C#
[IdlEnumAttribute]
enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
```

3.6 Struct

OMG IDL `structs` are mapped to CLS `structs` inheriting from `IDLEntity` and having the attribute `IdlStructAttribute`. The name is mapped according to Section 3.1.

The fields of the IDL struct and of the mapped CLS struct must be declared in the same order. Each field is mapped according to name and type mapping rules.

```
    [AttributeUsage(AttributeTargets.Struct)]
    public class IdlStructAttribute : Attribute, IDLAttribute {
        public IdlStructAttribute();
```

```
}
```

3.7 Union

An OMG IDL `union` is mapped to a CLS `struct` implementing `IDLEntity` and having the attribute `IdlUnionAttribute`. The name of the CLS struct is the mapped IDL union name as described in section 3.1.

```
[AttributeUsage(AttributeTargets.Struct)]
public class IdlUnionAttribute : Attribute, IDLAttribute {
    public IdlUnionAttribute();
}
```

The struct has a parameterless default constructor.

The struct contains a private field `m_discriminator` to store the discriminator value. The type of field `m_discriminator` is the mapped idl discriminator type.

and it contains a private field `m_value` to store the union value.and the field `m_value` is of type System.Object.

A `public` property `Discriminator` provides read-only access to the discriminator value. The property has the type of the discriminator.

A switch-case defines one or more case labels and the corresponding union element. For every switch-case the following is generated:

- A `public` accessor method with the name `Get` followed by the mapped element name.
  This method retrieves the union value. If one of this Get methods is called, although the union holds a value belonging to another switch-case, a `BAD_OPERATION` exception with minor code 34 should be throwed.
- A `public` modifier method with the name Set followed by the mapped element name.
  This methods sets the union value to the value of the first argument, which has as type the element type. If the switch-case has more than one case label, the modifier method allows to specify with a second argument which discriminator value to use. If an invalid value (discriminator value doesn't belong to this switch-case) is passed here, a `BAD_PARAM` exception with minor code 34 should be throwed.
- A `private` field with the name m_ followed by the mapped element name.
  The type of the field is the mapped element type.

If no default case is specified one additional method is generated:

- A `public` modifier method with the name `SetDefault`.
  It takes one argument, which allows to specify, which discriminator value to use. If an invalid value (discriminator value belongs to a switch-case) is passed here, a `BAD_PARAM` exception with minor code 34 should be throwed.

Note: For simplicity the above method is also generated, if all possible discriminator values are enumerated in the union.

To support serialisation and deserialisation, the union defines two additional static private methods:

- `FieldInfo GetFieldForDiscriminator(discriminator-type val)`. This methods returns the `FieldInfo` for the field corresponsing to the discriminator value or null, if not in range.

3.7.1 Example

```
// IDL
module a {
module b {
  union ExampleUnion switch(long) {
       case 0: short val0;
       case 1:
       case 2: long val1;
       default: boolean val2;
  };

};
};

// C#
namespace a.b {

  [IdlUnion]
  public struct ExampleUnion : IIdlEntity {

       private System.Int32 m_discriminator;

       private System.Int16 m_val0;
       private System.Int32 m_val1;
       private System.Boolean m_val2;

       public System.Int32 Discriminator {
            get {
                 return m_discriminator;
            }
       }

       public System.Int16 Getval0() {
            // checks;
            return m_val0;
       }

       public void Setval0(System.Int16 val) {
            m_value = val;
            m_discriminator = 0;
       }

       public System.Int32 Getval1() {
            // checks;
            return m_val1;
```

```
        }

        public void Setval1(System.Int32 val,
                            System.Int32 discriminator) {
            // check discriminator; set m_value and
            // m_discriminator
        }

        public System.Boolen Getval2() {
            // checks;
            return m_val2;
        }

        public void Setval2(System.Boolean val,
                            System.Int32 discriminator) {
            // check discriminator; set m_value and
            // m_discriminator

        }

        private static FieldInfo GetFieldForDiscriminator(
            System.Int32 discriminator) {
            switch(discriminator) {
                case 0: return s_type.GetField(“m_val0”, ..);
                case 1,2: return s_type.GetField(“m_val1”, ..);
                default: return s_type.GetField(“m_val2”, ..);
            }
        }

        private static bool
        DiscriminatorValueOutsideCoveredRange(
            System.Int32 discrimintaor) {
            return false;
        }

    }
}
```

3.8 Sequence

OMG IDL sequences are mapped to CLS one-dimensional arrays. The CLS array element type is the mapped IDL sequence element type. The CLS type is anonymous (like every CLS array) and is inlined where the OMG IDL typename is used.

The mapped CLS array has the attribute `IdlSequenceAttribute`.

```
    [AttributeUsage(AttributeTargets.Parameter |
AttributeTargets.ReturnValue | AttributeTargets.Field |
AttributeTargets.Property, AllowMultiple = true)]
    public class IdlSequenceAttribute : Attribute,
IDLAttribute {
        /// <summary>
        /// Constructor for unbounded sequences
        /// </summary>
        public IdlSequenceAttribute();

        /// <summary>
        /// constructor for bounded sequences
        /// </summary>
        /// <param name="bound">max nr of elements</param>
        public IdlSequenceAttribute(long bound);

        public long Bound {
            get;
        }

        /// <summary>
        /// is the sequence bounded or not
        /// </summary>
        /// <returns>bounded or not</returns>
        public bool IsBounded();
    }
```

For IDL bounded sequences, the `IdlSequence` attributes stores the bound defined in the Idl. The bound is accessible through the bound property. At runtime, the bound is checked to ensure type consistency.
For unbounded sequences, `IsBounded` returns `false` and `Bound` returns `0`.

Note: All attributes applied to the sequence element type are applied to the member declaring the mapped CLS array type.

3.9 Array

OMG IDL arrays are not mapped the CLS (will be addressed in a future version).

3.10 Interface

An OMG IDL interface (concrete and abstract) is mapped to a CLS interface extending `IDLEntity`. The name of the CLS interface is the mapped IDL interface name as described in section 3.1.

The mapped interfaces has the attribute `InterfaceTypeAttribute` to distinguish abstract from concrete interfaces during the serialization.

```
  public enum IDLTypeInterface {
      ConcreteInterface,
      AbstractInterface,
      AbstractValueType
  }

  [AttributeUsage(AttributeTargets.Interface)]
  public class InterfaceTypeAttribute : Attribute,
IDLAttribute {
      public InterfaceTypeAttribute(IDLTypeInterface
idlType);
      public IDLTypeInterface IDLType {get;}
  }
```

3.10.1 Inheritance Hierarchy Mapping

The inheritance relationship of the OMG IDL class is preserved; except for `CORBA::Object` and `CORBA::AbstractBase`, all inherited interfaces are mapped to the CLS and included in the mapped interface inheritance relationship.

3.10.2 IDL-Attributes

OMG IDL attributes are mapped to CLS public properties with get and set accessors; readonly attributes have only the get accessor. The type of the attribute is mapped to the CLS, and its mapped name is used in the property's declaration.

3.10.3 Methods

OMG IDL methods are mapped to methods in CLS. The method name is mapped according to Section 3.1; the return type is mapped to a CLS type.

All method's parameters are mapped in the same order as in the OMG IDL declaration.

3.10.3.1 Method Parameter passing modes

Passing modes are mapped as follows:

IDL parameter mode	*CLS parameter mode*
`in`	
`out`	`out`
`inout`	`ref`

3.10.3.2 Raise Clause

Raise clauses are ignored, because the CLS' methods have no support for them.

The exceptions in the raise clause are mapped to the CLS according to Section 3.11.8.

3.10.4 #pragma ID

If the OMG IDL type has an associated pragma ID, the CLS interface is marked with the attribute `RepositoryIDAttribute` parametrized with the actual id.

```
  [AttributeUsage(AttributeTargets.Class |
AttributeTargets.Interface | AttributeTargets.Struct)]
  public class RepositoryIDAttribute : Attribute, IDLAttribute
{
      public RepositoryIDAttribute(string id);
      public string Id {get;}
  }
```

3.10.5 Example

```
// IDL
module a {
module b {
  abstract interface BaseInterface {
      long method1(long arg1, short arg2);
      void method2();
  };
  #pragma ID BaseInterface "IDL:a/b/BaseInterface:1.0"

  interface ExtInterface : BaseInterface {
      Object method3() ;
      void method4(long arg1) ;
  };
  #pragma ID ExtInterface "IDL:a/b/ExtInterface:1.0"
};
};

// C#
namespace a.b {
  [RepositoryIDAttribute(“IDL:a/b/BaseInterface:1.0”)]
  [InterfaceTypeAttribute(IDLType.AbstractInterface)]
  public interface BaseInterface : IDLEntity {
      int method1(int arg1, short arg2);
      void method2();
  }
  [RepositoryIDAttribute(“IDL:a/b/ExtInterface:1.0”)]
  [InterfaceTypeAttribute(IDLType.ConcreteInterface)]
  public interface ExtInterface : BaseInterface, IDLEntity {
      MarshalByRefObject method3();
      void method4(int arg1);
  }
}
```

3.10.6 CORBA::Object

The OMG IDL type `CORBA::Object` is mapped to the CLS type `MarshalByRefObject`.

`CORBA::Object` is the interface implemented by all remotely accessible objects, all IDL-interfaces implicitely inherit from it.

3.10.7 CORBA::AbstractBase

The OMG IDL type `CORBA::AbstractBase` is mapped to the CLS type `System.Object`.

All abstract interfaces inherit from `Corba::AbstractBase`.

To tell AbstractBase apart from Object, CLS fields and parameters are marked with the attibute `ObjectIdlTypeAttribute` with value `IDLType.AbstractBase`.

```
  public enum IDLTypeObject {
      Any,
        AbstractBase,
        ValueBase
    }


    [AttributeUsage(AttributeTargets.Parameter |
AttributeTargets.ReturnValue | AttributeTargets.Field |
AttributeTargets.Property)]
    public class ObjectIdlTypeAttribute : Attribute,
IdlAttribute {
        public ObjectIdlTypeAttribute(IdlTypeObject idlType);
        public IDLTypeObject IdlLType {get;}
    }
```

Hint: Whenever this attribute is present, a runtime-check is performed to ensure that the actual value is a valid `Corba::AbstractBase`.

3.11 Value Types

IDL `valuetypes` instances are passed by value. The mapping for concrete valuetypes (Section 3.11.5) differs from the mapping for abstract valuetypes (Section 3.11.6).

3.11.1 #pragma ID

The rules defined in Section 3.10.4 are used.

3.11.2 Inheritance

All inherited valuetypes and supported interfaces are mapped to the CLS; the inheritance relationships are maintained.

3.11.3 Methods

OMG IDL operations are mapped to CLS abstract methods. The operation name is mapped according to Section 3.1; the return type is mapped to the CLS; all parameters are mapped in the same order as in the OMG IDL. The name of the mapped parameters is not relevant.

3.11.3.1 Parameter passing modes

Parameter passing modes are mapped according to Section 3.10.3.1.

Note: Operations for value types are local operations. Therefore the parameters are not marshalled / unmarshalled.

3.11.4 IDL-Attributes

Mapped according to Section 3.10.2.

3.11.5 Concrete Value Types

An IDL concrete value type is mapped to a CLS abstract class implementing the marker interface `IDLEntity` and having the `Serializable` attribute.

Inheritance is mapped according to Section 3.11.2, method are mapped according to Section 3.11.3, attributes are mapped according to Section 3.11.4.

3.11.5.1 State members

State members are mapped to corresponding CLS fields. Public state members are mapped to CLS public fields and private state members are mapped to CLS protected members.

3.11.5.2 Truncation

Truncatable valuetypes are not mapped to the CLS; truncatable types will be supported in a future version of this mapping.

3.11.5.3 Type Implementation

Hint: An implementation for the value type must be provided. As a convention, the marshaller will search for a class whose name is defined in the `ImplClassAttribute` attached to the mapped class; by default, the implementation name is the mapped type name suffixed by "`Impl`". If no such implementation class can be found, the marshaller must throw a `No_Implement` exception with minor-code = 1.

The implementation class must inherit from the mapped class, be serializable, implement the abstract methods and properties defined in the `valuetype`, and provide a parameterless constructor.

```
[AttributeUsage(AttributeTargets.Class)]
public class ImplClassAttribute : Attribute, IDLAttribute {
    public ImplClassAttribute(string implClass);
    public string ImplClass{get;}
}
```

By default, each mapped class carries the attribute `ImplClassAttribute` whose value is the class name suffixed by the string "`Impl`"

3.11.6 Abstract Value Types

Abstract value types can't have state.

OMG IDL abstract value types are mapped to CLS interfaces extending `IDLEntity`.

Inheritance, methods, and attributes are mapped like in a concrete value type (Section 3.11.5). No implementation class is needed.

To distinguish abstract values from abstract and concrete interfaces, the mapped type carries the CLS attribute `InterfaceTypeAttribute` with value `IDLTypeInterface.AbstractValueType`

3.11.6.1 Example

```
// IDL
module a {
  abstract valuetype AbstractValueType {
      boolean check(in long val);
  };
  #pragma ID AbstractValueType "IDL:a/AbstractValueType:1.0"

  valuetype ValueTypeNr1 : AbstractValueType {
      public long x;
      private long z;
      void inc();
      void test(in long x, in long y);
  };
  #pragma ID ValueTypeNr1 "IDL:a/ValueTypeNr1:1.0"
};
```

```
// C#
namespace a {
  [RepositoryIDAttribute("IDL:a/AbstractValueType:1.0")]
  [InterfaceTypeAttribute(IDLType.AbstractValueType)]
  public interface AbstractValueType : IDLEntity {
      bool check(int val);
  }

  [ImplClassAttribute("a.ValueTypeNr1Impl")]
  [RepositoryIDAttribute("IDL:a/ValueTypeNr1:1.0")]
  [Serializable]
  public abstract class ValueTypeNr1 : AbstractValueType,
IDLEntity {
      public int x;
      protected int z;
      public abstract bool check(int val);
      public abstract void inc();
      public abstract void test(int x, int y);
  }
}

// the user must provide the following class:
namespace a {
  [Serializable]
  public class ValueTypeNr1Impl : ValueTypeNr1 {
      public override bool check(int val) {
          // do it …
      }
      public override void inc() {
          // do it …
      }
      public override void test(int x, int y) {
          // do it …
      }
  }
}
```

3.11.7 Boxed Value Types

Boxed value types are a shorthand notation for valuetypes containing only one member. Boxed value types cannot be extended.

There are two mapping for boxed value types.

The first mapping is the generic one: it maps them to CLS classes according to the rules in Section 3.11.5. The boxed valuetype name is mapped to the a CLS name. The mapped class must implement `Ch.Elca.Iiop.Idl.BoxedValueBase` and `IIdlEntity` and be serializable.

The second mapping is optimized for fields, parameters, properties and return types: it directly uses the type of the boxed type's field. Additionaly, the member must carry the attribute `BoxedValueAttribute`.

```
  [AttributeUsage(AttributeTargets.Parameter |
AttributeTargets.ReturnValue | AttributeTargets.Field |
AttributeTargets.Property)]
  public class BoxedValueAttribute : Attribute, IDLAttribute {
      public BoxedValueAttribute(string repositoryID);
      public string RepositoryID {get;}
  }

  public abstract class BoxedValueBase {
      public BoxedValueBase() {…}
      public object unbox() {…}
      public abstract object GetValue();
  }
```

3.11.7.1 Example

```
// IDL
#pragma ID seq1_long “IDL:org/omg/BoxedArray/seq1_long:1.0”
valuetype seq1_long sequence<long>;
// C#
// If the above type
// occurs in method signature, field declarations it’s
// replaced by int[].

// type representing boxed value type
namespace org.omg.boxedRMI {

  [RepositoryIDAttribute("RMI:[I:0000000000000000")]
  [Serializable]
  public class seq1_long : Ch.Elca.Iiop.Idl.BoxedValueBase,
  IIdlEntity {

      [IDLSequence]
      private int[] m_val;

      /// <summary> constructor used for deserialisation
      ///</summary>
      public seq1_long() {
      }

      public seq1_long(int[] arg) {
          // assign m_val from int[]
```

```
                // null is not allowed
            }

            // implementation details
            // …
        }
    }
```

Note: The above mapping is also chosen because the CLS to IDL mapping does map arrays to a boxed valuetype. Therefore with this mapping, the IDL to CLS mapping reproduces a CLS array for this case.

3.11.8 CORBA::ValueBase

OMG IDL type `CORBA::ValueBase` is the abstract base class for all value types; all valuetypes implicitely inherit from it.

`CORBA::ValueBase` is mapped to `System.Object`. All fields and parameters having this type additionally carry the attribute `ObjectIdlTypeAttribute` (Section 3.10.7) with value `IDLType.ValueBase`.

Hint: Whenever this attribute is present, a runtime-check is performed to ensure that the actual value is valid for `Corba::ValueBase.`

3.12 Exceptions

3.12.1 Mapping of CORBA Exceptions

All CORBA exceptions inherit from the abstract base class `AbstractCORBASystemException`: The following table shows how to map the exception name to the CLS.

```
namespace org.omg.CORBA {

public enum CompletionStatus { Completed_YES, Completed_NO,
      Completed_MayBe }

  [Serializable]
  public abstract class AbstractCORBASystemException :
  Exception {

      private int m_minor;
      public int Minor {
          get { return m_minor; }
      }

      private CompletionStatus m_status;
      public CompletionStatus Status {
          get { return m_status; }
      }

      protected AbstractCORBASystemException(string reason,
      int minorCode, CompletionStatus status) : base(reason){
          m_minorCode = minorCode;
          m_status = status;
      }
  }
}
```

IDL-Exception	*CLS 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::INITALIZE	org.omg.CORBA.INITALIZE
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
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::INV_POLICY	org.omg. CORBA.INV_POLICY

IDL-Exception	*CLS class name*
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

3.12.2 Mapping of user defined Exceptions

OMG IDL user exceptions are mapped to a CLS class. The class must extend `org.omg.CORBA.AbstractUserException`. The exception members are mapped to public properties of this subclass.

```
namespace org.omg.CORBA {

  [Serializable]
  public abstract class AbstractUserException : Exception {

      private string m_reason;
      public string Reason {
          get { return m_reason; }
      }

      public AbstractUserException() { }

      protected AbstractUserException(string reason) :
      base(reason) {
      }
  }
}
```

3.13 Any

OMG IDL any is mapped to CLS `System.Object`.

The serialization of any types must carry a typecode.

Parameters and fields with type any must carry CLS attribute `ObjectIdlTypeAttribute` with value `IDLType.`Any is assigned to the parameters and fields with IDL-type any (see Section 3.10.7 for `ObjectIdlTypeAttribute`'s definition). This attribute is the information needed by the serializer to handle any correctly.

3.14 Nested Types

OMG IDL nested types are mapped to CLS non-nested types according to the type mapping rules. The nested type is declared in a new namespace inside the container type's namespace, whose name is the mapped type name suffixed with "`_package`".

3.15 TypeDef

OMG IDL `typedef` are not mapped; wherever an aliased type names is used in the IDL, it is substitued with the canonical type name in the CLS.

3.16 Constants

For every IDL constant, a public sealed class with the name of the constant is created.

This class is declared in the following namespace:

- if the constant is defined inside a type (e.g. an interface), the namespace is created in the same way as for nested types, see section 3.14.
- if the constant is defined outside a type, the module namespace is used.

The class contains the following items:

- a public static readonly field with the name `ConstVal` and type of the constant. It stores the value of the constant.
- a static initalizer for setting the constant value.
- a private default constructor to prevent creating instances of the class.

3.16.1.1 Example

```
// IDL
module test {
    interface X {
        const long MyConstant = 11;
    };
};

// C#
namespace test.X_package {
    public sealed class MyConstant {
        public static readonly ConstVal;

        static MyConstant() {
            ConstVal = 11;
        }

        private MyConstant() {
        }
    }
}

// interface mapping …
```

3.17 Used Attributes

The following table summarizes all CLS-attributes used in the mapped types.

Attribute	*Purpose*	*Usage*	*Definition in*
`RepositoryIDAttribute`	bind an IDL repository attribute to a CLS type	Class, struct, interface	3.10.4
`ImplClassAttribute`	name of the class implementing the valuetype	Class	3.11.5.3
`IdlStructAttribute`	original type is an IDL `struct`	Struct	3.6
`IdlEnumAttribute`	original type is an IDL `enum`	Enum	3.5
`IdlUnionAttribute`	original type is an IDL `union`	Union	3.7
`BoxedValueAttribute`	indicate a mapping from the boxed value type with the specified repository id	Parameter, Field, ReturnType, Property	3.11.7
`IdlSequenceAttribute`	original type is an IDL `sequence`	Parameter, Field, ReturnType, Property	3.8
`InterfaceTypeAttribute`	indicates whether the original IDL type is a concrete interface, an abstract interface, or an abstract value type	Interface	3.10
`ObjectIdlTypeAttribute`	indicates whether the original IDL type is any, AbstractBase, or ValueBase	Parameter, Field, ReturnType, Property	3.10.7

Attribute	*Purpose*	*Usage*	*Definition in*
`WideCharAttribute`	specifies if wide characters are allowed.	Parameter, Field, ReturnType, Property	3.3.2
`StringValueAttribute`	specifies whether the original IDL type is string or wstring	Parameter, Field, ReturnType, Property	3.3.2

4 Marshalling / Unmarshalling

This section describes how to marshal and unmarshal the data passed during a distributed method call.

The marshalling process generates for each instance a serialized form suitable for transport. The unmarshalling process reconstructs the instances based on the serialized form.

Usually, marshalling and unmarshalling depends on the static type, i.e. the type present in the declaration of the member to be converted. In some cases where the type mapping is ambiguous, the marshaller must decide at run-time which conversion to use, depending on the dynamic type or on the custom attributes on the type. Some runtime tests may also be required to intercept illegal values, whenever the source and destination type have different domains.

4.1 Marshalling of Parameters During Methods Calls

Method parameters are serialized according to the formal parameter type[1]:

- compatible with `MarshalByRefObject`
  The actual parameter is passed by reference.
- conforming value type
  The actual parameter is passed by value
- non-conforming value type
  If the actual parameter type is a conforming value type, the parameter is passed by value; otherwise an exception is thrown.
- `System.Object`
  The serialization of the formal parameter depends on the `ObjectIdlTypeAttribute`, which specifies the original OMG IDL type.

ObjectIdlTypeAttribute	*Serialized as*
`IDLType.Any` or omitted	OMG IDL any
`IDLType.AbstractBase`	Abstract interface
`IDLType.ValueBase`	By value

- Interface
  The serialization of the formal parameter depends on the InterfaceTypeAttribute

InterfaceTypeAttribute	*Serialized as*
`IDLType.ConcreteInterface`	By reference
`IDLType.AbstractValue`	By value
`IDLType.AbstractInterface` or omitted	Depending on the object's dynamic type:

[1] The type in the method signature

	• MarshalByRef: by reference • MarshalByValue: by value • Serializable: by value • Otherwise: error condition

- enum
  Parameter is passed by value.
- struct with `IDLStruct` attribute
  Parameter is passed by value.
- array
  Parameter is passed by value.

4.2 Marshalling / Unmarshalling of Value-Types

The value type is serialised/deserialised as described in the CORBA spec. The fields of the value type are serialised/deserialised in the order of their declaration. The serialization starts with the highest possible base type and continues up to the most derived type.

The marshalling of the fields is done according to their formal type:

- MarshalByRef
  the field is serialized by reference.
- conforming CLS value type
  the field is serialized by value.
- non-conforming CLS value type
  if the value is a conforming value type, the field is serialized by value is passed; otherwise an exception is thrown
- `System.Object`
  the serialization depends on the ObjectIdlTypeAttribute associated to the field:

ObjectIdlTypeAttribte	*Serialized as*
IDLType.Any or none	OMG IDL any
IDLType.AbstractBase	Abstract interface
IDLType.ValueBase	value

- Interface
  The serialization of the formal parameter depends on the InterfaceTypeAttribute

InterfaceTypeAttribute	*Serialized as*
`IDLType.ConcreteInterface`	By reference
`IDLType.AbstractValue`	By value
`IDLType.AbstractInterface` or omitted	Depending on the object’s dynamic type:

	• MarshalByRef: by reference • MarshalByValue: by value • Serializable: by value • Otherwise: error condition

- enum
  Parameter is passed by value.
- struct with `IDLStruct` attribute
  Parameter is passed by value.
- array
  Parameter is passed by value.