Date: July 2019

OBJECT MANAGEMENT GROUP

IDL4 to Java Language Mapping

FTF Beta 1

OMG Document Number: ptc/2019-07-02

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

This OMG document replaces the submission document (mars/2019-05-06, Alpha). It is an OMG Adopted Beta Specification and is currently in the finalization phase. Comments on the content of this document are welcome, and should be directed to issues@omg.org by October 01, 2019.

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

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

OMG's Issue Reporting Procedure

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

Table of Contents

Table of Tables

Preface

OMG

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

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

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

OMG Specifications

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

https://www.omg.org/spec

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

OMG Headquarters
109 Highland Avenue
Needham, MA 02494
USA

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

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

Issues

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

1 Scope

This specification defines the mapping of OMG Interface Definition Language v4 [IDL4] to the Java programming language. The language mapping covers all of the IDL constructs in the current Interface Definition Language specification (https://omg.org/spec/IDL) with the exception of middleware specific constructs that are better addressed in separate specifications. The language mapping makes use of modern Java language features as appropriate and natural.

2 Conformance

Conformance to this specification can be considered from two perspectives:

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

Table 2.1: Conformance Points

Implementation	A conformant implementation shall transform IDL input into Java source code output as specified in clause 7.
User	Application source code that conforms to this specification makes use of the Java data types and API's as defined in clause 7. Conformant application source code must 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.

[I2JAV] OMG, IDL to Java Language Mapping, Version 1.3, 2008

[IDL4] OMG, Interface Definition Language, Version 4.2, 2018

[J2SE 8.0] James Gosling, The Java Language Specification Java SE 8 Edition, 2015

[JavaBeans] Graham Hamilton, JavaBeans, 1997

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 [IDL4] clause 7, IDL Syntax and Semantics.

Camel Case

A naming convention that represents phrases composed of multiple words 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”.

Java

Java is a general-purpose computer programming language.

Language Mapping

An association of elements in one language to elements in another language (from IDL to Java, 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”.

Snake Case

A naming convention that represents phrases composed of multiple words using a single word, where every internal word is separated by underscores. For example, the Snake Case representation of the phrase “these are my words” would be “these_are_my_words”.

This specification introduces a variant of snake case named All Uppercase Snake Case, which represents compound words, separated by underscores in all uppercase. In this variant, “these are my words” would be “THESE_ARE_MY_WORDS”.

5 Symbols

The following acronyms are used in this specification.

Table 5.1: Acronyms

Acronym	Meaning
CCM	CORBA Component Model
CORBA	Common Object Request Broker Architecture
DDS	Data Distribution Service
J2SE	Java 2 Platform Standard Edition
IDL	Interface Definition Language

Acronym	Meaning
OMG	Object Management Group

6 Additional Information

6.1 Changes to Adopted OMG Specifications

This specification is an alternative to the existing OMG IDL to Java Mapping specification; it is distinct in that it provides a mapping for the constructs of IDL4, and the mapping exploits newer Java language features.

6.2 Acknowledgments

The following companies submitted this specification:

- ADLINK Technology Ltd.
- Real-Time Innovations, Inc.
- Twin Oaks Computing, Inc.

The following companies supported this specification:

- Kongsberg Defence & Aerospace
- Object Computing, Inc.

This page intentionally left blank.

7 IDL to Java Language Mapping

7.1 General

7.1.1 Names

7.1.1.1 Name Transformation Rules

IDL member names and type identifiers are mapped to corresponding Java names and identifiers according to the rules specified in this chapter. Some of the rules require the transformation of the corresponding IDL member name or type identifier into Pascal Case, Camel Case, or All Uppercase Snake case[1] (see chapter 4 for a formal definition of each term). Below we define the transformation rules for every style.

If the mapped name or identifier collides with a reserved name in 7.1.2, the collision is resolved by prepending an underscore (`"_"`) to the mapped name[2].

7.1.1.1.1 Pascal Case Transformation

An IDL member name or type identifier shall be transformed into Pascal Case according to the following rules:

- If the IDL name or type identifier is all lowercase without any separating underscores, the first character shall be capitalized and the rest of characters shall remain unchanged.
- If the IDL name or type identifier is all uppercase without any separating underscores, it shall remain unchanged.
- If the IDL name or type identifier is separated by underscores, the first letter shall be capitalized, all underscores shall be removed, and the first letter after each underscore shall be capitalized. The rest of characters shall remain unchanged.
- If the IDL name or type identifier mixes uppercase and lowercase, the transition from uppercase to lowercase and from lowercase to uppercase determines the beginning of a new word. The first letter of every transition shall be capitalized, including the first letter of the first word.

For example:

- "pascalcase" shall be transformed into "Pascalcase".
- "PASCALCASE" shall remain "PASCALCASE".
- "Pascal_Case" shall be transformed into "PascalCase", "pascal_case" into "PascalCase", "Pascal_case" into "PascalCase", "Pascal_Case" into "PascalCase", "PASCAL_case" into "PASCALCase", and "PASCAL_CASE" into "PASCALCASE".
- "pascalCase" shall be transformed into "PascalCase", "PascalCase" shall remain "PascalCase", "PASCALcase" shall be transformed into "PASCALCase".

7.1.1.1.2 Camel Case Transformation

An IDL member name or type identifier shall be transformed into Camel Case by following first the conversion rules for Pascal Case defined in 7.1.1.1.1, and then transforming the first letter of the member name or type identifier to lowercase.

For example:

---

[1] The resulting mappings follow naming patterns defined in the JavaBeans 1.01 [JavaBeans] specification.

[2] Implementations of this specification should report as an error collisions caused by the transformation of IDL member names and type identifiers resulting in the same name. For example, without the appropriate error handling, two IDL **`struct`**s named **`MyType`** and **`My_Type`** within the same scope, will be mapped onto two different classes named **`MyType`**.

- “camelcase” shall remain “camelcase”.
- “CAMELCASE” shall become “cAMELCASE”.
- “Camel_Case” shall be transformed into “camelCase”, “camel_case” into “camelCase”, “Camel_case” into “camelCase”, “Camel_Case” into “camelCase”, “CAMEL_case” into “cAMELCase”, and “CAMEL_CASE” into “cAMELCASE”.
- “camelCase” shall remain “camelCase”, “CamelCase” shall be transformed into “camelCase”, “CAMELcase” into “cAMELCase”, and “CAMELCase” into “cAMELCase”.

7.1.1.1.3 All Uppercase Snake Case Transformation

An IDL member name or type identifier shall be transformed into All Uppercase Snake Case according to the following rules:

- If the IDL name or type identifier is in all uppercase with or without separating underscores, the name shall remain unchanged.
- If the IDL name or type identifier is in all lowercase with or without separating underscores, the name shall be transformed to all uppercase keeping the separating underscores—if any.
- If the IDL name or type identifier mixes lowercase and uppercase, the transition from uppercase to lowercase and from lowercase to uppercase after the first letter shall be considered the beginning of a new word. In that case, the IDL name shall be transformed into all uppercase, and an underscore shall be added right before the transition to a new word.

For example:

- “ALL” shall remain “ALL” and “ALL_UPPERCASE” shall remain “ALL_UPPERCASE”.
- “all” shall be transformed into “ALL” and “all_uppercase” shall be transformed into “ALL_UPPERCASE”.
- “allUppercase” shall be transformed into “ALL_UPPERCASE”, “AllUppercase” into “ALL_UPPERCASE”, and “ALLUppercase” into “ALL_UPPERCASE”.

7.1.1.2 Suffixes

In addition, because of the nature of the Java language, a single IDL construct may be mapped to several (differently named) Java constructs. The additional names are constructed by appending a descriptive suffix. If an IDL name ends in a reserved suffix (for example, `Abstract`), then an underscore is prepended to the mapped name. For example, an IDL struct whose name is `FooAbstract` shall be mapped to `_FooAbstract`, regardless of whether another IDL type named `foo` exists. Any synthesized names (for example the abstract class in clause 7.6) will be based on the modified IDL name. For example, the abstract class for `struct FooAbstract` is named `_FooAbstractAbstract.`

7.1.2 Reserved Names

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

- The Java class `<type>Abstract`, where `<type>` is the name of an IDL defined valuetype.
- The Java class `Constants`, defined in each Java package `<moduleName>` resulting from an IDL defined module named `<moduleName>`.
- The keywords in the Java language. For example for the Java Language Specification [J2SE 8.0], clause 3.9 the keywords are:

```
abstract        catch           do
assert          char            double
boolean         class           else
break           const           enum
byte            continue        extends
case            default         final
```

finally	native	switch
float	new	synchronized
for	package	this
goto	private	throw
if	protected	throws
implements	public	transient
import	return	try
instanceof	short	void
int	static	volatile
interface	strictfp	while
long	super	

- The additional Java constants/literals:

true	false	null

- The following names are treated as reserved if used in a context where the mapping collides with the following methods on **java.lang.Object** (from [J2SE 8.0], clause 4.3.2):

clone	notifyAll	getClass
notify	finalize	wait
equals	toString	hashCode

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

7.1.3 Holder class

The following classes shall be used as a box to hold objects of a related type. These holder types are required in cases when an IDL defined data type is passed to an operation as an **inout** or **out** parameter. Primitive types utilize the **Holder<E>** class parameterized with the associated box type (e.g., **Holder<Integer>** for the **int** primitive). Non-primitive types utilize the generic **Holder<E>** class parameterized with the non-primitive type,

```
package org.omg.type;
public class Holder<E> {
    public E value;
};
```

7.1.4 Java Language Version Requirements

Some features of this language mapping depend on certain Java language support that is not available in some older versions of the Java Language. The following table identifies pertinent Java language features, and in which Java language version they become available.

Table 7.1: Java Language Versions and Features

Feature	Java Version Minimum
Enumerations	J2SE 5.0
Generics (e.g., **List<T>**, **Map<K,V>**)	J2SE 5.0
Annotation application (type declaration)	J2SE 5.0
Annotation application (type use)	Java SE 8.0

7.1.5 Code Examples

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

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 clauses.

7.2.2 Modules

An IDL module is mapped to a Java **`package`** with the same name. All IDL declarations within the module are mapped to Java class or interface declarations within the corresponding package.

IDL declarations not enclosed in any modules are mapped to classes or interfaces in the (unnamed) Java global scope.

7.2.3 Constants

Constants are mapped into a **`public final class`** with the name **`Constants`** defined inside the package corresponding to the IDL module where constant was defined, if any (see clause 7.2.2). The class shall contain a **`public final static`** member for each constant in the module.

Constant variable names shall be mapped to variables names in All Uppercase Snake Case, following the transformation rules defined in 7.1.1.1.3.

For example the IDL:

```
module MyMath {
    const double pi_value = 3.141592;
    const double PI_VALUE_CONSTANT = 3.14;
    const double myConstant = 34.5;
    const double constantexample = 22.4;
    const double EValue = 2.718282;
};
```

would result in the Java declarations below:

```
package MyMath;
public final class Constants {
    public final static double PI_VALUE = 3.141592;
    public final static double PI_VALUE_CONSTANT = 3.14;
    public final static double MY_CONSTANT = 34.5;
    public final static double CONSTANTEXAMPLE = 22.4;
    public final static double E_VALUE  = 2.718282;
}
```

7.2.4 Data Types

7.2.4.1 Basic Types

7.2.4.1.1 Integer Types

IDL integer types shall be mapped as shown in Table 7.2.

Table 7.2: Mapping of Integer Types

IDL Type	Java Type
`int8` `uint8`	`byte`

`short`
`int16`
`unsigned short`
`uint16`
	`short`

`long`
`int32`
`unsigned long`
`uint32`
	`int`

`long long`
`int64`
`unsigned long long`
`uint64`
	`long`

7.2.4.1.2 Floating-Point Types

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

Table 7.3: Floating-Point Types Mapping

IDL Type	Java Type
`float`	`float`
`double`	`double`
`long double`	`java.math.BigDecimal`

7.2.4.1.3 Char Types

The IDL **`char`** shall be mapped to the Java primitive type **`char`**[3].

7.2.4.1.4 Wide Char Types

The IDL **`wchar`** shall be mapped to the Java primitive type **`char`**.

7.2.4.1.5 Boolean Types

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

7.2.4.1.6 Octet Type

The IDL type **`octet`**, an 8-bit quantity, shall be mapped to the Java type **`byte`**.

[3] IDL characters are 8-bit quantities representing elements of a character set while Java characters are 16-bit unsigned quantities representing Unicode characters.

7.2.4.2 Template Types

7.2.4.2.1 Sequences

7.2.4.2.1.1 Sequence of Basic Types

IDL sequences of Basic Types shall be mapped to the interfaces shown in Table 7.4. Each interface provides a type-specific sequence interface to the underlying sequence primitives, facilitating a more performant implementation when compared to the `List<E>` generic list interface.

Table 7.4: Mapping of Sequences of Basic Types

IDL Type	Java Interface
`sequence<boolean>`	`BooleanSeq extends java.util.List<Boolean>`
`sequence<char>` `sequence<wchar>`	`CharSeq extends java.util.List<Char>`

`sequence<octet>`
`sequence<int8>`
`sequence<uint8>`
	`ByteSeq extends java.util.List<Byte>`

`sequence<int16>`
`sequence<short>`
`sequence<uint16>`
`sequence<unsigned short>`
	`ShortSeq extends java.util.List<Short>`

`sequence<int32>`
`sequence<long>`
`sequence<uint32>`
`sequence<unsigned long>`
	`IntegerSeq extends java.util.List<Integer>`

`sequence<int64>`
`sequence<long long>`
`sequence<uint64>`
`sequence<unsigned long long>`
	`LongSeq extends java.util.List<Long>`

`sequence<float>`	`FloatSeq extends java.util.List<Float>`
`sequence<double>`	`DoubleSeq extends java.util.List<Double>`
`sequence<long double>`	`BigDecimalSeq extends java.util.List<BigDecimal>`

These type-specific interfaces shall be defined as follows for every primitive type:

```
package org.omg.type;

interface <InterfaceName> extends java.util.List<MappedType> {
    public <InterfaceName>(int initialCapacity);
    public <InterfaceName>(<PrimitiveType>[] array);

    public void add(<PrimitiveType> element);
    public void add(int index, <PrimitiveType> element);
```

```
    public <PrimitiveType> add(<PrimitiveType>[] elements);
    public <PrimitiveType> add(<PrimitiveType>[] elements, int index);
    public <PrimitiveType> add(<PrimitiveType>[] elements, int index, int count);

    public <PrimitiveType> get(int index);
    public void set(int index, <PrimitiveType> element);
    public void set(int dstIndex, <PrimitiveType>[] elements,
                    int srcIndex, int length);

    public <PrimitiveType>[] toArray(<PrimitiveType>[] array);
}
```

Where:

- **`<IntarfaceName>`** is the interface name indicated in Table 7.4.
- **`<MappedType>`** is the corresponding primitive type in Java, following the mapping rules specified in clause 7.2.4.1.

Bounds checking on bounded sequences shall raise a **`java.lang.IndexOutOfBoundsException`** exception if necessary.

For example, the interface for **`BooleanSeq`** would be:

```
package org.omg.type;

interface BooleanSeq extends java.util.List<Boolean> {
    public BooleanSeq(int initialCapacity {...}
    public BooleanSeq(boolean[] array) {...}

    public void add(boolean element) {...}
    public void add(int index, boolean element) {...}

    public boolean add(boolean[] elements) {...}
    public boolean add(boolean[] elements, int index) {...}
    public boolean add(boolean[] elements, int index, int count) {...}

    public boolean get(int index) {...}
    public void set(int index, boolean element) {...}
    public void set(int dstIndex, boolean[] elements,
                    int srcIndex, int length) {...}

    public boolean[] toArray(boolean[] array) {...}
}
```

7.2.4.2.1.2 Sequence of non Basic Types

IDL **`sequence`**s of non basic types shall be mapped to the Java generic **`java.util.List<E>`** interface, instantiated with the mapped type **`E`** of the sequence element. In the mapping, everywhere the sequence type is needed, a **`List<E>`** shall be used.

Bounds checking on bounded **`sequence`**s shall raise a **`java.lang.IndexOutOfBoundsException`** exception shall be raised if necessary.

For example the IDL declaration:

```
struct Foo {
    ...
};

struct MyType {
     sequence<long> long_sequence;
     sequence<Foo>  foo_sequence;
```

```
};
```

shall result in the Java classes:

```
import java.util.List;

public class Foo implements java.io.Serializable {
    ...
};

public class MyType implements java.io.Serializable {
    public MyType() {...}
    public MyType(LongSeq longSequence, List<Foo> fooSequence) {...}
    public LongSeq getLongSequence() {...}
    public void setLongSequence(LongSeq longSequence) {...}
    public List<Foo> getFooSequence() {...}
    public void setFooSequence(List<Foo> fooSequence) {...}
};
```

7.2.4.2.2 Strings

The IDL **`string`**, both bounded and unbounded variants, shall be mapped to **`java.lang.String`**.

Range checking for characters in the **`string`** as well as bounds checking of the **`string`** shall raise a **`java.lang.IndexOutOfBoundsException`** exception if necessary.

7.2.4.2.3 Wstrings

The IDL **`wstring`**, both bounded and unbounded variants, shall be mapped to **`java.lang.String`**.

Range checking for characters in the **`string`** as well as bounds checking of the **`string`** shall raise a **`java.lang.IndexOutOfBoundsException`** exception if necessary.

7.2.4.2.4 Fixed Type

The IDL **`fixed`** type shall be mapped to the Java **`java.math.BigDecimal`** class. Range checking shall raise a **`java.lang.ArithmeticException`** if necessary.

7.2.4.3 Constructed Types

7.2.4.3.1 Structures

An IDL **`struct`** shall be mapped to a Java **`public class`** of the same name in Pascal Case according to the transformation rules defined in 7.1.1.1.1. The class shall provide the following:

- implements **`java.io.Serializable`**[4]
- a public accessor (getter) method for each member
- a public modifier (setter) method for each member
- a public constructor that accepts parameters for each members (the all values constructor)
- a public constructor that takes no parameters (the default constructor)

The all values constructor shall initialize member fields from the corresponding parameter.

The default constructor shall initialize member fields as follows:

[4] Implementers of this specification may override the default Java serialization by providing an implementation of the **`writeObject()`** and **`readObject()`** method.

- all primitive members shall be left as initialized by the Java default initialization
- all string members shall be initialized to the empty string (**""**)
- all array members shall be initialized to an array of 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
- these rules may be modified by annotations as described in clause 8.

The naming of the accessor and modifier methods shall follow the pattern `get<MemberName>()` and `set<MemberName>()`. The mapped `<MemberName>` shall be spelled in Pascal Case according to the transformation rules defined in 7.1.1.1.1. The accessor return type shall match the member type; the modifier method accepts a parameter of the member type. Lastly, the mapped parameter name shall be spelled in Camel Case following the transformation rules defined in 7.1.1.1.2.

For example, the following IDL:

```
struct S1 {
    long long_variable;
    short short_variable;
    long long longLongVariable;
    string URL;
};
```

maps to the following Java:

```
public class S1 implements java.io.Serializable {
    public S1() {...}
    public S1(int longVariable, short shortVariable,
              long longLongVariable, String URL) {...}
    public int getLongVariable() {...}
    public void setLongVariable(int longVariable) {...}
    public short getShortVariable() {...}
    public void setShortVariable(short shortVariable) {...}
    public long getLongLongVariable() {...}
    public void setLongLongVariable(long longLongVariable) {...}
    public String getURL() {...}
    public void setURL(String URL) {...}
};
```

7.2.4.3.2 Unions

An IDL union shall be mapped to a Java `public final class` with the same name in Pascal Case, following the transformation rules defined in 7.1.1.1.1.

The class shall implement `java.io.Serializable` and provide the following:

- implements `java.io.Serializable`
- a public default constructor, which shall set the discriminator to the default value for the discriminator type. If this selects a branch, then the selected member shall also be set to the default value for the member type.
- a public accessor method for the discriminator, named `getDiscriminator()`
- a public accessor method for each member
- a public modifier method for each member

- for each member that has more than one case label, an additional public modifier method that takes the discriminator value
- a public modifier method for the member corresponding to the default label, if present
- a public default modifier method, if needed

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

The member accessor and modifier methods shall be named **get<MemberName>()** and **set<MemberName>()**, respectively, where **<MemberName>** shall be the corresponding member name in Pascal Case following the transformation rules defined in 7.1.1.1.1. The accessor method return type shall match the member type; the modifier method accepts a parameter of the member type, which shall have the **<MemberName>** spelled in Camel Case according to the transformation rules defined in 7.1.1.1.2. Accessor methods shall raise a **java.lang.IllegalStateException** exception if the expected member has not been set.

If there is more than one case label corresponding to a member, an extra modifier **set<MemberName>()** method that takes an explicit **discriminator** parameter of the discriminator type shall be generated. The extra modifier method shall throw a **java.lang.IllegalArgumentException** exception when a value is passed for the discriminator that is not among the case labels for the member.

If a member corresponds to the default case label, its simple modifier shall set the discriminant to the first available default value starting from a 0 index of the discriminant type. In addition, an extra modifier that takes an explicit discriminator parameter shall be generated. The extra modifier method shall throw a **java.lang.IllegalArgumentException** exception when a value is passed for the discriminator that is not among the case labels for the default branch.

Two default modifier methods, both named **__default()**, are generated if there is no explicit default case label, and the set of case labels does not completely cover the possible values of the discriminant. The first modifier method shall take no arguments, return void, and setsthe discriminant to the first available default value starting from a 0 index of the discriminant type. The second modifier method method shall take a **discriminator** parameter of the discriminator type and return void. Both methods shall leave the union with a discriminator value set, and the value member uninitialized.

For example, the following IDL:

```
union U1 switch (octet) {
    case 1:  long  long_variable;
    case 2:
    case 3:  short short_variable;
    default: octet octet_variable;
};
```

maps to the following Java:

```
final public class U1 implements java.io.Serializable {
    public U1() {...}
    public byte getDiscriminator() {...}

    public int getLongVariable() {...}
    public void setLongVariable(int val) {...}
    public short getShortVariable() {...}
    public void setShortVariable(short shortVariable) {...}
    public void setShortVariable(short shortVariable, byte discriminator) {...}
    public byte getOctetVariable() {...}
    public void setOctetVariable(byte octetVariable) {...}
    public void setOctetVariable(byte octetVariable, byte discriminator) {...}
};
```

Accordingly, the following IDL:

```
union U2 switch (long) {
      case 1:  short short_variable;
      case 2:  long long_variable;
};
```

maps to the following Java:

```
final public class U2 implements java.io.Serializable {
    public U2() {...}
    public int getDiscriminator() {...}
    public int getShortVariable() {...}
    public void setShortVariable(short shortVariable) {...}
    public long getLongVariable() {...}
    public void setLongVariable(long longVariable) {...}
    public void __default() {...}
    public void __default(int discriminator) {...}
}
```

7.2.4.3.3 Enumerations

An IDL **enum** shall be mapped to a Java **public enum** with the same name as the IDL **enum** type in Pascal Case, following the transformation rules defined in 7.1.1.1.1.

The Java **enum** type shall include a list of the enumerators, a private member to hold the value, and a private constructor to initialize the enumerators with the constant value and name. Additionally, the Java **enum** type shall have the helper method **valueOf(int)** to get an enumerator instance from an **int**.

For example, the IDL:

```
enum AnEnum { @value(1) ONE, @value(2) TWO };
```

Maps to:

```
public enum AnEnum {
    ONE(1),
    TWO(2);

    private int value;
    private AnEnum(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
    public static AnEnum valueOf(int v) {
        // return ONE, TWO, or raise java.lang.RuntimeException
    }
};
```

7.2.4.3.4 Constructed Recursive Types

Constructed recursive types are supported by mapping the involved types directly to Java as described elsewhere in clause 7.

7.2.4.4 Arrays

An IDL array shall be mapped to a Java array of the mapped element type. In the mapping, everywhere the array type is needed, an array of the mapped element type shall be used. Bound violations for the array shall raise a **java.lang.IndexOutOfBoundsException** exception.

For example the IDL declaration[5]:

```
const long foo_array_length = 200;

struct S2 {
    long array1[100];
    short array2[10];
    Foo array3[foo_array_length];
    Bar array4[12];
};
```

shall result in the Java classes:

```
public final class Constants {
     public final static double FOO_ARRAY_LENGTH = 200;
};

public class S2 implements java.io.Serializable {
    public S2() {...}
    public S2(int[] array1, short[] array2, Foo[] array3, Bar[] array4) {...}
    public int[] getArray1();
    public void setArray1(int[] array1);
    public short[] getArray2();
    public void setArray2(short[] array2);
    public Foo[] getArray3();
    public void setArray3(Foo[] array3);
    public Bar[] getArray4();
    public void setArray4(Bar[] array4);
};
```

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.

7.2.4.6 Naming Data Types [typedef]

Java does not have a **`typedef`** construct; therefore, the IDL **`typedef`** does not result in any Java types. The use of an IDL typedef type shall be replaced with the type referenced by the **`typedef`** type. This rule shall apply recursively.

For example the IDL declaration:

```
typedef long Length;

struct S3 {
    Length member_length;
};
```

shall result in the Java classes:

```
public class S3 implements java.io.Serializable {
    public S3() {...}
    public S3(int memberLength) {...}
    public int getMemberLength() {...}
    public void setMemberLength(int memberLength) {...}
};
```

That is, the **`typedef`** type **`Length`** is replaced with IDL **`long`** (i.e., the type it references) which then maps to Java as **`int`**.

---

5 The length of the array can be made available in the mapped Java 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.

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> lengths;
};
```

shall be mapped as if the IDL declaration had been:

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

7.3 Any

The IDL any type shall be mapped to org.omg.type.Any type. The implementation of the org.omg.type.Any is middleware 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 Java public interface with the same name in Pascal Case, following the transformation rules defined in 7.1.1.1.1. The Java interface shall be defined in the package corresponding to the IDL module of the interface. If the IDL interface derives from other IDL interfaces, then the Java interface shall be declared to extend the Java classes resulting from the mapping of the base interfaces.

Each attribute defined in the IDL interface shall map to two methods in the Java interface: One method to get the attribute and the other to set the attribute. The name of the get and set methods shall be get<AttributeName>() and set<AttributeName>(), respectively. Where <AttributeName> is he name of the attribute in Pascal Case, applying the transformation rules defined in 7.1.1.1.1. The get method shall take no parameters and its return type shall match the type of the attribute. The set method shall take one parameter of the type of the attribute, and shall return no value. If the attribute is readonly, the set method shall be omitted.

Each operation defined in the IDL interface shall map to a method in the Java interface. The name of the method shall be the same as the name of the IDL operation in Camel Case, following the transformation rules defined in 7.1.1.1.2. The number and order of the method arguments shall be as defined in the IDL. The types of arguments to the method shall be mapped as defined in clause 7.1.1, and their names shall be represented in Camel Case applying the transformation rules defined in 7.1.1.1.2. The method declaration shall specify any exceptions listed in the IDL with a throws clause. Any out or inout arguments shall be mapped to their Holder types.

For example, the following IDL:

```
interface AnInterface {
    attribute long long_attribute;
    readonly attribute long long_ro_attribute;
    void op1(in long inParam, inout long inOutParam, out long outParam);
};
```

maps to the following Java:

```
public interface AnInterface  {
    public AnInterface() {...}
    public AnInterface(int longAttribute, int longRoAttribute) {...}
    public int getLongAttribute() {...}
    public void setLongAttribute(int longAttribute) {...}
    public int getLongRoAttribute() {...}
```

```
        public abstract void op1(int inParam,
                                 org.omg.type.Holder<Integer> inOutParam,
                                 org.omg.type.Holder<Integer> outParam);
    };
```

7.4.1 Exceptions

An IDL **exception** shall be mapped to a Java class extending the **java.lang.RuntimeException** class with the same name as the IDL exception in Pascal Case, applying the transformation rules defined in 7.1.1.1.1. Any members in the IDL exception are mapped to members in the Java class following the rules of the IDL **struct** mapping defined in 7.2.4.3.1. The mapped exception shall also include constructors that follow the rules of the IDL **struct** mapping as well.

For example, the following IDL:

```
exception CustomException {
    long error_code;
;
interface InterfaceException {
    void op1(in long in_param) raises(AnException);
};
```

maps to the following Java:

```
public class CustomException extends java.lang.RuntimeException {
    public CustomException() {...}
    public CustomException(int errorCode) {...}
    public int getErrorCode();
    public void setErrorCode();
};

public interface InterfaceException {
    void op1(int inParam) throws CustomException;
};
```

7.4.2 Interface Forward Declaration

An interface forward declaration has no mapping to the Java language.

7.5 Interfaces – Full

This building block complements Interfaces – Basic adding the ability to embed in the interface body additional declarations such as types, exceptions, and constants. The embedded elements (types, exceptions, and constants) shall be mapped to a public declaration within the scope of the Java interface.

For example, the following IDL:

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

shall result in the Java:

```
public interface  FullInterface  {
    public class S implements java.io.Serializable {
        public S() {...}
        public S(int a) {...}
```

```
        public int getA() {...}
        public void setA(int a) {...}
    };

    public static final float PI = 3.14;
    public abstract void op1(S sIn);
};
```

7.6 Value Types

An IDL **valuetype** type shall be mapped to two Java classes:

- A helper abstract class with the suffix **Abstract** (the "abstract" class).
- A class with the same name as the IDL **valuetype** (the "non-abstract" class).

The mapped non-abstract class shall inherit from the abstract class. If the IDL **valuetype** inherits from a base **valuetype**, the mapped abstract class shall inherit from the non-abstract class that resulted from mapping the base **valuetype**. If the IDL **valuetype** supports an **interface** type, then the mapped abstract class shall **implement** the corresponding mapped Java interface.

The **valuetype** members shall be mapped onto the abstract class the same way as class members, with the addition that **private** members are protected with the Java **protected** access modifier. The **valuetype** operations shall be mapped onto the abstract class the same way as for interfaces. Each **valuetype** initializer (i.e., **factory** operation) is mapped onto the abstract class to a method returning **void** and accepting the specified **in** parameters.

The non-abstract class has **@Override** for all the methods in the abstract class and any implemented interfaces, and it is expected to fill them. These operations have empty implementations (or throw a not-implemented exception).

References to the value type from other classes map to references to the non-abstract class.

For example, the following IDL:

```
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;
};
```

shall result in the Java:

```
public abstract class VT1Abstract  {
    public int  aLongAttr;
    public abstract void vtOp(int pLong);
    public int aPublicLong;
    protected int aPrivateLong;
    public abstract void vtFactory(int aLong, short aShort);
};
public class VT1 extends VT1Abstract {
    public VT1() {...}
    @Override
    public void vtOp(int pLong) {...}
    @Override
    public void vtFactory(int aLong, short aShort) {...}
```

```
};
public interface MyInterface  {
    ...
};
public abstract class VT2Abstract extends VT1 implements MyInterface {
    ...
};
public class VT2 extends VT2Abstract {
    ...
};
```

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 [IDL4], and mapped to Java accordingly.

7.10 Components – Homes

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

7.13 Template Modules

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

7.14 Extended Data Types

7.14.1 Structures with Single Inheritance

If the IDL **`struct`** inherits from a base IDL **`struct`**, then the Java **`class`** shall be declared to extend the base **`class`** that resulted from mapping the base IDL **`struct`**. The "all values" constructor for the derived **`struct`**'s Java **`class`** shall take as its first parameter a non-null instance of the base **`struct`**'s Java class.

For example, extending the IDL **`struct`** in clause 7.2.4.3.1 with the following:

```
struct S5 : S1 {
    float float_variable;
};
```

maps to the following additional Java:

```
public class S5 extends S1 implements java.io.Serializable {
    public S5() {...}
    public S2(S1 parent, float floatVariable) {...}
    public float getFloatVariable() {...}
    public void setFloatVariable(float floatVariable) {...}
};
```

7.14.2 Union Discriminators

This IDL4 block adds **`int8`**, **`uint8`**, **`wchar`** and **`octet`** to the set of valid types for a discriminator. The mapping of these union discriminator types are covered 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 Java generic **`java.util.Map`** instantiated with the Java equivalent key type and value type. In the mapping, everywhere the map type is needed, a Map of the key type and value type shall be used. If the IDL type of the key or the value is a Basic Type, the mapped type shall be the Java boxed type specified in the table below. For example, if the IDL key type is **`int32`**, the map shall have key of type **`Integer`**.

Table 7.5: Mapping of Map key type

IDL Basic Type	Java Boxed Type
`boolean`	`Boolean`
`char` `wchar`	`Char`

`octet`
`int8`
`uint8`
	`Byte`

`int16` `short`	`Short`
`uint16` `unsigned short`	`Integer`
`int32` `long`	`Integer`
`uint32` `unsigned long`	`Long`
`int64` `long long`	`Long`
`uint64` `unsigned long long`	`java.math.BigInteger`
`float`	`Float`
`double`	`Double`

IDL Basic Type	Java Boxed Type
`long double`	`java.math.BigDecimal`

Bounds checking shall raise a **`java.lang.IndexOutOfBoundsException`** exception if necessary.

For example the IDL declaration:

```
struct S4 {
     map<long, string> map1;
     map<string, Foo> map2;
};
```

shall result in the Java classes:

```
public class S4 implements java.io.Serializable {
    public S4() {...}
    public S4(java.lang.Map<Integer, String> map1,
              java.lang.Map<String,Foo> map2) {...}
    public java.lang.Map<Integer, String> getMap1() {...}
    public void setMap1(java.lang.Map<Integer, String> map1) {...}
    public java.lang.Map<String,Foo> getMap2() {...}
    public void setMap2(java.lang.Map<String, Foo> map2) {...}
}
```

7.14.3.2 Bitsets

An IDL **`bitset`** shall map to Java as a **`public class`** with the same name in Pascal Case, following the transformation rules defined in 7.1.1.1.1.

The class shall contain accessor and modifier methods for each named **`bitfield`** in the set. The naming of the accessor and modifier methods shall follow the pattern **`get<BitfieldName>()`** and **`set<BitfieldName>()`**. The mapped **`<BitfieldName>`** shall be spelled in Pascal Case according to the transformation rules defined in 7.1.1.1.1. The accessor method return type shall match the member type; the modifier method accepts a parameter of the member type, which shall have the **`<BitfieldName>`** spelled in Camel Case according to the transformation rules defined in 7.1.1.1.2.

The IDL type of each **`bitfield`** member, if not specified in the IDL, shall take as default value the smallest type able to store the bit field with no loss (i.e. **`boolean`** if size is 1, **`octet`** if it is between 2 and 8, **`unsigned short`** if it is between 9 and 16, **`unsigned long`** if it is between 17 and 32 and **`unsigned long long`** if it is between 33 and 64).

For example the IDL declaration:

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

maps to the following Java:

```
public class MyBitset {
    public byte getA() {...}
    public void setA(byte a) {...}
    public boolean getB() {...}
    public void setB(boolean b) {...}
    public short getC() {...}
    public void setC(short c) {...}
```

```
        public short setD() {...}
        public void getD(short d) {...}
    };
```

7.14.3.3 Bitmask type

The IDL **bitmask** type shall map to a Java **enum** and a **java.util.BitSet**. The Java **enum** name is the IDL **bitmask** name in Pascal Case, following the transformation rules defined in 7.1.1.1.1, with the **Flags** suffix appended.

The Java **enum** shall contain a member for each named member of the IDL **bitmask**. The value of each Java **enum** member is dictated by the position property (**@position**) of the corresponding IDL **bitmask** member. If no position is specified for a literal, the Java **enum** literal shall be set to the value of the next power of 2, relative to the previous literal. The **enum** constants can be used to set, clear, and or test individual bits in the **java.util.BitSet** instance[6].

If the size (number of bits) exceeds that specified by the **@bit_bound** annotation, a **java.lang.IndexOutOfBoundsException** exception shall be raised.

For example:

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

    struct BitmaskExample {
        MyBitMask a_bitset;
    };
```

Is mapped to:

```
    enum MyBitMaskFlags {
        flag0, flag1, flag2
    };

    class BitmaskExample implements java.io.Serializable {
        java.util.BitSet  aBitset;
    };
```

7.15 Anonymous Types

No impact to the Java language mapping.

7.16 Annotations

7.16.1 Defining Annotations

User-defined annotations may be propagated to the generated code. If user defined annotations are mapped to Java, then the following requirements apply.

An IDL annotation type named **<AnnotationName>**, defining members **<Member1>** through **<MemberN>**, shall be represented by the following Java annotation types:

```
    public @interface <AnnotationName> {
       <Member1Type>  <Member1Name>() [ default <DefaultValue> ];
          ...
       <MemberNType>  <MemberNName>() [ default <DefaultValue > ];
    }
```

[6] In addition to **set()**, **clear()**, and **get()** to operate on individual bits in the **bitset**, the **java.util.BitSet** implementation provides common logical operations such as AND, OR, XOR etc, which are also useful.

```
public @interface <AnnotationName>Group {
   <AnnotationName>[] getValue();
}
```

The **<MemberXType>** shall be the Java type corresponding to the type of the IDL member. If a default value is specified for a given member, it shall be reflected in the Java definition. Otherwise, the Java definition shall have no default value.

For example, the IDL user defined annotation,

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

maps to Java like this:

```
public @interface MyAnnotation {
    boolean value() default true;
};
public @interface MyAnnotationGroup {
    MyAnnotation[] value();
};
```

7.16.2 Applying User-Defined Annotations

For each IDL element to which a single instance user-defined annotation is applied, the corresponding Java element shall be annotated with the mapped Java annotation of the same name.

For example, the IDL user defined annotation,

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

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

maps to Java like this:

```
public @interface MyAnnotation {
    boolean value() default true;
};
public @interface MyAnnotationGroup {
    MyAnnotation[] value();
};

@MyAnnotation
public class AnnotatedStruct {
    public int aLong;
};
```

For each IDL element to which multiple instances of the annotation are applied, the corresponding Java element shall be annotated with the mapped annotation bearing the **Group** suffix; each application of the user-defined annotation shall correspond to a member of the array in the group.

For example, the IDL user defined annotation,

```
@annotation MyAnnotation {
```

```
        boolean value default TRUE;
    };

    @MyAnnotation(true)
    @MyAnnotation(false)
    struct MultiAnnotatedStruct {
        long a_long;
    };
```

maps to Java like this:

```
    public @interface MyAnnotation {
        boolean value() default true;
    };
    public @interface MyAnnotationGroup {
        MyAnnotation[] value();
    };

    @MyAnnotationGroup({@MyAnnotation(value=true), @MyAnnotation(value=false)})
    public class MultiAnnotatedStruct {
        public int aLong;
    };
```

7.17 Standardized Annotations

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

7.17.1 Group of Annotations: General Purpose

Table 7.6 identifies the mapping impact of the IDL defined General Purpose Annotations.

Table 7.6: General Purpose Annotation Impact

General Purpose Annotation	Impact on Language Mapping
`@id`	No impact on mapping
`@autoid`	No impact on mapping
`@optional`	Replaces type with boxed type, for Basic Types. No impact on other types.
`@position`	Impacts the mapping of `bitmask`. See clause 7.14.3.3.
`@value`	Impacts the mapping of `enum`. See clause 7.2.4.3.3.
`@extensibility`	No impact on mapping
`@final`	No impact on mapping
`@mutable`	No impact on mapping
`@appendable`	No impact on mapping

7.17.2 Group of Annotations: Data Modeling

Table 7.7 identifies the mapping impact of the IDL defined Data Modeling Annotations.

Table 7.7: Data Modeling Annotation Impact

Data Modeling Annotation	Impact on Language Mapping
`@key`	No impact on mapping
`@must_understand`	No impact on mapping
`@default_literal`	Value used in default constructor

7.17.3 Group of Annotations: Units and Ranges

Table 7.8 identifies the mapping impact of the IDL defined Units and Ranges Annotations.

Table 7.8: Units And Ranges Annotation Impact

Unit and Ranges Annotation	Impact on Language Mapping
`@default`	Value used in default constructor
`@range`	The provided value is tested in the member modifier (setter), and a **`java.lang.IllegalArgumentException`** is raised if the parameter does not meet requirements
`@min`	The provided value is tested in the member modifier (setter), and a **`java.lang.IllegalArgumentException`** is raised if the parameter does not meet requirements
`@max`	The provided value is tested in the member modifier (setter), and a **`java.lang.IllegalArgumentException`** is raised if the parameter does not meet requirements
`@unit`	No impact on mapping

7.17.4 Group of Annotations: Data Implementation

Table 7.9 identifies the mapping impact of the IDL defined Data Implementation Annotations.

Table 7.9: Data Implementation Annotation Impact

Data Implementation Annotation	Impact on Language Mapping
`@bit_bound`	Impacts the mapping of **`bitmask`**. See clause 7.14.3.3.
`@external`	Replaces type with boxed type, for Basic Types. No impact on other types.
`@nested`	No impact on mapping

7.17.5 Group of Annotations: Code Generation

Table 7.10 identifies the mapping impact of the IDL defined Code Generation Annotations.

Table 7.10: Code Generation Annotation Impact

Code Generation Annotation	Impact on Language Mapping
`@verbatim`	Copies verbatim text to the indicated output position when the indicated language is `"*"` or `"java"`.

7.17.6 Group of Annotations: Interfaces

Table 7.11 identifies the mapping impact of the IDL defined Interface Annotations.

Table 7.11: Interface Annotation Impact

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

7.18

8 IDL to Java Language Mapping Annotations

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

8.1 @java_mapping Annotation

This annotation provides the means to customize the way a number of IDL constructs are mapped to the Java 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 **@java_mapping** annotation is:

```
@annotation java_mapping {
    string constants_container default "Constants";
    boolean promote_integer_width default FALSE;
    boolean apply_naming_convention default TRUE;
    string string_type default "java.lang.String";
}
```

The behavior associated with each parameter is defined below.

8.1.1 constants_container Parameter

constants_container defines the name of the Java class that holds the constants may be changed from its default value (i.e., **Constants**) to a user-defined value[7].

8.1.2 promote_integer_width Parameter

The lack of unsigned primitives in the Java language introduces a challenge when mapping the IDL **unsigned** integral types. For example, in order to support the full range of an IDL **unsigned short** which has a range of [0, 65535], it is mapped to the Java primitive int, with range [-2147483648, 2147483647], instead of the Java short which has a range of only [-32768, 32767].

promote_integer_width specifies whether IDL **unsigned** integers shall be mapped to a Java primitive type of the same size or to a bigger type capable of holding the full range of the corresponding **unsigned** integer. By default, as specified in clause 7.2.4.1.1, integer width is preserved (i.e., **promote_integer_width** is **FALSE**).

Table 8.1 shows the mapping of IDL integer types according to the value of **promote_integer_width**.

Table 8.1: Mapping of Integer Types According to promote_integer_width

IDL Type	Java Type	
	promote_integer_width = FALSE	promote_integer_width = TRUE
int8	byte	byte

7 If two distinct IDL files contain **const** definitions in the same **namespace**, then the mapping of each IDL file will output to the same *Constants.java* source file. The compiler is not required to perform a merge on the resulting **Constants** class. To extend the example, each IDL file could contain one or more incompatible **struct** definitions (same type name but different contents). This scenario is not detectable by the IDL compiler and would result in a similar collision during output; and therefore, such a scenario is considered to be out of scope. Further, if the IDL files cannot be restructured to remove the collision, then this scenario could be remedied by annotating one of the IDL files to select an alternate **constants_scope** name.

IDL Type	Java Type	
	`promote_integer_width = FALSE`	`promote_integer_width = TRUE`
`uint8`	`byte`	`short`
`short` `int16`	`short`	`short`
`unsigned short` `uint16`	`short`	`int`
`long` `int32`	`int`	`int`
`unsigned long` `uint32`	`int`	`long`
`long long` `int64`	`long`	`long`
`unsigned long long` `uint64`	`long`	`java.math.BigInteger`

8.1.3 apply_naming_convention Parameter

`apply_naming_convention` specifies whether the IDL to Java language mapping shall preserve the naming conventions of type identifier and member names in the IDL definition when mapping them to the corresponding Java construct or whether it shall adapt them to use Java naming conventions.

Table 8.2 shows the mapping of IDL member names and type identifiers according to the value of **`apply_naming_convention`**.

Table 8.2: Type Identifier and Member Name Mapping According to apply_naming_convention Value

IDL Construct	Java Mapping Naming Convention	
	`apply_naming_convention = TRUE`	`apply_naming_convention = FALSE`
Constant Variable Name	Name in All Uppercase Snake Case	Name as in IDL definition
Structure Type Name	Name in Pascal Case	Name as in IDL definition
Structure Member Name in Accesor/Modifier Methods	Name in Pascal Case	Name as in IDL definition
Structure Member Name in Modifier Method Parameter	Name in Camel Case	Name as in IDL definition
Union Type Name	Name in Pascal Case	Name as in IDL definition
Union Member Name in Accesor/Modifier Methods	Name in Pascal Case	Name as in IDL definition

IDL Construct	Java Mapping Naming Convention	
	`apply_naming_convention = TRUE`	`apply_naming_convention = FALSE`
Union Member Name in Modifier Method Parameter	Name in Camel Case	Name as in IDL definition
Enumeration Type Name	Name in Pascal Case	Name as in IDL definition
Interface Type Name	Name in Pascal Case	Name as in IDL definition
Interface Attribute Name in Accesor/Modifier Methods	Name in Pascal Case	Name as in IDL definition
Interface Attribute Name in Modifier Method Parameter	Name in Camel Case	Name as in IDL definition
Interface Method Name	Name in Camel Case	Name as in IDL definition
Interface Method Parameter Name	Name in Camel Case	Name as in IDL definition
Exception Type Name	Name in Pascal Case	Name as in IDL definition
Exception Member Name in Accesor/Modifier Methods	Name in Pascal Case	Name as in IDL definition
Bitset Type Name	Name in Pascal Case	Name as in IDL definition
Bitfield Name in Bitset Accesor/Modifier Methods	Name in Pascal Case	Name as in IDL definition
Bitfield Name in BitSet Modifier Method Parameter	Name in Camel Case	Name as in IDL definition
Bitmask Type Name	Name in Pascal Case	Name as in IDL definition

8.1.4 string_type Parameter

`string_type` defines the Java type IDL **`string`** and **`wstring`** types shall be mapped to. By default, as specified in clause 7.2.4.2.2 and 7.2.4.2.3, IDL **`string`** and **`wstring`** types are mapped to **`java.lang.String`** (i.e., **`string_type = "java.lang.String"`**).

Examples of alternative values for **`string_type`** may include **`"java.lang.StringBuilder"`** and **`"java.lang.StringBuffer"`**.

Annex A: Platform-Specific Mappings

(normative)

A.1 CORBA-Specific Mappings

This specification does not modify the existing IDL to Java mappings for CORBA defined in [I2JAV]. CORBA-specific building blocks shall therefore be mapped as specified in [I2JAV].

CORBA implementations may combine the language mappings defined in this specification for building blocks introduced in [IDL4] (e.g., for Annotations and Standardized Annotations) with the existing mappings defined in [I2JAV].

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 Java mappings defined in chapters 7 and 8 for the building blocks that compose the DDS profiles defined in clause 9.3 of [IDL4].

Annex B: Building Block Traceability Matrix

(non-normative)

The building block traceability matrix provides an indication of where (which document clause) each IDL building block is addressed in this language mapping.

Table B.1: Building Block Traceability Matrix

Building Block	Clause
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