Date: March 2020

OBJECT MANAGEMENT GROUP

IDL4 to C# Language Mapping, Version 1.0

FTF Beta 1

OMG Document Number	ptc/2020-03-02
Normative Reference:	https://www.omg.org/spec/IDL4-CSHARP

This OMG document replaces the submission document (mars/19-11-01, Alpha). It is an OMG Adopted Beta Specification and is currently in the finalization phase. Comments on the content of this document are welcome, and should be directed to issues@omg.org by May 01, 2020.

You may view the pending issues for this specification from the OMG revision issues web page https://issues.omg.org/issues/lists.

The FTF Recommendation and Report for this specification will be published in December 2020. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.

OMG's Issue Reporting Procedure

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

Table of Contents

Table of Tables

Preface

OMG

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

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

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

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from the OMG website at:

https://www.omg.org/spec

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

OMG Headquarters
109 Highland Avenue
Needham, MA 02494
USA

Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: pubs@omg.org

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

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification by completing the Issue Reporting Form listed on the main web page https://www.omg.org, under Documents, Report a Bug/Issue.

0 Response Details

0.1 OMG Response Details

This specification is submitted in response to the "Interface Definiton Language v4 (IDL4) to C# Language Mapping" RFP issued with OMG document number mars/2018-12-02.

0.2 Copyright Waiver

0.3 Contacts

- Real-Time Innovations, Inc. Fernando Garcia-Aranda fgarcia AT rti.com
- Twin Oaks Computing, Inc. Clark Tucker ctucker AT twinoakscomputing.com
- ADLINK Technology Ltd. Erik Hendriks erik.hendriks AT adlinktech.com
- Objective Interface Systems, Inc. Chuck Abbott chuck.abbott AT ois.com
- Micro Focus International Plc. Matteo Vescovi Matteo.Vescovi AT microfocus.com

0.4 Problem Statement

Version 4 of the Interface Definition Language (IDL) specification extended the traditional IDL syntax and defined a comprehensive set of building blocks to categorize it. This evolution requires a new set of IDL to language mappings to leverage the new IDL constructs and to adapt to the new building block-based document structure. Moreover, it requires mappings to popular programming languages, such as C#, for which there was no existing language mapping.

This proposal defines complete mapping of the Interface Definition Language v4 (IDL4) to the C# programming language.

0.5 Overview of this Specification

C# is one of the most prominent general-purpose programming languages. Indeed, in the last decades multiple vendors have implemented custom mappings of IDL to C# in order to extend OMG technologies, such as DDS and CORBA, to new industries and use cases. However, these solutions fail to provide a vendor-independent interoperable solution that many users have requested.

This specification defines the mapping of OMG Interface Definition Language v4 to the C# programming language [ECMA-334]. The language mapping covers all of the IDL constructs in the current Interface Definition Language specification [OMG-IDL4]. The language mapping makes use of C# language features as appropriate and natural.

0.6 Statement of Proof of Concept

Submitters of this proposal have developed proof of concept implementations of the IDL to C# language mappings defined in this specification. The resulting mappings have also been evaluated by experienced software engineers who specialize in C#.

0.7 Mapping to RFP Requirements

The specification resolves the mandatory requirements listed in Table 0.1.

Table 0.1: Mandatory RFP Requirements

Requirement	Description	References	Remarks
6.5.1	General Requirements		
6.5.1.1	Proposals shall provide a complete mapping of IDL to the C# language.	7	**Satisfied** Chapter 7 provides a complete mapping of the C# programming language that reuses the building block structure defined in [OMG-IDL4].
6.5.1.2	Proposals shall identify the minimum C# language version required by the mapping.	7.1.3	**Satisfied** Clause 7.1.3 provides both the minimum version of the C# language and the minimum version of the .NET standard required by the mapping, and presents the specific features that demand such versions.
6.5.2	Document Structure Requirements		
6.5.2.1	Proposals shall include a traceability matrix cross referencing each IDL building block to the clauses or paragraphs that map it.	Annex B	**Satisfied** Table B.1 provides such traceability with references to each IDL building block and the corresponding clause that map it.
6.5.2.2	Proposals shall follow an outline that parallels the IDL specification. The outline shall minimally include the following clauses (excluding the standard boiler-plate sections and subject to numbering changes)	7	**Satisfied** The proposal includes all the clauses dictated by the requirement, parallel to the IDL specification, although it alters slightly the ordered to facilitate the implementation of the mappings.

This specification resolves the non-mandatory requirements listed in Table 0.2.

Table 0.2: Non-Mandatory RFP Requirements

Requirement	Description	References	Remarks
6.6.1	Language Version		

Requirement	Description	References	Remarks
6.6.1.1	The language mapping may utilize C# language features from any C# version.	7.1.3	**Satisfied** As shown in Table 7.1, this specification uses language features that have been available at least since version 4.0 of the C# programming language. Depending on the building blocks, versions 1.0 and 2.0 of C# are also sufficient.
6.6.2	Annotations		
6.6.2.1	Proposals may define specialized annotations to control the code generation.	7.2.3	**Satisfied** This specification defines in Clause 7.2.3 a new IDL annotation named **`@csharp_mapping`**, which provides mechanisms to control code generation aspects, such as coding conventions.

0.8 Responses to RFP Issues to be Discussed

The RFP issues to be discussed are addressed in Table 0.3.

Table 0.3: RFP Issues to be Discussed

Issue	Description	Discussion
6.7.1	Submissions shall provide a rationale for any IDL construct that is not covered by the proposed language mapping.	This submission provides mappings for all the IDL constructs that have a direct language mapping. For IDL constructs that have no direct language mapping, such those defined by the building blocks Components – Basic, Components – Homes, Components – Ports and Connectors, and Template Modules, we state that the actual language mapping is derived from the transformation of the corresponding intermediate IDL into C# following the rules defined in this specification.
6.7.2	Submissions shall describe how the specified mappings comply with the .NET Common Type System.	The .NET Common Type System defines how "types are declared, used, and managed in the common language runtime" (see https://docs.microsoft.com/en-us/dotnet/standard/base-types/common-type-system). All primitive types, classes, structures, enumerations, and interfaces used and defined in this specification are based on C# and .NET data types that are compliant with the .NET Common Type System.
6.7.3	Submissions shall identify and justify the required minimum C# language specification version.	The minimum version of the C# programming language is justified, based on the features used as part of the mappings defined in this specification in Table 7.1.

Issue	Description	Discussion
		Each of the mappings specified for the different building block aims to provide the most natural mechanism to maintain the semantics of the original IDL constructs in the mapped C# code.
6.7.4	Submissions shall identify and justify the required minimum .NET Standard version.	The minimum version .NET Standard version is justified, based on the features used as part of the mappings defined in this specification, in Table 7.1. Each of the mappings specified for the different building block aims to provide the most natural mechanism to maintain the semantics of the original IDL constructs in the mapped C# code.

1 Scope

This specification defines the mapping of OMG Interface Definition Language v4 to the C# programming language [ECMA-334]. The language mapping covers all of the IDL constructs in the current Interface Definition Language specification [OMG-IDL4]. The language mapping makes use of C# language features as appropriate and natural.

2 Conformance Criteria

Conformance to this specification can be considered from two perspectives:

1. implementations (for example, a tool [compiler] that applies the mapping to generate C# source code from IDL); and
2. users (for example, application source code that interacts with the C# source code generated by a compiler).

Table 2.1: Conformance Points

Implementation	A conformant implementation shall transform IDL input into C# source code output as specified in Chapter 7.
User	Application source code that conforms to this specification makes use of the C# data types and APIs as defined in Chapter 7. Conformant application source code shall make no assumptions about the underlying implementation or utilize any unspecified API or behavior beyond what is specified in the language mapping. Conformant application source code, as a result, will be portable across implementations.

3 Normative References

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

[CORBA-IFC] OMG, Common Object Request Broker Architecture, Part 1: CORBA Interfaces, Version 3.3, https://www.omg.org/spec/CORBA/3.3

[ECMA-334] ECMA, C# Language Specification, 5th Edition, https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf

[OMG-IDL4] OMG, Interface Definition Language, Version 4.3, https://www.omg.org/spec/IDL/4.3

[.NET-GUIDE] Krzysztof Cwalina, Brad Abrams, Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2008

[.NET-STD] .NET Implementers, .NET Standard, https://docs.microsoft.com/en-us/dotnet/standard/net-standard

4 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply.

Building Block

A Building Block is a consistent set of IDL rules that together form a piece of IDL functionality. Building blocks are atomic, meaning that if selected, they must be totally supported.

Building blocks are described in [OMG-IDL4] Chapter 7, IDL Syntax and Semantics.

C#

C# is a general-purpose computer programming language.

Camel Case

A naming convention that represents phrases composed of multiple word using a single word where spaces and punctuation are removed, and every word begins with a capital letter.

In this specification, the term Camel Case refers to the variation of Camel Case commonly-known as Lower Camel Case, where the first letter is not capitalized. For example, the Camel Case representation of “these are my words” would be “theseAreMyWords”.

Language Mapping

An association of elements in one language to elements in another language (from IDL to C#, in this case) that facilitates a transformation from one language to another.

Pascal Case

Also known as Upper Camel Case, is a variation of Camel Case where the first letter is capitalized. For example, the Pascal Case representation of the phrase “these are my words” would be “TheseAreMyWords”.

5 Symbols

The acronyms used in this specification are show in Table 5.1.

Table 5.1: Acronyms

Acronym	Meaning
CCM	Corba Component Model
CLI	Common Language Infrastructure
CLS	Common Language Specification
CORBA	Common Object Request Broker Architecture
CTS	Common Type System
DDS	Data Distribution Service

Acronym	Meaning
IDL	Interface Definition Language

6 Additional Information

6.1 Changes to Adopted OMG Specifications

This specification does not change any adopted OMG specification.

6.2 Acknowledgments

The following companies submitted this specification:

- Real-Time Innovations, Inc.
- Twin Oaks Computing, Inc.
- ADLINK Technology Ltd.
- Objective Interface Systems, Inc.
- Micro Focus International Plc.

The following companies supported this specification:

- Kongsberg Defence & Aerospace
- Object Computing, Inc.

The following individuals have contributed content that was incorporated into this specification:

- Submitting contributors:
  - Chuck Abbott
  - Michel Bagnol
  - Fernando Garcia-Aranda (editor)
  - Erik Hendriks
  - Simon McQueen
  - Adam Mitz
  - José Morato
  - Benito Palacios Sanchez
  - Gerardo Pardo-Castellote
  - Håvard N. Skjevling
  - Ørnulf Staff
  - Clark Tucker
  - Matteo Vescovi

7 IDL to C# Language Mapping

7.1 General

7.1.1 Names

IDL member names and type identifiers shall map to equivalent C# names and type identifiers. This specification defines two naming schemes that determine the name transformation behavior:

- *IDL Naming Scheme* (defined in Clause 7.1.1.1), which preserves the naming conventions of the original IDL names and type identifiers.
- *.NET Framework Design Guidelines Naming Scheme* (defined in Clause 7.1.1.2), which transforms names and type identifiers to follow the naming conventions defined in the .NET Framework Design Guidelines [.NET-GUIDE].

The **`@csharp_mapping`** annotation defined in Clause 8.1 provides a mechanism to select the appropriate naming scheme. Implementations of this specification may also provide custom compiler settings or compiler parameters for such purpose.

Regardless of the naming scheme of choice, if a mapped name or identifier collides with one of the names reserved in Clause 7.1.2, the collision shall be resolved by prepending the `"@"` character to the mapped name.

7.1.1.1 IDL Naming Scheme

IDL member names and type identifiers shall map to C# names and identifiers without case transformation, maintaining the original IDL names.

Table 8.1 (**`apply_naming_convention = IDL_NAMING_CONVENTION`** column) defines the name mapping for every IDL construct according to the naming scheme.

7.1.1.2 .NET Framework Design Guidelines Naming Scheme

IDL member names and type identifiers shall map to C# names and identifiers that follow the coding guidelines defined in the Framework Design Guidelines of [.NET-GUIDE].

Table 8.1 (**`apply_naming_convention = DOTNET_NAMING_CONVENTION`** column) defines the name mapping for every IDL construct according to this naming scheme. Most of the rules defined in Table 8.1 require transforming IDL names into either Pascal Case or Camel Case; in such cases, the transformation shall be performed according to the rules defined in Clauses 7.1.1.2.1 and 7.1.1.2.2, respectively.

7.1.1.2.1 Pascal Case Transformation

When required, an IDL member name or type identifier shall be transformed into Pascal Case according to the following rules:

- The first letter after each underscore shall be capitalized and all underscores shall be removed.
- The first letter of the IDL name shall be capitalized.

For example:

- “pascalcase” maps to “Pascalcase”.
- “PASCALCASE” remains “PASCALCASE”.
- “Pascal_Case” maps to “PascalCase”, “pascal_case” to “PascalCase”, “Pascal_case” to “PascalCase”, “PASCAL_case” to “PASCALCase”, “PASCAL_CASE” to “PASCALCASE”, “_pascalCase” to “PascalCase”, “_PascalCase” to “PascalCase”, and “pascal_case_” to “PascalCase”.

- “pascalCase” maps to “PascalCase”, “PascalCase” remains “PascalCase”, “PASCALcase” remains “PASCALcase”, and “PASCALCase” remains “PASCALCase”.

7.1.1.2.2 Camel Case Transformation

When required, an IDL member name or type identifier shall be transformed into Camel according to the following rules:

- The first letter after each underscore shall be capitalized and all underscores shall be removed.
- The first letter of the IDL name shall be lower case.

For example:

- “camelcase” remains “camelcase”.
- “CAMELCASE” becomes “cAMELCASE”.
- “Camel_Case” maps to “camelCase”, “camel_case” to “camelCase”, “Camel_case” to “camelCase”, “camel_Case” to “camelCase”, “CAMEL_case” to “cAMELCase”, “CAMEL_CASE” to “cAMELCASE”, “_camelCase” to “camelCase”, “_CamelCase” to “camelCase”, and “camel_case_” to “camelCase”.
- “camelCase” remains “camelCase”, “CamelCase” maps to “camelCase”, “CAMELcase” to “cAMELcase”, and “CAMELCase” to “cAMELCase”.

7.1.2 Reserved Names

This specification reserves the use the following names for its own purposes:

- The keywords in the C# language specified in Clause 7.4.4 of [ECMA-334].
- The C# class name **`Constants`**, defined in each C# **`namespace <moduleName>`** resulting from the mapping of an IDL-defined **`module`** named **`<moduleName>`**.

In accordance with 7.1.1, the use of any of these names for a user-defined IDL type or interface (assuming it is also a legal IDL name) shall result in the mapped name preceded by a prepended `"@"` character.

7.1.3 C# Language Version Requirements

The language mappings defined in this specification rely on features of the C# programming languages that are not available in all versions of the C# Language and the .NET standard [.NET-STD]. Table 7.1 identifies such C# language features and provides the minimum version of the C# language and .NET standard that provides them.

Table 7.1: C# Language Version and Features

Feature	C# Minimum Version	.NET Standard Minimum Version
`IList<T>`	2.0	1.0
`IDictionary<TKey, TValue>`	2.0	1.0
`dynamic` type	4.0	N/A
System Exceptions: **`Exception`**, **`ArithmeticException`**, **`ArgumentOutOfRangeException`**, **`InvalidOperationException`**	1.0	1.0
`System.FlagsAttribute`	N/A	1.0

`System.Collections.BitArray`	N/A	1.0

NOTE—For readability purposes, some of the examples included in this specification use expression body definition syntax that requires C# 6. The use of such syntax is limited to non-normative parts of the document and is therefore not required; normative parts are solely ruled by the C# language version requirements listed in Table 7.1.

7.2 Core Data Types

7.2.1 IDL Specification

There is no direct mapping of the IDL Specification itself. The elements contained in the IDL specification are mapped as described in the following sections.

7.2.2 Modules

IDL **`module`**s shall be mapped to C# **`namespace`**s of the same name. All IDL type declarations within the IDL **`module`** shall be mapped to corresponding C# declarations within the generated **`namespace`**.

IDL declarations not enclosed in any **`module`** shall be mapped into the global scope.

For example, the following **`module`** declaration in IDL:

```
// ...
module my_math {
    // ...
};
```

would map to the following C# **`namespace`** declaration according to the *IDL Naming Scheme*:

```
// ...
namespace my_math
{
    // ...
}
```

or to the following C# **`namespace`** declaration when using the *.NET Framework Design Guidelines Naming Scheme*:

```
// ...
namespace MyMath
{
    // ...
}
```

7.2.3 Constants

IDL constants shall be mapped to **`public sealed class`**es of the same name within the equivalent scope and **`namespace`** where they are defined. The mapped **`class`** shall contain a **`public const`** called **`Value`** assigned to the value of the IDL constant.

For example, the IDL **`const`** declarations below:

```
module my_math {
    const double PI = 3.141592;
    const double e = 2.718282;
    const string my_string = "My String Value";
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
namespace my_math
{
    public sealed class PI
```

```
    {
        public const double Value = 3.141592;
    }
    public sealed class e
    {
        public const double Value = 2.718282;
    }
    public sealed class my_string
    {
        public const string Value = "My String Value";
    }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
namespace MyMath
{
    public sealed class PI
    {
        public const double Value = 3.141592;
    }
    public sealed class E
    {
        public const double Value = 2.718282;
    }
    public sealed class MyString
    {
        public const string Value = "My String Value";
    }
}
```

NOTE—The mapping of constants defined above provides a complete solution for mapping IDL constants to the C# programming language. In particular, it enables users to define constants in two different assemblies within the same scope avoiding potential name clashes. To accommodate simpler use cases, where constants will never be defined in separate C# assemblies within the same scope, this specification provides in Clause 7.2.3.1 an alternative mapping that groups all constant declarations within a scope in a single class.

7.2.3.1 Alternative Mapping

Every scope containing a constant declaration shall contain a **`public partial class`**. By default, the mapped **`class`** shall be named **`Constants`**. The class name may be modified using the **`@csharp_mapping`** annotation defined in Clause 8.1.2, preceding the declaration of the IDL module containing the constants or the constant declaration itself:

```
@csharp_mapping(constants_container="<ContainerName>")
```

For every constant IDL constant, the mapped **`public partial class`** shall contain a C# constant declaration of the equivalent type with the same name and value. In accordance with Clause 7.2.2, if the constants are not enclosed in any module, the **`public public partial class`** shall be placed under the global scope.

For example, the IDL **`const`** declarations below:

```
@csharp_mapping(constants_container="Constants")
module my_math {
    const double PI = 3.141592;
    const double e = 2.718282;
    const string my_string = "My String Value";
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
namespace my_math
{
    public partial class Constants
```

```
    {
        public const double PI = 3.141592;
        public const double e = 2.718282;
        public const string my_string = "My String Value";
    }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
namespace MyMath
{
    public partial class Constants
    {
        public const double PI = 3.141592;
        public const double E  = 2.718282;
        public const string MyString = "My String Value";
    }
}
```

7.2.4 Data Types

7.2.4.1 Basic Types

7.2.4.1.1 Integer Types

Integer types shall be mapped as shown in Table 7.2.

Table 7.2: Mapping of Integer Types

IDL Type	C# Type
`int8`	`sbyte`
`uint8`	`byte`
`short` `int16`	`short`
`unsigned short` `uint16`	`ushort`
`long` `int32`	`int`
`unsigned long` `uint32`	`uint`
`long long` `int64`	`long`
`unsigned long long` `uint64`	`ulong`

7.2.4.1.2 Floating-Point Types

IDL floating-point types shall be mapped as shown Table 7.3.

Table 7.3: Floating-Point Types Mapping

IDL Type	C# Type
`float`	`float`
`double`	`double`

IDL Type	C# Type
long double	**decimal**

NOTE—According to [OMG-IDL4], **long double** values represent IEEE double-extended floating-point numbers, which have an exponent of at least 15 bits in length and a signed fraction of at least 64 bits[1]. As a result, whilst the most natural mapping of IDL **long double**s to C# is **decimal**, this mapping could cause marshalling and unmarshalling issues in applications using other language mappings. In particular, they may cause precision lost in the transmission of **long double** values.

7.2.4.1.3 Char Types

The IDL **char** type shall be mapped to the C# type **char**[2].

7.2.4.1.4 Wide Char Types

The IDL **wchar** type shall be mapped to the C# type **char**.

7.2.4.1.5 Boolean Types

The IDL **boolean** type shall be mapped to the C# **bool**, and the IDL constants **TRUE** and **FALSE** shall be mapped to the corresponding C# boolean literals **true** and **false**.

7.2.4.1.6 Octet Type

The IDL type **octet**, which defines an 8-bit quantity, shall be mapped to the C# type **byte**.

7.2.4.2 Template Types

7.2.4.2.1 Sequences

IDL sequences shall be mapped to the C# **System.Collections.Generic.IList<T>** interface, instantiated with the mapped type **T** of the sequence elements[3]. In the mapping, everywhere the sequence type is needed, a **System.Collections.Generic.Ilist<T>** shall be used.

Implementations of **System.Collections.Generic.IList<T>** shall be writable.

Bounds checking on bounded sequences may raise an exception if necessary.

As an example, Table 7.4 shows the mapping for sequences of basic types.

Table 7.4: Mapping of Sequences of Basic Types

IDL Type	C# Type
sequence<boolean>	**System.Collections.Generic.IList<bool>**
sequence<char> **sequence<wchar>**	**System.Collections.Generic.IList<char>**
sequence<int8>	**System.Collections.Generic.IList<sbyte>**

1 See IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985, for a detailed specification.

2 IDL characters are 8-bit quantities representing elements of a character set, while C# characters are 16-bit unsigned quantities representing Unicode characters in UTF-16 encoding.

3 This allows implementers to use different **List** implementations (e.g., **System.Collections.Generic.List<T>**) while remaining interface compliant.

sequence<uint8> **sequence<octet>**	**System.Collections.Generic.IList<byte>**
sequence<int16> **sequence<short>**	**System.Collections.Generic.IList<short>**
sequence<uint16> **sequence<unsigned short>**	**System.Collections.Generic.IList<ushort>**
sequence<int32> **sequence<long>**	**System.Collections.Generic.IList<int>**
sequence<uint32> **sequence<unsigned long>**	**System.Collections.Generic.IList<uint>**
sequence<int64> **sequence<long long>**	**System.Collections.Generic.IList<long>**
sequence<uint64> **sequence<unsigned long long>**	**System.Collections.Generic.IList<ulong>**
sequence<float>	**System.Collections.Generic.IList<float>**
sequence<double>	**System.Collections.Generic.IList<double>**
sequence<long double>	**System.Collections.Generic.IList<decimal>**

7.2.4.2.2 Strings

IDL **string**s, both bounded and unbounded variants, shall be mapped to C# **string**s. The resulting strings shall be encoded in UTF-16 format.

7.2.4.2.3 Wstrings

IDL **wstring**s, both bounded and unbounded variants, shall be mapped to C# **string**s. The resulting strings shall be encoded in UTF-16 format.

7.2.4.2.4 Fixed Type

The IDL **fixed** type shall be mapped to the C# **decimal** type.

Range checking shall raise a **System.ArithmeticException** exception, or a derived exception, if necessary.

7.2.4.3 Constructed Types

7.2.4.3.1 Structures

An IDL **struct** shall be mapped to a C# **public class** with the same name. The class shall provide the following:

- A public property of the equivalent type for each member of the structure, including both a getter and a setter.
- A public default constructor that takes no parameters (i.e., the default constructor).
- A public copy constructor that takes as a parameter an object of the mapped class.
- A public constructor that accepts parameters for each member (i.e., the all values constructor).

The default constructor shall initialize member fields as follows:

- All primitive members shall be left as initialized by the C# default initialization.
- All **string** members in the **struct** shall be initialized to **string.Empty**.
- All array members shall be initialized to an array of the declared size whose elements are initialized with their default constructor.
- All sequence members shall be initialized to zero-length sequences of the corresponding type.

- All other members shall be initialized to an object created with their respective default constructor.

For example, the IDL **`struct`** declaration below:

```
struct MyStruct {
    long a_long;
    short a_short;
    long a_long_array[10];
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
public class MyStruct
{
    public MyStruct() {...}
    public MyStruct(MyStruct object) {...}
    public MyStruct(int a_long, short a_short, int[] a_long_array) {...}

    public int a_long { get; set; }
    public short a_short { get; set; }
    public int[] a_long_array { get; set; }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
public class MyStruct
{
    public MyStruct() {...}
    public MyStruct(MyStruct object) {...}
    public MyStruct(int aLong, short aShort, int[] aLongArray) {...}

    public int ALong { get; set; }
    public short AShort { get; set; }
    public int[] ALongArray { get; set; }
}
```

7.2.4.3.2 Unions

An IDL **`union`** shall be mapped to a C# **`public class`** with the same name. The class shall provide the following:

- A public default constructor.
- A public read-only property named **`Discriminator`**.
- A public property with getters and setters for each member.
- A public property with getters and setters for the member corresponding to the default label, if present.

Name clashes with the **`Discriminator`** property name, the mapped union type name or the name of any of the union fields shall be resolved in accordance with the conflict resolution rules specified in Clause 7.1.2, prepending a "`@`" character to the conflicting mapped name.

Property getters shall raise a **`System.InvalidOperationException`** if the expected member has not been set.

If there is more than one case label corresponding to a member, the setter of the property representing such member shall set **`Discriminator`** to the first possible case label. If the member corresponds to the default case label, then **`Discriminator`** shall be set to the first available default value starting from the zero-index of the discriminant type. For all such members, the union shall provide a modifier method **`Set<MemberName>(<MemberType> value, <DiscriminatorType> discriminator)`** to set the corresponding property value and the discriminator value of choice. The modifier method shall throw a **`System.ArgumentException`** exception when a value is passed for the discriminator that is not among the case labels for the member.

For example, the IDL **`union`** declaration below:

```
union AUnion switch (octet) {
```

```
    case 1:
        long  a_long;
    case 2:
    case 3:
        short a_short;
    default:
        octet a_byte_default;
};
```

would map to the following C# to the *IDL Naming Scheme*:

```
public class AUnion
{
    // ...

    public byte Discriminator { get; private set; }

    public int a_long
    {
        get
        {
            if (Discriminator != 1)
                throw new System.InvalidOperationException();
            // ...
        }
        set
        {
            Discriminator = 1;
            // ...
        }
    }

    public short a_short
    {
        get
        {
            if (Discriminator != 2 && Discriminator != 3)
                throw new System.InvalidOperationException();
            // ...
        }
        set
        {
             Discriminator = 2;
             // ...
        }
    }

    public void Seta_short(short value, byte discriminator)
    {
         if (discriminator != 2 && discriminator != 3)
                throw new System.InvalidOperationException();

         Discriminator = discriminator;
         // ...
    }

    public byte a_byte_default
    {
        get
        {
            if (Discriminator == 1 || Discriminator == 2 || Discriminator == 3)
                throw new System.InvalidOperationException();
            // ...
```

```
        }
        set
        {
            Discriminator = 0;
            // ...
        }
    }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
public class AUnion
{
    // ...

    public byte Discriminator { get; private set; }

    public int Along
    {
        get
        {
            if (Discriminator != 1)
                throw new System.InvalidOperationException();
            // ...
        }
        set
        {
            Discriminator = 1;
            // ...
        }
    }

    public short AShort
    {
        get
        {
            if (Discriminator != 2 && Discriminator != 3)
                throw new System.InvalidOperationException();
            // ...
        }
        set
        {
             Discriminator = 2;
             // ...
        }
    }

    public void SetAShort(short value, byte discriminator)
    {
         if (discriminator != 2 && discriminator != 3)
                throw new System.InvalidOperationException();

         Discriminator = discriminator;
         // ...
    }

    public byte AByteDefault
    {
        get
        {
            if (Discriminator == 1 || Discriminator == 2 || Discriminator == 3)
                throw new System.InvalidOperationException();
            // ...
```

```
        }
        set
        {
            Discriminator = 0;
            // ...
        }
    }
}
```

7.2.4.3.3 Enumerations

An IDL **enum** shall be mapped to a C# **public enum** with the same name as the IDL **enum** type.

If the IDL enumeration declaration is preceded by a **@bit_bound** annotation; the corresponding C# **enum** type shall be **sbyte** for bit bound values between 1 and 8; **short**, for bit bound values between 9 and 16; **int**, for bit bound values between 17 and 32; and **long**, for bit bound values between 33 and 64.

For example, the IDL **enum** declaration below:

```
enum AnEnum {
    @value(1) one,
    @value(2) two
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
public enum AnEnum
{
    one = 1,
    two = 2
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
public enum AnEnum
{
    One = 1,
    Two = 2
}
```

Also, the IDL **enum** declaration below:

```
@bit_bound(6)
enum ABoundEnum {
    @value(1) one,
    @value(2) two
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
public enum ABoundEnum : sbyte
{
    one = 1,
    two = 2
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
public enum ABoundEnum : sbyte
{
    One = 1,
    Two = 2
}
```

7.2.4.3.4 Constructed Recursive Types

Constructed recursive types are supported by mapping the involved types directly to C# as described elsewhere in Clause 7.

7.2.4.4 Arrays

An IDL array shall be mapped to a C# array of the mapped element type[4] or to a C# **class** offering an interface compatible with that of a C# native array of the mapped element type. In the mapping, everywhere the array type is needed, an array or an equivalent **class** of the mapped element type shall be used. The bounds for the array shall be checked by the setter of the corresponding property and a **System.ArgumentOutOfRangeException** shall be raised if a bounds violation occurs.

For example, the IDL declaration below:

```
const long foo_array_length = 200;

struct MyType {
    long long_array[100];
    Foo foo_array[foo_array_length];
};
```

could map to the following C# according to the *IDL Naming Scheme*:

```
public sealed class foo_array_length
{
    public const int Value = 200;
}

public class MyType
{
    // ...

    public MyType() {...}
    public MyType(MyType object) {...}
    public MyType(int[] long_array, Foo[] foo_array) {...}

    public int[] long_array
    {
        get
        {
            // ...
        }
        set
        {
            if (value.Length != 100)
                throw new ArgumentOutOfRangeException(nameof(long_array));
            // ...
        }
    }
    public Foo[] foo_array
    {
        get
        {
            // ...
        }
        set
        {
            if (value.Length != foo_array_length.Value)
```

4 The length of the array can be made available in the mapped C# source code by bounding the IDL array with an IDL constant, which will be mapped as per the rules for constants. For example, see **foo_array_length** in the example above.

```
                throw new ArgumentOutOfRangeException(nameof(foo_array));
            // ...
        }
    }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
public sealed class FooArrayLength
{
    public const int Value = 200;
}

public class MyType
{
    // ...

    public MyType() {...}
    public MyType(MyType object) {...}
    public MyType(int[] longArray, Foo[] fooArray) {...}

    public int[] LongArray
    {
        get
        {
            // ...
        }
        set
        {
            if (value.Length != 100)
                throw new ArgumentOutOfRangeException(nameof(LongArray));
            // ...
        }
    }
    public Foo[] FooArray
    {
        get
        {
            // ...
        }
        set
        {
            if (value.Length != FooArrayLength.Value)
                throw new ArgumentOutOfRangeException(nameof(FooArray));
            // ...
        }
    }
}
```

7.2.4.5 Native Types

IDL provides a declaration to define an opaque type whose representation is specified by the language mapping. This language mapping specification does not define any native types, but compliant implementations may provide the necessary mechanisms to map native types to equivalent type names in C#.

7.2.4.6 Naming Data Types

C# does not have a **`typedef`** construct; therefore, the declaration of types using **`typedef`** in IDL shall not result in the creation of any C# type. Instead, the use of an IDL **`typedef`** type shall be replaced with the type referenced by the **`typedef`** statement. For nested **`typedef`**s, the **`typedef`**ed type shall be replaced with the original type in the sequence of **`typedef`** statements.

For example the IDL declaration below:

```
typedef long Length;

struct MyType {
    Length my_type_length;
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
public class MyType
{
    public MyType() {...}
    public MyType(MyType object) {...}
    public MyType(int my_type_length) {...}

    public int my_type_length { get; set; }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
public class MyType
{
    public MyType() {...}
    public MyType(MyType object) {...}
    public MyType(int myTypeLength) {...}

    public int MyTypeLength { get; set; }
}
```

NOTE—Implementers of this specification may define exceptions to the rules above to generate custom types as a result of **typedef** statements in the original IDL files. These custom types may provide custom implementations of template types, such as sequences, maps, or arrays. In such cases, the generated types shall conform to the interfaces dictated by the mapping rules for the original IDL type. For example, the IDL declaration:

```
typedef sequence<long> IntSeq;
```

may result in the declaration of the following **public sealed class** implementing the **IList<>** interface:

```
public sealed class IntSeq : IList<int> {...}
```

IntSeq may be used wherever a **sequence<long>**—or a **typedef** reference to it—is required.

7.3 Any

The IDL any type shall be mapped to **Omg.Types.Any** type. The implementation of the **Omg.Types.Any** is platform-specific, and should include operations that allow programmers to insert and access the value contained in an **any** instance as well as the actual type of that value.

7.4 Interfaces – Basic

Each IDL **interface** shall be mapped to a C# **public interface** with the same name as the IDL **interface**, prepending the **"I"** prefix. The C# **interface** shall be defined in the **namespace** corresponding to the IDL **module** of the **interface**. If the IDL **interface** derives from other IDL **interface**s, the equivalent C# **interface** shall be declared to extend the C# **interface**s resulting from the mapping of the base interfaces.

Each attribute defined in the IDL **interface** shall map to a property of the C# **interface**. Properties representing attributes shall have a getter and a setter. If the attribute is read only, the mapping shall omit the setter.

Each operation defined in the IDL **interface** shall map to a method in the C# **interface**. The name of the mapped method shall be the name of the IDL operation. The number and order of the parameters to the mapped method shall be the same as in the IDL operation. The name of the method parameters shall be name of the IDL method argument. The type of the method parameter shall be mapped following the mapping rules defined in this chapter for the specific type.

Lastly, IDL **out** arguments shall be mapped to C# **out** parameters, **inout** arguments to **ref** parameters, and **in** arguments to **in** parameters.

For example, the IDL **interface** declaration below:

```
interface AnInterface {
    attribute long attr;
    readonly attribute long ro_attr;
    void op1(in long i_param, inout long io_param, out long o_param, out Foo fo_param);
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
public interface IAnInterface
{
    int attr { get; set; }
    int ro_attr { get; }
    void op1(int i_param, ref int io_param, out int o_param, out Foo fo_param);
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
public interface IAnInterface
{
    int Attr { get; set; }
    int RoAttr { get; }
    void Op1(int iParam, ref int ioParam, out int oParam, out Foo foParam);
}
```

7.4.1 Exceptions

An IDL **exception** shall be mapped to a C# **class** with the same name as the IDL **exception**. The mapped **class** shall extend the **System.Exception** class.

Any members of the IDL **exception** shall be mapped to properties in the C# **class** following the mapping rules for IDL **struct**s defined in Clause 7.2.4.3.1. The mapped **class** shall define a constructor taking as arguments all the members of the IDL **exception** to set the corresponding properties.

For example, the IDL declarations below:

```
exception AnException {
    long error_code;
};

interface MyInterfaceException {
    void op1(in long in_param) raises(AnException);
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
public class AnException : System.Exception
{
    public AnException() {...}
    public AnException(AnException object) {...}
    public AnException(int error_code) {...}

    public int error_code { get; set; }
}

public interface IMyInterfaceException
{
    void op1(int in_param);
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
public class AnException : System.Exception
{
    public AnException() {...}
    public AnException(AnException object) {...}
    public AnException(int errorCode) {...}

    public int ErrorCode { get; set; }
}

public interface IMyInterfaceException
{
    void Op1(int inParam);
}
```

7.4.2 Interface Forward Declaration

An IDL **`interface`** forward declaration has no mapping to the C# language.

7.5 Interfaces – Full

This building block complements Interfaces – Basic adding the ability to embed declarations such as types, exceptions, and constants in the interface body.

In this case, each IDL interface shall result in the creation of the following C# **`interfaces`** and **`classes`** in the **`namespace`** corresponding to the containing IDL **`module`**:

- A C# **`public interface`** named **`I<InterfaceName>Operations`** with the methods and attributes of the original IDL interface, which shall mapped according to the rules defined for Interfaces – Basic in Clause 7.4.
- A C# **`public interface`** named **`I<InterfaceName>`** inheriting from **`I<InterfaceName>Operations`**.
- A C# **`public class`** named **`<InterfaceName>`** containing the declaration of the classes and exceptions that result from the mapping of all the types and exceptions declared within the IDL **`interface`**, which shall be mapped according to the rules for the corresponding types defined in this chapter. The class may inherit from **`I<InterfaceName>`** to provide an implementation of the IDL **`interface`**'s methods and attributes.

For example, the IDL **`interface`** declaration below:

```
interface FullInterface {
    struct S {
        long a_long;
    };
    const double PI = 3.14;
    void op1(in S s_in);
    attribute long an_attribute;
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
public interface IFullInterfaceOperations
{
    void op1(FullInterface.S s_in);
    int an_attribute { get; set; }
}

public interface IFullInterface : IFullInterfaceOperations
{
}

public class FullInterface : IFullInterface
{
    public sealed class PI
    {
```

```
        public const double Value = 3.14;
    }

    public class S
    {
        public S() {...}
        public S(S object) {...}
        public S(int a_long) {...}

        public int a_long { get; set; }
    }

    public void op1(FullInterface.S s_in) {...}

    public int an_attribute
    {
        get {...}
        set {...}
    }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
public interface IFullInterfaceOperations
{
    void Op1(FullInterface.S sIn);
    int AnAttribute { get; set; }
}

public interface IFullInterface : IFullInterfaceOperations
{
}

public class FullInterface : IFullInterface
{
    public sealed class PI
    {
        public const double Value = 3.14;
    }

    public class S
    {
        public S() {...}
        public S(S object) {...}
        public S(int aLong) {...}

        public int ALong { get; set; }
    }

    public void Op1(FullInterface.S sIn) {...}

    public int AnAttribute
    {
        get {...}
        set {...}
    }
}
```

7.6 Value Types

An IDL **`valuetype`** type shall be mapped to a C# **`abstract class`**.

If the IDL **valuetype** inherits from a base **valuetype**, the mapped **abstract class** shall inherit from the **abstract class** that resulted from mapping the base **valuetype**. If the IDL **valuetype** supports an interface type, the mapped **abstract class** shall implement the corresponding mapped C# interface.

valuetype members shall be mapped onto the abstract class the same way as **struct** members, with the addition that **private** members shall have the C# **protected** access modifier (so that derived concrete classes may access them).

valuetype operations shall be mapped onto the abstract class the same way as for interfaces. Each **valuetype factory** operation shall be mapped onto the **abstract class** to a method returning **void** and accepting the specified **in** parameters, and shall be annotated with the **FactoryAttribute**, a custom attribute within the **Omg.Types** namespace defined by this specification as follows:

```
namespace Omg.Types
{
    public class FactoryAttribute : System.Attribute
    {
        public FactoryAttribute ()
        {
            IsFactory = true;
        }
        public bool  IsFactory { get; set; }
    }
}
```

For example, the IDL **valuetype** declaration below:

```
valuetype VT1 {
    attribute long a_long_attr;
    void vt_op(in long p_long);
    public long a_public_long;
    private long a_private_long;
    factory vt_factory (in long a_long, in short a_short);
};

interface MyInterface {
    void op();
};

valuetype VT2 : VT1 supports MyInterface {
    public long third_long;
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
public abstract class VT1
{
    public int a_long_attr { get; set; }
    public abstract void vt_op(int pLong);
    public int a_public_long { get; set; }
    protected int a_private_long { get; set; }

    [Omg.Types.Factory]
    public abstract void vt_factory(int aLong, short aShort);
}

public interface IMyInterface
{
    void op();
}

public abstract class VT2 : VT1, IMyInterface
{
    public int  third_long { get; set; }
```

```
    public void op() {...}
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
public abstract class VT1
{
    public int ALongAttr { get; set; }
    public abstract void VtOp(int pLong);
    public int APublicLong { get; set; }
    protected int AprivateLong { get; set; }

    [Omg.Types.Factory]
    public abstract void VtFactory(int aLong, short aShort);
}

public interface IMyInterface
{
    void Op();
}

public abstract class VT2 : VT1, IMyInterface
{
    public int ThirdLong { get; set; }
    public void Op() {...}
}
```

7.7 CORBA-Specific – Interfaces

CORBA-specific mappings are defined in Clause A.1 of Annex A: Platform-Specific Mappings.

7.8 CORBA-Specific – Value Types

CORBA-specific mappings are defined in Clause A.1 of Annex A: Platform-Specific Mappings.

7.9 Components – Basic

Basic components have no direct language mapping; they shall be mapped to intermediate IDL, as specified in [OMG-IDL4], and mapped to C# accordingly.

7.10 Components – Homes

Homes have no direct language mapping; they shall be mapped to intermediate IDL, as specified in [OMG-IDL4], and mapped to C# accordingly.

7.11 CCM-Specific

CORBA-specific mappings are defined in Clause A.1 of Annex A: Platform-Specific Mappings.

7.12 Components – Ports and Connectors

Ports and Connectors have no direct language mapping; they shall be mapped to intermediate IDL, as specified in [OMG-IDL4], and mapped to C# accordingly.

7.13 Template Modules

Template module instances have no direct language mapping; they shall be mapped to intermediate IDL, as specified in [OMG-IDL4], and mapped to C# accordingly.

7.14 Extended Data Types

7.14.1 Structures with Single Inheritance

An IDL **`struct`** that inherits from a base IDL **`struct`**, shall be declared as a C# **`public class`** that extends the **`class`** resulting from mapping the base IDL **`struct`**.

The resulting C# **`public class`** shall be mapped according to the general mapping rules for IDL **`struct`**s defined in Clause 7.2.4.3.1 with the following additions:

- The public copy constructor shall call the "all values constructor" of the base class with the value of the members in the new instance that are derived from the base IDL **`struct`**.
- The public "all values constructor" shall take as parameters an object of the base class, followed parameters for each member of the IDL **`struct`**. The "all values constructor" shall call the copy constructor of the base class using the object of the base class provided as a parameter.

NOTE—The derived structure may include additional constructors, such as an "all values constructor" that takes parameters for members of both the base and the derived IDL **`struct`**.

For example, an IDL struct extending the **`MyStruct`** structure defined in Clause 7.2.4.3.1:

```
struct ChildStruct : MyStruct {
    float a_float;
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
public class ChildStruct : MyStruct
{
    public ChildStruct() {...}
    public ChildStruct(ChildStruct object)
        : base(object.a_long, object.a_short, object.a_long_array) {...}
    public ChildStruct(MyStruct parentObject, float a_float)
        : base(parentObject) {...}

    public float a_float { get; set; }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
public class ChildStruct : MyStruct
{
    public ChildStruct() {...}
    public ChildStruct(ChildStruct object)
        : base(object.ALong, object.AShort, object.ALongArray) {...}
    public ChildStruct(MyStruct parentObject, float aFloat)
        : base(parentObject) {...}

    public float AFloat { get; set; }
}
```

7.14.2 Union Discriminators

This building block adds the **`int8`**, **`uint8`**, **`wchar`**, and **`octet`** IDL types to the set of valid types for a discriminator. The mapping of union discriminators of such types shall be mapped as specified in Clause 7.2.4.3.2.

7.14.3 Additional Template Types

7.14.3.1 Maps

An IDL **map** shall be mapped to a C# generic **System.Collections.Generic.IDictionary<TKey,TValue>** instantiated with the equivalent C# key type and value type. In the mapping, everywhere the **map** type is needed, a property of type **IDictionary** with the equivalent C# key type and value type shall be used[5].

Bounds checking shall raise an exception if necessary.

For example the IDL declaration below:

```
struct MyType {
    map<long, string> long_str_map;
    map<string, Foo> str_foo_map;
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
using System.Collections.Generic;
public class MyType
{
    public MyType() {...}
    public MyType(MyType object) {...}
    public MyType(
        IDictionary<int, string> long_str_map,
        IDictionary<string, Foo> str_foo_map) {...}

    public IDictionary<int, string> long_str_map { get; set; }
    public IDictionary<string, Foo> str_foo_map { get; set; }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
using System.Collections.Generic;
public class MyType
{
    public MyType() {...}
    public MyType(MyType object) {...}
    public MyType(
        IDictionary<int, string> longStrMap,
        IDictionary<string, Foo> strFooMap) {...}

    public IDictionary<int, string> LongStrMap { get; set; }
    public IDictionary<string, Foo> StrFooMap { get; set; }
}
```

7.14.3.2 Bitsets

An IDL **bitset** shall map to a C# **struct** with public properties for each named **bitfield** in the set. The IDL type of each **bitfield** member, if not specified in the IDL, shall take the smallest unsigned integer type able to store the bit field with no loss; that is, **byte** if it is between 1 and 8, **ushort** if it is between 9 and 16, **uint** if it is between 17 and 32 and **ulong** if it is between 33 and 64.

For example the IDL declaration below:

```
bitset MyBitset {
    bitfield<3> a;
    bitfield<1> b;
    bitfield<4>;
    bitfield<12,short> d;
    bitfield<20> e;
```

[5] This allows implementers to use different Dictionary implementations (e.g., **System.Collections.Generic.Dictionary<TKey,TValue>**) while remaining interface compliant.

```
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
public struct MyBitset
{
    public byte a { get; set; }
    public byte b { get; set; }
    public short d { get; set; }
    public uint e { get; set; }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
public struct MyBitset
{
    public byte A { get; set; }
    public byte B { get; set; }
    public short D { get; set; }
    public uint E { get; set; }
}
```

7.14.3.3 Bitmask Type

The IDL **bitmask** type shall map to a C# **public enum** with the same name, followed by the **"Flags"** suffix. In the mapping, everywhere the **bitmask** type is needed, a **System.Collections.BitArray** shall be used.

The C# **enum** shall have the **System.FlagsAttribute**, and shall contain a literal for each named member of the IDL **bitmask**. The value of each C# **enum** literal is dictated by the **@position** annotation of the corresponding IDL **bitmask** member. If no position is specified, the C# **enum** literals shall be set to the value of the next power of two. The corresponding **enum** literals can be used to set, clear, and test individual bits in the corresponding **System.Collections.BitArray** instance.

The size (number of bits) held in the **bitmask** determines the corresponding C# **enum** type. In particular, the **enum** type shall be **byte**, for bit bound values between 1 and 8; **ushort**, for bit bound values between 9 and 16; **uint**, for values between 17 and 32; and **ulong** for bit bound values between 33 and 64.

For example the IDL **bitmask** declaration below:

```
bitmask MyBitMask {
    flag0,
    flag1,
    flag2,
    flag3,
    flag4
};

struct BitMaskExample {
    MyBitMask a_bitmask;
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
[Flags]
public enum MyBitMaskFlags
{
    flag0 = 1 << 0,
    flag1 = 1 << 1,
    flag2 = 1 << 2,
    flag3 = 1 << 3,
    flag4 = 1 << 4
}
```

```
public class BitMaskExample
{
    public BitMaskExample() {...}
    public BitMaskExample(BitMaskExample object) {...}
    public BitMaskExample(System.Collections.BitArray a_bitmask) {...}

    public System.Collections.BitArray a_bitmask { get; set; }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
[Flags]
public enum MyBitMaskFlags
{
    Flag0 = 1 << 0,
    Flag1 = 1 << 1,
    Flag2 = 1 << 2,
    Flag3 = 1 << 3,
    Flag4 = 1 << 4
}

public class BitMaskExample
{
    public BitMaskExample() {...}
    public BitMaskExample(BitMaskExample object) {...}
    public BitMaskExample(System.Collections.BitArray aBitmask) {...}

    public System.Collections.BitArray ABitmask { get; set; }
}
```

7.15 Anonymous Types

No impact to the C# language mapping.

7.16 Annotations

7.16.1 Defining Annotations

User-defined annotations are propagated to the generated code as C# attributes inheriting from the **`System.Attribute`** class. The name of the corresponding attributes shall be that of the original IDL annotation, appending the **`"Attribute"`** suffix when applying the *.NET Framework Design Guidelines Naming Scheme* (see Table 8.1).

Each annotation member shall be mapped to a property with public getters and setters. Moreover, the mapped attribute shall have a public constructor with default values (default constructor) and shall be annotated with the following attribute: **`[AttributeUsage(AttributeTargets.All, AllowMultiple = true)]`**. If the IDL annotation definition provides a default value for a given member, it shall be reflected in the C# definition accordingly; otherwise, the equivalent C# definition shall have no default value.

For example, the IDL user-defined **`annotation`** below:

```
@annotation MyAnnotation {
    boolean value default TRUE;
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
[AttributeUsage(AttributeTargets.All, AllowMultiple = true)]
public class MyAnnotation : System.Attribute
{
    public MyAnnotation()
    {
```

```
        this.value = true;
    }

    public MyAnnotation(bool value)
    {
        this.value = value;
    }

    public bool value { get; set; }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
[AttributeUsage(AttributeTargets.All, AllowMultiple = true)]
public class MyAnnotationAttribute : System.Attribute
{
    public MyAnnotationAttribute()
    {
        Value = true;
    }

    public MyAnnotationAttribute(bool value)
    {
        Value = value;
    }

    public bool Value { get; set; }
}
```

7.16.2 Applying Annotations

IDL elements annotated with user-defined annotations shall map to equivalent C# elements annotated with the corresponding attribute following the mappings defined in this specification.

For example, the IDL user-defined annotation below:

```
@annotation MyAnnotation {
    boolean value default TRUE;
};

@MyAnnotation
struct AnnotatedStruct {
    long a_long;
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
[AttributeUsage(AttributeTargets.All, AllowMultiple = true)]
public class MyAnnotation : System.Attribute
{
    public MyAnnotation()
    {
        value = true;
    }

    public bool value { get; set; }
}

[MyAnnotation]
public class AnnotatedStruct
{
    public AnnotatedStruct() {...}
    public AnnotatedStruct(int a_long) {...}
```

```
    public int a_long { get; set; }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
[AttributeUsage(AttributeTargets.All, AllowMultiple = true)]
public class MyAnnotationAttribute : System.Attribute
{
    public MyAnnotationAttribute()
    {
        Value = true;
    }

    public bool Value { get; set; }
}

[MyAnnotation]
public class AnnotatedStruct
{
    public AnnotatedStruct() {...}
    public AnnotatedStruct(AnnotatedStruct object) {...}
    public AnnotatedStruct(int aLong) {...}
    public int ALong { get; set; }
}
```

7.16.2.1 Applying Annotations in Naming Data Types

Annotations on an IDL **`typedef`** shall be applied to uses of the **`typedef`** in other type declarations.

For example the IDL declaration:

```
typedef @max(100) long Length;
struct MyType {
    Length a;
    sequence<Length> a_seq;
};
```

would be mapped as if the IDL declaration had been:

```
struct MyType {
    @max(100) long a;
    sequence<@max(100) long> a_seq;
};
```

7.17 Standardized Annotations

[OMG-IDL4] defines some annotations and assigns them to logical groups. These annotations may be applied to various constructs throughout an IDL document, and their impact on the language mapping is dependent on the context in which they are applied. The following sections describe the impact these defined annotations have on the language mapping, and provide cross references to earlier document sections where the details are given.

7.17.1 Group of Annotations: General Purpose

Table 7.5 identifies the mapping impact of the IDL-defined General Purpose Annotations.

Table 7.5: General Purpose Annotation Impact

General Purpose Annotation	Impact on C# Language Mapping
`@id`	No impact on language mapping
`@autoid`	No impact on language mapping

<table>
<tr>
<td><code>@optional</code></td>
<td>

IDL elements annotated with **@optional** whose type **T** maps to a C# value type shall map to **System.Nullable<T>**. IDL types mapped to reference types shall remain unchanged.

All mapped optional elements shall be annotated with **OptionalAttribute**, a custom attribute within the **Omg.Types** namespace that is defined as follows:

```
namespace Omg.Types
{
    public class OptionalAttribute : System.Attribute
    {
        public OptionalAttribute()
        {
            IsOptional = true;
        }
        public bool IsOptional { get; set; }
    }
}
```

The "all values constructor" (see Clause 7.2.4.3.1) of the mapped C# **class** shall follow the same mapping in the definition of the input parameters associated with optional members, annotating each optional parameter with **Omg.Types.OptionalAttribute** as well.

NOTE—Version 1.1 of the .NET Standard introduced **OptionalAttribute**, an attribute in the **System.Runtime.InteropServices** namespace that may be used to annotate optional parameters; however, the .NET Standard **OptionalAttribute** may not be used to annotate class properties, such as those representing IDL optional members. To mitigate this limitation, this specification introduces an alternative **OptionalAttribute** within the **Omg.Types** namespace that may be used for both optional properties as well. For consistency and simplicity (to minimize name collisions within the declaration of a class generated from the definition of an IDL structure containing optional members) this specification uses the **OptionalAttribute** within the **Omg.Types** namespace to annotate both optional class properties and optional parameters in the "all values constructor."

For example, the IDL declaration:

```
struct Coordinates {
    long x;
    long y;
    @optional long z;
    @optional CoordinatesInfo extra_info;
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
using Omg.Types;
public class Coordinates
{
    public Coordinates() {...}
    public Coordinates(Coordinates object) {...}
    public Coordinates(
        int x,
        int y,
        [Optional] System.Nullable<int> z,
        [Optional] CoordinatesInfo extra_info) {...}

    public int x { get; set; }
```

</td>
</tr>
</table>

	`` public int y { get; set; } [Optional] public System.Nullable<int> z { get; set; } [Optional] public CoordinatesInfo extra_info { get; set; } } ``` or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*: ``` using Omg.Types; public class Coordinates { public Coordinates() {...} public Coordinates(Coordinates object) {...} public Coordinates(int x, int y, [Optional] System.Nullable<int> z, [Optional] CoordinatesInfo extraInfo) {...} public int X { get; set; } public int Y { get; set; } [Optional] public System.Nullable<int> Z { get; set; } [Optional] public CoordinatesInfo ExtraInfo { get; set; } } ```
@position	Impacts the mapping of **bitmask** types as defined in Clause 7.14.3.3.
@value	Impacts the mapping of **enum** types as defined in Clause 7.2.4.3.3.
@extensibility	No impact on language mapping
@final	No impact on language mapping
@mutable	No impact on language mapping
@appendable	No impact on language mapping

7.17.2 Group of Annotations: Data Modeling

Table 7.6 identifies the mapping impact of the IDL defined Data Modeling Annotations.

Table 7.6: Data Modeling Annotation Impact

Data Modeling Annotation	Impact on C# Language Mapping
@key	IDL elements annotated with **@key** shall result in equivalent C# elements annotated with **KeyAttribute**, a custom attribute within the **Omg.Types** namespace defined by this specification as follows: ``` namespace Omg.Types { public class KeyAttribute : System.Attribute { public KeyAttribute() { ```

<table>
<tr><th>Data Modeling Annotation</th><th>Impact on C# Language Mapping</th></tr>
<tr><td></td><td>

```
            IsKey = true;
        }
        public bool IsKey { get; set; }
    }
}
```

For example, the IDL declaration:

```
struct ShapeType {
    @key string color;
    long x;
    long y;
    long shapesize;
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
public class ShapeType
{
    public ShapeType() {...}
    public ShapeType(ShapeType object) {...}
    public ShapeType(
        string color,
        int x,
        int y,
        int shapesize) {...}

    [Omg.Types.Key]
    public string color { get; set; }
    public int x { get; set; }
    public int y { get; set; }
    public int shapesize { get; set; }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
public class ShapeType
{
    public ShapeType() {...}
    public ShapeType(ShapeType object) {...}
    public ShapeType(
        string color,
        int x,
        int y,
        int shapesize) {...}

    [Omg.Types.Key]
    public string Color { get; set; }
    public int X { get; set; }
    public int Y { get; set; }
    public int Shapesize { get; set; }
}
```

</td></tr>
<tr><td>@must_understand</td><td>No impact on language mapping.</td></tr>
<tr><td>@default_literal</td><td>The C# element declared as result of the mappings defined in this specification shall be initialized to element indicated by the annotation.

Following the mapping of IDL enum types map to C# enum types defined in Clause 7.2.4.3.3; the effect of applying @default_literal to an IDL-defined</td></tr>
</table>

Data Modeling Annotation	Impact on C# Language Mapping
	enumeration literal shall be to set the C# **`enum`** value to that of the element to which **`@default_literal`** applies (in the constructor of the **`enum`** class).

7.17.3 Group of Annotations: Units and Ranges

Table 7.7 identifies the mapping impact of the IDL defined Units and Ranges Annotations.

Table 7.7: Units and Ranges Annotation Impact

Units and Ranges Annotation	Impact on C# Language Mapping
`@default`	C# elements declared as result of the mappings defined in this specification containing a **`@default`** annotation shall be initialized to the value of the annotation.
`@range`	C# elements declared as a result of the mappings defined in this specification containing a **`@range`** annotation shall throw a **`System.ArgumentOutOfRangeException`** if they are set to a value out of the corresponding range. Therefore, the setter of a property created as a result of an IDL element annotated with **`@range`** shall implement the corresponding checks and throw a **`System.ArgumentOutOfRangeException`** if the checks fail.
`@min`	C# elements declared as a result of the mappings defined in this specification containing a **`@min`** annotation shall throw a **`System.ArgumentOutOfRangeException`** if they are set to a value smaller than the value of the **`@min`** annotation. Therefore, the setter of a property created as a result of an IDL element annotated with **`@min`** shall implement the corresponding check and throw a **`System.ArgumentOutOfRangeException`** if the check fails.
`@max`	C# elements declared as a result of the mappings defined in this specification containing a **`@max`** annotation, shall throw a **`System.ArgumentOutOfRangeException`** if they are set to a value bigger than the value of the **`@max`** annotation. Therefore, the setter of a property created as a result of an IDL element annotated with **`@max`** shall implement the corresponding check and throw a **`System.ArgumentOutOfRangeException`** if the check fails.
`@unit`	Shall result in the addition of a **`UnitAttribute`** to the mapped element, according to the following definition: `namespace Omg.Types` `{` `    public class UnitAttribute : System.Attribute` `    {` `        public UnitAttribute(string unitName)` `        {` `            UnitName = unitName;` `        }`

	`` public string UnitName { get; set; } } } ```

7.17.4 Group of Annotations: Data Implementation

Table 7.8 identifies the mapping impact of the IDL defined Data Implementation Annotations.

Table 7.8: Data Implementation Annotation Impact

Data Implementation Annotation	Impact on C# Language Mapping
`@bit_bound`	Impacts the mapping of **`bitmask`** as described in Clause 7.14.3.3.
`@external`	No impact on the language mapping.
`@nested`	No impact on the language mapping

7.17.5 Group of Annotations: Code Generation

Table 7.9 identifies the mapping impact of the IDL defined Code Generation Annotations.

Table 7.9: Code Generation Annotation Impact

Code Generation Annotation	Impact on C# Language Mapping
`@verbatim`	Copies verbatim text to the indicated output position when the indicated language is `"*"`, `"c#"`, `"cs"`, or `"csharp"`.

7.17.6 Group of Annotations: Interfaces

Table 7.10 identifies the mapping impact of the IDL defined Interface Annotations.

Table 7.10: Interface Annotation Impact

Interface Annotation	Impact on C# Language Mapping
`@service`	Options are `"CORBA"`, `"DDS"`, `"*"`. Impact is platform-specific.
`@oneway`	Impact is platform-specific.
`@ami`	Impact is platform-specific.

8 IDL to C# Language Mapping Annotations

This chapter defines specialized annotations that extend the standard set defined in [OMG-IDL4] to control the C# code generation.

8.1 @csharp_mapping Annotation

This annotation provides the means to customize the way a number of IDL constructs are mapped to the C# programming language. This annotation can therefore be used to modify the default mapping behavior of the mappings specified in Chapter 7.

The IDL definition of the **`@csharp_mapping`** annotation is:

```
enum

@annotation csharp_mapping {
    enum NamingConvention {
        IDL_NAMING_CONVENTION,
        DOTNET_NAMING_CONVENTION
    };
    NamingConvention apply_naming_convention;
    string constants_container default "Constants";
    string struct_type default "class";
}
```

The behavior associated with each parameter is defined below.

8.1.1 apply_naming_convention Parameter

`apply_naming_convention` specifies whether the IDL to C# language mapping shall apply the *IDL Naming Scheme* or the *.NET Framework Design Guidelines Naming Scheme* when mapping IDL names to C#. In particular,

- If **`apply_naming_convention`** is **`IDL_NAMING_CONVENTION`**, the code generator shall generate type identifiers and names according to the *IDL Naming Scheme*, leaving the name of the corresponding IDL construct unchanged, as shown in Table 8.1.
- If **`apply_naming_convention`** is **`DOTNET_NAMING_CONVENTION`**, the code generator shall generate type identifiers and names according to the *.NET Framework Design Guidelines Naming Scheme*, following the rules defined in Table 8.1 for the corresponding IDL construct.

Table 8.1: Type Identifier and Member Name Mapping According to Naming Schemes

IDL Construct	C# Mapping Naming Convention	
	`apply_naming_convention = IDL_NAMING_CONVENTION`	**`apply_naming_convention = DOTNET_NAMING_CONVENTION`**
Module Name	Name as in IDL definition	Name in Pascal Case
Constant Variable Name	Name as in IDL definition	Name in Pascal Case
Structure Type Name	Name as in IDL definition	Name in Pascal Case
Structure Member Name in Mapped Class Properties	Name as in IDL definition	Name in Pascal Case

IDL Construct	C# Mapping Naming Convention	
	`apply_naming_convention = IDL_NAMING_CONVENTION`	`apply_naming_convention = DOTNET_NAMING_CONVENTION`
Union Type Name	Name as in IDL definition	Name in Pascal Case
Union Member Name in Mapped Class Properties	Name as in IDL definition	Name in Pascal Case
Enumeration Type Name	Name as in IDL definition	Name in Pascal Case
Enumeration Value Name	Name as in IDL definition	Name in Pascal Case
Interface Type Name	Name as in IDL definition, preceded by an "I".	Name in Pascal Case, preceded by an "I".
Interface Attribute Name in Mapped Interface Property	Name as in IDL definition	Name in Pascal Case
Interface Method Name	Name as in IDL definition	Name in Pascal Case
Interface Method Parameter Name	Name as in IDL definition	Name in Camel Case
Exception Type Name	Name as in IDL definition	Name in Pascal Case
Exception Member Name in Mapped Class Property	Name as in IDL definition	Name in Pascal Case
Bitset Type Name	Name as in IDL definition	Name in Pascal Case
Bitfield Name in Bitset Properties Methods	Name as in IDL definition	Name in Pascal Case
Bitfield Name in BitSet Modifier Method Parameter	Name as in IDL definition	Name in Camel Case
Bitmask Type Name	Name as in IDL definition, followed by "Flags" suffix.	Name in Pascal Case, followed by "Flags" suffix.
Annotation Type Name	Name as in IDL definition	Name in Pascal Case, followed by the "Attribute" suffix.

8.1.2 constants_container Parameter

`constants_container` activates the alternative mapping for constants defined in Clause 7.2.3.1 and specifies the name of the C# **`class`** that holds the constants, changing it from its default value (i.e., **`Constants`**) to a user-defined value.

For example, the IDL **`const`** declarations below:

```
@csharp_mapping(constants_container="MathematicalConstants")
module my_math {
    const double PI = 3.141592;
    const double e = 2.718282;
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
namespace my_math
{
    public partial class MathematicalConstants
    {
        public const double PI = 3.141592;
        public const double e = 2.718282;
    }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
namespace MyMath
{
    public partial class MathematicalConstants
    {
        public static double PI = 3.141592;
        public static double E  = 2.718282;
    }
}
```

8.1.3 struct_type Parameter

`struct_type` defines the C# type the IDL **`struct`** type map to. By default, as specified in Clause 7.2.4.3.1, IDL **`struct`**s are mapped to a C# **`class`**. This parameter allows changing the default behavior to map an IDL **`struct`** to a C# **`struct`**.

For example, the IDL **`struct`** declaration below:

```
@csharp_mapping(struct_type="struct")
struct MyStruct {
    long my_long;
    long my_short;
};
```

would map to the following C# according to the *IDL Naming Scheme*:

```
public struct MyStruct
{
    //...

    public int my_long { get; set; }
    public short my_short { get; set; }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
public struct MyStruct
{
    //...

    public int MyLong { get; set; }
    public short MyShort { get; set; }
}
```

Annex A: Platform-Specific Mappings

(normative)

A.1 CORBA-Specific Mappings

This clause describes platform-specific mapping rules that shall be followed when mapping IDL constructs to the C# programming language for CORBA. These mappings rules are built upon the platform-independent rules defined in Chapters 7 and 8 for the building blocks that compose the CORBA profiles defined in Clause 9.2 of [OMG-IDL4].

A.1.1 Exceptions

An IDL **`exception`** shall be mapped to a C# **`Exception`** class following the mapping rules defined in Clause 7.4.1. The resulting C# **`Exception`** class shall inherit from the **`Corba.UserException`** class, which is defined as follows:

```
namespace Corba
{
    public class UserException : System.Exception
    {
    }
}
```

For example, the following IDL;

```
exception AnException {
    long error_code;
};
```

would map to the following C# for CORBA according to the *IDL Naming Scheme*:

```
public class AnException : Corba.UserException
{
    public AnException() {...}
    public AnException(AnException object) {...}
    public AnException(int error_code) {...}
    public int error_code { get; set; }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
public class AnException : Corba.UserException
{
    public AnException() {...}
    public AnException(AnException object) {...}
    public AnException(int errorCode) {...}
    public int ErrorCode { get; set; }

}
```

A.1.2 TypeCode

A CORBA **`TypeCode`** represents type information. The IDL **`TypeCode`** type shall map to a C# **`public class`** named **`Corba.TypeCode`** according to the following definition:

```
namespace Corba
{
    public class TypeCode
    {
        public class Bounds : Corba.UserException {...}
        public class BadKind : Corba.UserException {...}

        public bool equal(Corba.TypeCode tc) {...}
```

```
        public bool equivalent(Corba.TypeCode tc) {...}
        public Corba.TypeCode get_compact_typecode() {...}
        public Corba.TCKind kind() {...}
        public string id() {...}
        public string name() {...}
        public uint member_count() {...}
        public string member_name(uint index) {...}
        public Corba.TypeCode member_type(uint index) {...}
        public Corba.Any member_label(uint index) {...}
        public Corba.TypeCode discriminator_type() {...}
        public int default_index() {...}
        public uint length() {...}
        public Corba.TypeCode content_type() {...}
        public ushort fixed_digits() {...}
        public short fixed_scale() {...}
        public Corba.Visibility member_visibility(uint index) {...}
        public Corba.ValueModifier type_modifier() {...}
        public Corba.TypeCode concrete_base_type() {...}
    }
}
```

Except **Any** (which is defined Clause A.1.4) and **TypeCode**, all types used in the declaration of **TypeCode** shall be derived from their IDL definition in [CORBA-IFC] following the mapping rules defined in Chapter 7, applying the *IDL Naming Scheme* defined in Clause 7.1.1.1. The resulting C# definitions shall be placed under the **Corba** namespace.

NOTE—The use of *IDL Naming Scheme* is mandated here to define classes and interfaces that follow the PIDL names defined in [CORBA-IFC].

A.1.3 Object

The CORBA **Object** interface shall be mapped to C# according to the mapping rules for Interfaces – Full defined in Clause 7.5. The resulting **Object** class and the **IObject** interface shall be placed under the **Corba** namespace. The mapping of the CORBA **Object** interface shall be done according to the IDL *IDL Naming Scheme* defined in Clause 7.1.1.1.

NOTE—The use of *IDL Naming Scheme* is mandated here to define classes and interfaces that follow the PIDL names defined in [CORBA-IFC].

A.1.4 Any

The IDL type **any** maps to a **public class** named **Corba.Any** with the following definition:

```
namespace Corba
{
    public class Any
    {
        public Corba.TypeCode type { get; }

        public void insert_short(short value) {...}
        public short extract_short() {...}

        public void insert_long(int value) {...}
        public int extract_long() {...}

        public void insert_longlong(long value) {...}
        public long extract_longlong() {...}

        public void insert_ushort(ushort value) {...}
        public ushort extract_ushort() {...}
```

```
        public void insert_ulong(uint value) {...}
        public uint extract_ulong() {...}

        public void insert_ulonglong(ulong value) {...}
        public ulong extract_ulonglong() {...}

        public void insert_float(float value) {...}
        public float extract_float() {...}

        public void insert_double(double value) {...}
        public double extract_double() {...}

        public void insert_boolean(bool value) {...}
        public bool extract_boolean() {...}

        public void insert_char(char value) {...}
        public char extract_char() {...}

        public void insert_wchar(char value) {...}
        public char extract_wchar() {...}

        public void insert_octet(byte value) {...}
        public byte extract_octet() {...}

        public void insert_any(Corba.Any value) {...}
        public Corba.Any extract_any() {...}

        public void insert_object(Corba.Object value) {...}
        public Corba.Object extract_object() {...}
    }
}
```

A.1.5 Interfaces

IDL **interface**s shall be mapped to C# according to the mapping rules for Interfaces – Full defined in Clause 7.5. The C# **interface** generated in the mapping shall also inherit from **Corba.IObject**, whereas the C# **class** shall inherit also from **Corba.Object**. **Corba.IObject** and **Corba.Object** are defined in Clause A.1.3.

For example, the IDL **interface** declaration below:

```
interface FullInterface {
    struct S {
        long a_long;
    };
    const double PI = 3.14;
    void op1(in S s_in);
    attribute long an_attribute;
};
```

would map to the following C# for CORBA according to the *IDL Naming Scheme*:

```
public interface IFullInterfaceOperations
{
    void op1(FullInterface.S s_in);
    int an_attribute { get; set; }
}

public interface IFullInterface : Corba.IObject, IFullInterfaceOperations
{
}
```

```
public class FullInterface : Corba.Object, IFullInterface
{
    public sealed class PI
    {
        public const double Value = 3.14;
    }

    public class S
    {
        public S() {...}
        public S(S object) {...}
        public S(int a_long) {...}

        public int a_long { get; set; }
    }

    public void op1(FullInterface.S s_in) {...}

    public int an_attribute
    {
        get {...}
        set {...}
    }
}
```

or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*:

```
public interface IFullInterfaceOperations
{
    void Op1(FullInterface.S sIn);
    int AnAttribute { get; set; }
}

public interface IFullInterface : Corba.IObject, IFullInterfaceOperations
{
}

public class FullInterface : Corba.Object, IFullInterface
{
    public sealed class PI
    {
        public const double Value = 3.14;
    }

    public class S
    {
        public S() {...}
        public S(S object) {...}
        public S(int aLong) {...}

        public int ALong { get; set; }
    }

    public void Op1(FullInterface.S sIn) {...}

    public int AnAttribute
    {
        get {...}
        set {...}
    }
}
```

A.1.6 Value Types

IDL **`valuetype`**s shall be mapped to C# according to the mapping rules for Value Types defined in Clause 7.6.

A.2 DDS-Specific Mappings

DDS requires no additional platform-specific language mappings. Implementations of this specification targeting DDS shall therefore be based solely on the IDL to C# mappings defined in Chapters 7 and 8 for the building blocks that compose the DDS profiles defined in Clause 9.3 of [OMG-IDL4].

Annex B: Building Block Traceability Matrix

(non-normative)

The building block traceability matrix in Table B.1 provides an indication of which clause within this specification addresses each IDL building block.

Table B.1: Building Block Traceability Matrix

Building Block	Section(s)
Core DataTypes	7.2 Core Data Types
Any	7.3 Any
Interfaces – Basic	7.4 Interfaces – Basic
Interfaces – Full	7.5 Interfaces – Full
Value Types	7.6 Value Types
CORBA-Specific – Interfaces	7.7 CORBA-Specific – Interfaces
CORBA-Specific – Value Types	7.8 CORBA-Specific – Value Types
Components – Basic	7.9 Components – Basic
Components – Homes	7.10 Components – Homes
CCM-Specific	7.11 CCM-Specific
Components – Ports and Connectors	7.12 Components – Ports and Connectors
Template Modules	7.13 Template Modules
Extended Data Types	7.14 Extended Data Types
Anonymous Types	7.15 Anonymous Types
Annotations	7.16 Annotations