An OMG IDL4 to C# Language Mapping Publication

IDL4 to C# Language Mapping, Version 1.0

OMG Document Number: formal/21-07-01

Release Date: April 2022

Normative Reference: https://www.omg.org/spec/IDL4-CSHARP

Copyright © 2021, Object Management Group, Inc.
Copyright © 2019, Real-Time Innovations, Inc.
Copyright © 2019, Twin Oaks Computing, Inc.
Copyright © 2019, ADLINK Technology Ltd.
Copyright © 2019, Objective Interface Systems, Inc.
Copyright © 2019, Micro Focus International Plc.

USE OF SPECIFICATION – TERMS, CONDITIONS & NOTICES

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

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

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

GENERAL USE RESTRICTIONS

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

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 9C Medway Road, PMB 274, Milford, MA 01757 U.S.A.

TRADEMARKS

CORBA®, CORBA logos®, FIBO®, Financial Industry Business Ontology®, FINANCIAL INSTRUMENT GLOBAL IDENTIFIER®, IIOP®, IMM®, Model Driven Architecture®, MDA®, Object Management Group®, OMG®, OMG Logo®, SoaML®, SOAML®, SysML®, UAF®, Unified Modeling Language®, UML®, UML Cube Logo®, VSIPL®, and XMI® are registered trademarks of the Object Management Group, Inc.

For a complete list of trademarks, see: https://www.omg.org/legal/tm_list.htm. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

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
9C Medway Road, PMB 274
Milford, MA 01757
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.

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,

[.NET-GUIDE] https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/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

This page intentionally left blank.

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 when the name collides with a C# language keyword, or the `"_"` character when the name collides with a name introduced by this specification.

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 for conflicts with C# keywords, or a prepended `"_"` character for other conflicts.

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
System Exceptions: **`Exception`**, **`ArithmeticException`**, **`ArgumentOutOfRangeException`**,	1.0	1.0

`InvalidOperationException`		
`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.

7.1.4 Mapping Extensibility

Unless otherwise specified, implementers of this specification may extend C# types mapped according to the rules specified in this document to add new constructors and methods, override existing methods, and implement additional interfaces.

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

This specification provides two alternatives for mapping IDL constants to the C# programming language. The Standalone Constants Mapping defined in Clause 7.2.3.1 maps constants to standalone classes, which allows the definition of 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, the Constants Container Mapping defined in Clause 7.2.3.2 groups all constant declarations within a scope in a single class.

7.2.3.1 Standalone Constants Mapping

IDL constants shall be mapped to **`public static 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 static class PI
    {
        public const double Value = 3.141592;
    }
    public static class e
    {
        public const double Value = 2.718282;
    }
    public static 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 static class PI
    {
        public const double Value = 3.141592;
    }
    public static class E
    {
        public const double Value = 2.718282;
    }
    public static class MyString
    {
        public const string Value = "My String Value";
    }
}
```

7.2.3.2 Constants Container Mapping

Every scope containing a constant declaration shall contain a **`public static 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 static 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 static 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 static 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 static 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`

IDL Type	C# Type
`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`
`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# **`Omg.Types.ISequence<T>`** interface, instantiated with the mapped type **`T`** of the sequence elements[3]. In the mapping, everywhere the sequence type is needed, an **`Omg.Types.ISequence<T>`** interface shall be used.Implementations of **`Omg.Types.ISequence<T>`** shall extend

[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 sequence implementations while remaining interface compliant.

System.Collections.Generic.IList<T>, and include all the methods defined below, which have the same signature and semantics of their namesake in the **System.Collections.Generic.List<T>** class:

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;

namespace Omg.Types
{
    public interface ISequence<T> : IList<T>
    {
        int Capacity { get; set; }
        void AddRange(IEnumerable<T> collection);
        ReadOnlyCollection<T> AsReadOnly();
        int BinarySearch(int index, int count, T item, IComparer<T> comparer);
        int BinarySearch(T item);
        int BinarySearch(T item, IComparer<T> comparer);
        ISequence<TOutput> ConvertAll<TOutput>(Converter<T, TOutput> converter);
        void CopyTo(T[] array);
        void CopyTo(int index, T[] array, int arrayIndex, int count);
        bool Exists(Predicate<T> match);
        T Find(Predicate<T> match);
        ISequence<T> FindAll(Predicate<T> match);
        int FindIndex(Predicate<T> match);
        int FindIndex(int startIndex, Predicate<T> match);
        int FindIndex(int startIndex, int count, Predicate<T> match);
        T FindLast(Predicate<T> match);
        int FindLastIndex(Predicate<T> match);
        int FindLastIndex(int startIndex, Predicate<T> match);
        int FindLastIndex(int startIndex, int count, Predicate<T> match);
        void ForEach(Action<T> action);
        int IndexOf(T item, int index);
        int IndexOf(T item, int index, int count);
        void InsertRange(int index, IEnumerable<T> collection);
        int LastIndexOf(T item);
        int LastIndexOf(T item, int index);
        int LastIndexOf(T item, int index, int count);
        int RemoveAll(System.Predicate<T> match);
        void RemoveRange(int index, int count);
        void Reverse();
        void Reverse(int index, int count);
        void Sort();
        void Sort(IComparer<T> comparer);
        void Sort(int index, int count, IComparer<T> comparer);
        T[] ToArray();
        void TrimExcess();
        bool TrueForAll(Predicate<T> match);
    }
}
```

NOTE—Unlike **System.Collections.Generic.IList<T>**, which provides very limited functionality, **Omg.Types.ISequence<T>** defines a comprehensive interface with a one-to-one correspondence with **System.Collections.Generic.List<T>**. The use of an interface instead of a concrete class, such as **List<T>,** allows vendors to define their own sequence implementation (e.g., to provide direct access to elements in a way that makes serialization more efficient) while remaining interface compliant.

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>`	`Omg.Types.ISequence<bool>`
`sequence<char>` `sequence<wchar>`	`Omg.Types.ISequence<char>`
`sequence<int8>`	`Omg.Types.ISequence<sbyte>`
`sequence<uint8>` `sequence<octet>`	`Omg.Types.ISequence<byte>`
`sequence<int16>` `sequence<short>`	`Omg.Types.ISequence<short>`
`sequence<uint16>` `sequence<unsigned short>`	`Omg.Types.ISequence<ushort>`
`sequence<int32>` `sequence<long>`	`Omg.Types.ISequence<int>`
`sequence<uint32>` `sequence<unsigned long>`	`Omg.Types.ISequence<uint>`
`sequence<int64>` `sequence<long long>`	`Omg.Types.ISequence<long>`
`sequence<uint64>` `sequence<unsigned long long>`	`Omg.Types.ISequence<ulong>`
`sequence<float>`	`Omg.Types.ISequence<float>`
`sequence<double>`	`Omg.Types.ISequence<double>`
`sequence<long double>`	`Omg.Types.ISequence<decimal>`

As noted in Clauses 7.2.4.3.1 and 7.2.4.3.2, sequence members of IDL **`struct`**s and **`union`**s map to read-only properties.

NOTE—Such mapping follows a well-known design pattern in C#, where externally visible writable properties of a type that implements **`System.Collections.ICollection`** are implemented as read-only properties. The preferred design pattern to replace a whole collection is to use methods to first remove all elements and then repopulate the entire collection (e.g., using methods like **`Clear()`** and **`AddRange()`**). For more information. see https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca2227?view=vs-2019).

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. Property setters shall perform shallow assignments of reference types and deep copies of value types.
  - In general, properties representing IDL sequences and maps shall include only a getter.
  - As an exception, properties representing sequences and maps that are marked with the **`@external`** annotation (see the Standardized Annotations building block) shall include both a getter and a setter. As described in Clause 7.17.4, properties representing external sequence members are mapped to the more generic **`System.Collections.Generic.IEnumerable<T>`** interface.
- 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.
  - The copy constructor shall perform a deep copy of every member of the structure.
  - Implementations supporting the Standardized Annotations building block shall perform a shallow assignment for members annotated with **`@external`** (see Clause 7.17.4).
- A public constructor that accepts parameters for each member (i.e., the all values constructor). The constructor shall perform shallow assignments of reference types and deep copies of value types.

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.

The class shall implement the **`IEquatable<T>`** interface, where **`T`** is the corresponding class name.

Implementations may add additional constructors and methods, as well as override existing methods. Mapped structures may also implement additional interfaces in addition to the mandatory **`IEquatable<T>`** interface.

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

```
struct MyStruct {
    long a_long;
    short a_short;
    long a_long_array[10];
    sequence<long> a_long_seq;
};
```

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

```
public class MyStruct : IEquatable<MyStruct>
{
    public MyStruct() {...}
    public MyStruct(MyStruct object) {...}
    public MyStruct(
        int a_long,
        short a_short,
        int[] a_long_array,
        Omg.Types.ISequence<int> a_long_seq) {...}
```

```
    public bool Equals(MyStruct other) {...}

    public int a_long { get; set; }
    public short a_short { get; set; }
    public int[] a_long_array { get; set; }
    public Omg.Types.ISequence<int> a_long_seq { get; }
}
```

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

```
public class MyStruct : IEquatable<MyStruct>
{
    public MyStruct() {...}
    public MyStruct(MyStruct object) {...}
    public MyStruct(
        int aLong,
        short aShort,
        int[] aLongArray,
        Omg.Types.ISequence<int> aLongSeq) {...}

    public bool Equals(MyStruct other) {...}

    public int ALong { get; set; }
    public short AShort { get; set; }
    public int[] ALongArray { get; set; }
    public Omg.Types.ISequence<int> ALongSeq { get; }
}
```

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 copy constructor that takes as a parameter an object of the mapped class. The copy constructor shall perform a deep copy of the member selected by the discriminator, if any.
  - Implementations supporting the Standardized Annotations shall perform a shallow assignment if the selected member is annotated with **@external** (see Clause 7.17.4).
- A public read-only property named **Discriminator**.
- A public property with getters and setters for each member. Property setters shall perform shallow assignments of reference types and deep copies of value types.
  - In general, properties representing IDL sequences and maps, shall include only a getter. In such cases:
    - For every sequence member, the union shall define two modifier methods. The first modifier method shall have the following prototype: **void Set<SequenceMemberName>()**. The method shall clear the sequence and update the discriminator value. The second modifier method shall have the following prototype: **void Set<SequenceMemberName>(System.Collections.IEnumerable<T> elements)**, where **T** is the equivalent type of the IDL sequence elements. The method shall clear the sequence, populate it with the elements received as an input, and update the discriminator value.
    - For every map member, the union shall define two modifier methods. The first modifier method shall have the following prototype: **void Set<MapMemberName>()**. The method shall remove all the elements in the property representing the map and update the discriminator value. The second modifier method shall have the following prototype: **void Set<MapMemberName>(System.Collections.IEnumerable<Generic.KeyValuePair<TKey ,Tvalue>> elements)**, where **TKey** is the equivalent key type, and **TValue** is the equivalent value

type. The method shall remove all elements in the equivalent map, populate it with the elements received as an input, and update the discriminator value.

    - As an exception, properties representing sequences and maps that are marked with the **`@external`** annotation (see the Standardized Annotations building block) shall include both a getter and a setter. As described in Clause 7.17.4, properties representing external sequence members are mapped to the more generic **`System.Collections.Generic.IEnumerable<T>`** interface.

- A public property with getters and setters for the member corresponding to the default label, if present.

The normal name conflict resolution rule shall apply (i.e., prepend an **`"_"`**) to the discriminator property name if there is a name clash with the mapped union type name or any of the field names.

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 **`void 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.

The class representing the IDL **`union`** shall implement the **`IEquatable<T>`** interface, where **`T`** is the corresponding class name.

Implementations may add additional constructors and methods, as well as override existing methods. Mapped unions may also implement additional interfaces in addition to the mandatory **`IEquatable<T>`** interface.

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

```
union AUnion switch (octet) {
    case 1:
        long  a_long;
    case 2:
    case 3:
        short a_short;
    case 4:
        sequence<long> a_long_seq;
    default:
        octet a_byte_default;
};
```

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

```
public class AUnion : IEquatable<AUnion>
{

    public AUnion() {...}
    public AUnion(AUnion object) {...}

    public bool Equals(AUnion other) {...}
    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 Omg.Types.ISequence<int> a_long_seq
    {
        get
        {
            // ...
        }
    }

    public void Seta_long_seq()
    {
         a_long_seq.Clear();
         Discriminator = 4;
    }

    public void Seta_long_seq(System.Collections.IEnumerable<int> elements)
    {
         a_long_seq.Clear();
         a_long_seq.AddRange(elements);
         Discriminator = 4;
    }

    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 : IEquatable<AUnion>
{

    public AUnion() {...}
    public AUnion(AUnion object) {...}

    public bool Equals(AUnion other) {...}

    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 Omg.Types.ISequence<int> ALongSeq
    {
        get
        {
```

```
            // ...
        }
    }

    public void SetALongSeq()
    {
         ALongSeq.Clear();
         Discriminator = 4;
    }

    public void SetALongSeq(System.Collections.IEnumerable<int> elements)
    {
         ALongSeq.Clear();
         ALongSeq.AddRange(elements);
         Discriminator = 4;
    }

    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*[5]:

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

public class MyType : IEquatable<MyType>
{
    // ...

    public MyType() {...}
```

[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.

[5] This example, as well as other examples in this Chapter, assumes the Standalone Constants Mapping for constants defined in Clause 7.2.3.1. The Constants Container Mapping defined in Clause 7.2.3.2 could also be applied to these examples.

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

    public bool Equals(MyType other) {...}

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

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

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

public class MyType : IEquatable<MyType>
{
    // ...

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

    public bool Equals(MyType other) {...}

    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 : IEquatable<MyType>
{
    public MyType() {...}
    public MyType(MyType object) {...}
    public MyType(int my_type_length) {...}

    public bool Equals(MyType other) {...}

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

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

```
public class MyType : IEquatable<MyType>
{
    public MyType() {...}
    public MyType(MyType object) {...}
    public MyType(int myTypeLength) {...}

    public bool Equals(MyType other) {...}

    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 **Omg.Types.ISequence<>** interface:

```
public sealed class IntSeq : Omg.Types.ISequence<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 parameters without any modifier.

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 be 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 static class PI
    {
        public const double Value = 3.14;
    }

    public class S : IEquatable<S>
    {
        public S() {...}
        public S(S object) {...}
        public S(int a_long) {...}

        public Equal(S other) {...}

        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 static class PI
    {
        public const double Value = 3.14;
    }

    public class S : IEquatable<S>
    {
        public S() {...}
        public S(S object) {...}
        public S(int aLong) {...}

        public Equal(S other) {...}

        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, IEquatable<ChildStruct>
{
    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 bool Equals(ChildStruct other) {...}

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

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

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

    public bool Equals(ChildStruct other) {...}

    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[6].

Bounds checking shall raise an exception if necessary.

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

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 : IEquatable<MyType>
{
    public MyType() {...}
    public MyType(MyType object) {...}
    public MyType(
        IDictionary<int, string> long_str_map,
        IDictionary<string, Foo> str_foo_map) {...}

    public bool Equals(MyType other) {...}

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

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

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

    public bool Equals(MyType other) {...}

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

As noted in Clauses 7.2.4.3.1 and 7.2.4.3.2, map members of IDL **`struct`**s and **`union`**s map to read-only properties.

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.

The mapped C# **`struct`** shall implement the **`IEquatable<T>`** interface, where **`T`** is the corresponding **`bitset`** name.

For example the IDL declaration below:

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

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

```
public struct MyBitset : IEquatable<MyBitset>
```

```
{
    public bool Equals(MyBitset other) {...}

    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 : IEquatable<MyBitset>
{
   public bool Equals(MyBitset other) {...}

    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 : IEquatable<BitmaskExample>
{
    public BitMaskExample() {...}
    public BitMaskExample(BitMaskExample object) {...}
    public BitMaskExample(System.Collections.BitArray a_bitmask) {...}

    public bool Equals(BitMaskExample other) {...}

    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 : IEquatable<BitmaskExample>
{
    public BitMaskExample() {...}
    public BitMaskExample(BitMaskExample object) {...}
    public BitMaskExample(System.Collections.BitArray aBitmask) {...}

    public bool Equals(BitMaskExample other) {...}

    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

`@optional`	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 : IEquatable<Coordinates>`
	`{`
	`  public Coordinates() {...}`
	`  public Coordinates(Coordinates object) {...}`
	`  public Coordinates(`
	`  int x,`

	<pre> int y,
	        [Optional] System.Nullable<int> z,
	        [Optional] CoordinatesInfo extra_info) {...}
	    public bool Equals(Coordinates other) {...}
	
	    public int x { get; set; }
	    public int y { get; set; }
	    [Optional]
	    public System.Nullable<int> z { get; set; }
	    [Optional]
	    public CoordinatesInfo extra_info { get; set; }
	}</pre> or to the following C# when using the *.NET Framework Design Guidelines Naming Scheme*: <pre>using Omg.Types;
	public class Coordinates : IEquatable<Coordinates>
	{
	    public Coordinates() {...}
	    public Coordinates(Coordinates object) {...}
	    public Coordinates(
	        int x,
	        int y,
	        [Optional] System.Nullable<int> z,
	        [Optional] CoordinatesInfo extraInfo) {...}
	    public bool Equals(Coordinates other) {...}
	
	    public int X { get; set; }
	    public int Y { get; set; }
	    [Optional]
	    public System.Nullable<int> Z { get; set; }
	    [Optional]
	    public CoordinatesInfo ExtraInfo { get; set; }
	
	}</pre>

@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**

Data Modeling Annotation	Impact on C# Language Mapping

	namespace defined by this specification as follows:
	`namespace Omg.Types`
	`{`
	`  public class KeyAttribute : System.Attribute`
	`  {`
	`  public KeyAttribute()`
	`  {`
	`  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 : IEquatable<ShapeType>`
	`{`
	`  public ShapeType() {...}`
	`  public ShapeType(ShapeType object) {...}`
	`  public ShapeType(`
	`  string color,`
	`  int x,`
	`  int y,`
	`  int shapesize) {...}`
	`  public bool Equals(ShapeType other) {...}`
	
	`  [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 : IEquatable<ShapeType>`
	`{`
	`  public ShapeType() {...}`
	`  public ShapeType(ShapeType object) {...}`
	`  public ShapeType(`
	`  string color,`
	`  int x,`
	`  int y,`
	`  int shapesize) {...}`
	`  public bool Equals(ShapeType other) {...}`
	
	`  [Omg.Types.Key]`
	`  public string Color { get; set; }`
	`  public int X { get; set; }`
	`  public int Y { get; set; }`
	`  public int Shapesize { get; set; }`

Data Modeling Annotation	Impact on C# Language Mapping
	}
@must_understand	No impact on language mapping.
@default_literal	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 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`	IDL elements annotated with **`@external`** whose type **`T`** maps to a C# value type shall map to **`System.Nullable<T>`**. IDL types whose type maps to a reference type shall remain unchanged. Setters associated with properties of the mapped type shall perform shallow copies. Constructors copying members of the mapped type shall perform shallow copies as well. Properties created as a result of mapping IDL sequences annotated with **`@external`** shall map to the C# **`System.Collections.Generic.IEnumerable<T>`** interface, instantiated with the mapped type **`T`** of the sequence elements. Such properties, as well as those created to represent IDL maps annotated with **`@external`**, shall provide both a setter and a getter.
`@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.

This page intentionally left blank.

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 different options for mapping constants defined in Clause 7.2.3. To enable the Standalone Constants Mapping defined in Clause 7.2.3.1, **`constants_container`** shall be set to an empty string. To enable the Constants Container Mapping defined in Clause 7.2.3.2, **`constants_container`** shall bet set to a valid string. The default name for the containing class is **`Constants`**.

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;
};

@csharp_mapping(constants_container="")
module my_properties {
    const float speed_of_light = 1080000000;
    const float speed_of_sound = 1234.8;
};
```

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

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

namespace my_properties
{
    public static class speed_of_light
    {
        public const float Value = 1080000000;
    }
    public static class speed_of_sound
    {
        public const float Value = 1234.8;
    }
}
```

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

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

namespace MyProperties
{
    public static class SpeedOfLight
    {
        public const float Value = 1080000000;
    }
    public static class SpeedOfSound
    {
       public const float Value = 1234.8;
    }
}
```

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`**.

When mapping an IDL **`struct`** to C# **`struct`** as a result of this annotation, every setter and constructors shall perform a deep copy, regardless of annotations modifying the copy behavior, such as **`@external`** (see Clause 7.17.4).

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 : IEquatable<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 : IEquatable<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 static class PI
    {
        public const double Value = 3.14;
    }

    public class S : IEquatable<S>
    {
        public S() {...}
        public S(S object) {...}
        public S(int a_long) {...}

        public Equals(S other) {...}

        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 static class PI
    {
        public const double Value = 3.14;
    }

    public class S : IEquatable<S>
    {
        public S() {...}
        public S(S object) {...}
        public S(int aLong) {...}

        public Equals(S other) {...}

        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].

This page intentionally left blank.

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