OMG

OBJECT MANAGEMENT GROUP

Java 5 Language PSM for DDS

Version 1.0

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 *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/report_issue.htm).

Table of Contents

Preface

About the Object Management Group

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 Formal Specifications are available from this URL:

http://www.omg.org/spec

Specifications are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

- **CORBA/IIOP**
- **Data Distribution Services**
- **Specialized CORBA**

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

- **UML, MOF, CWM, XMI**
- **UML Profile**

Modernization Specifications

Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

- **CORBAServices**
- **CORBAFacilities**

OMG Domain Specifications

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

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 to *http://www.omg.org/report_issue.htm*.

1   Scope

This specification defines a platform-specific model (PSM) for the OMG Data Distribution Service for Real-Time Systems (DDS). It specifies an API only for the Data-Centric Publish-Subscribe (DCPS) portion of that specification; it does not address the Data Local Reconstruction Layer (DLRL). In addition, it encompasses (a) the DDS APIs introduced by [DDS-XTypes] and (b) an API to specifying QoS libraries and profiles such as were specified by [DDS-CCM].

This specification also defines a means of publishing and subscribing Java objects with DDS-the Java Type Representation-without first describing the types of those objects in another language, such as XML or OMG IDL.

2   Conformance

This specification consists of this document as well as a Java jar library file and the source files that generated it, identified on the cover page (all are normative). In the event of a conflict between them, the latter shall prevail.

Conformance to this specification parallels conformance to the DDS specification itself and consists of the same conformance levels. For example, an implementation may conform to the DDS Minimum Profile with respect to this PSM, meaning that all of the programming interfaces identified by the DDS specification as pertaining to that conformance level must be implemented in this PSM. The one exception to this rule is the Object Model Profile, which includes in part the Data Local Reconstruction Layer (DLRL); DLRL is outside of the scope of this PSM.

In addition to the conformance levels defined in the DDS specification itself, this PSM recognizes and implements the Extensible and Dynamic Types conformance level for DDS defined by the Extensible and Dynamic Topic Types for DDS specification [DDS-XTypes].

This PSM furthermore defines methods to create Entities and to set their QoS based on the XML QoS libraries and profiles defined by the DDS for Lightweight CCM specification. Implementations that support these XML QoS profiles shall implement these operations fully; other implementations shall throw java.lang.UnsupportedOperationException.

Finally, any conformant implementation must support at least one of the OMG-specified Type Representations defined by [DDS-XTypes] and/or in the Java Type Representation section of this specification (Clause 8).

3   References

3.1   Normative References

The following normative documents contain provisions that, 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.

- [DDS] Data Distribution Service for Real-Time Systems Specification, version 1.2 (OMG document formal/2007-01-01).

- [DDS-CCM] DDS for Lightweight CCM, version 1.0 Beta 1 (OMG document ptc/2009-02-02).

- [DDS-XTypes] Extensible and Dynamic Topic Types for DDS, version 1.0 Beta 1 (OMG document ptc/2010-05-12).

- [Java-MAP] IDL to Java Language Mapping, Version 1.3 (OMG document formal/2008-01-11)

- [Java-Lang] The Java Language Specification, Third Edition, published by Addison Wesley in 2005 with ISBN 0321246780

- [XML] Extensible Markup Language (XML), version 1.1, Second Edition (W3C recommendation, August 2006).

3.2 Non-Normative References

The following non-normative references are provided for informational purposes.

- [JMS] Java Message Service Specification, version 1.1 (Sun Microsystems, http://java.sun.com/products/jms/docs.html).

4 Terms and Definitions

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

Data-Centric Publish-Subscribe (DCPS)

The mandatory portion of the DDS specification used to provide the functionality required for an application to publish and subscribe to the values of data objects.

Data Distribution Service (DDS)

An OMG distributed data communications specification that allows Quality of Service policies to be specified for data timeliness and reliability. It is independent of implementation languages.

Data Local Reconstruction Layer

The optional portion of the DDS specification used to provide the functionality required for an application for direct access to data exchanged at the DCPS layer. This latter builds upon the DCPS layer.

Java Archive (JAR)

A zip file, whose name ends in the suffix .jar, that contains the compiled Java class files and other artifacts that comprise a Java library.

Java Runtime Environment (JRE)

The environment within which Java applications execute. The JRE consists of an executing instance of a JVM, a set of class libraries, and potentially other components.

Java Virtual Machine (JVM)

An abstract computing machine capable of executing interpreted and/or compiled Java byte code. JVM implementations typically take the form of executables that run as processes under operating systems, but this style of implementation is not mandatory.

Platform-Independent Model (PIM)

An abstract definition of a facility, often expressed with the aid of formal or semi-formal modeling languages such as OMG UML, that does not depend on any particular implementation technology.

Platform-Specific Model (PSM)

A concrete definition of a facility, typically based on a corresponding PIM, in which all implementation-specific dependencies have been resolved.

5   Symbols

This specification does not define any symbols or abbreviations.

6   Additional Information

6.1   Changes to Adopted OMG Specifications

This specification does not extend or modify any existing OMG specifications.

6.2   Relationships to Non-OMG Specifications

This specification depends on version 5 of the Java Standard Edition platform. Service implementations may impose additional constraints; the nature and scope of these, if any, are unspecified.

Design Rationale (non-normative)

As of the publication of this specification, Java SE remains the predominant platform for the development and deployment of DDS Java applications.

- Introducing a dependency on Java EE would have brought little additional capability to the PSM and would have put it outside of the reach of many potential users, especially those deploying to embedded operating systems, many of which do support Java EE.

- Allowing the PSM to support Java ME would have made it less usable for the majority of potential users. As of the publication of this specification, Java ME platforms lack support for many modern collections and Java language features. At the same time, support for Java ME would not have significantly increased the availability of implementations of this specification. Many embedded platforms already support Java SE, and existing DDS vendors have not observed significant customer demand for Java ME support in existing products.

6.3   Acknowledgements

The following companies submitted this specification:

- Real-Time Innovations, Inc. (RTI)

- PrismTech

7 Java 5 Language PSM for DDS

7.1 Specification Organization

This specification is organized according to the module defined by the DDS specification and the types and operations defined within them.

7.2 General Concerns and Conventions

This sub clause defines those elements of this specification that cut across multiple DDS modules.

7.2.1 Packages and Type Organization

This PSM is defined in a set of Java packages, the names of each beginning with the prefix `org.omg.dds`. Each of these contains a Java interface or abstract class for each type in the corresponding DDS module.

All of these packages, and the types within them, are packaged into a single JAR file, `omgdds.jar` (see Annex A: Java JAR Library File).

All those types that are abstract—including interfaces and abstract classes—are intended to be implemented concretely by the Service implementation. In addition, the subtypes defined by the implementation may expose additional implementation-specific properties and operations; however, the nature of these, if any, is undefined.

Design Rationale (non-normative)

This PSM divides the types it defines into multiple packages, rather than colocating them in a single package, for the following reasons:

- DDS defines a large number of types. Grouping them into multiple packages makes it clear which are more closely related to one another.
- The package organization improves traceability to the DDS PIM ([DDS]).
- The package organization parallels the namespace organization of the C++ PSM for DDS, facilitating cross-training across languages.

7.2.2 Implementation Coexistence

To facilitate the coexistence of multiple DDS implementations within the same JVM instance, each implementation of this PSM shall cooperate at the API level with other JVM-local implementations in at least the following ways:

- It shall be possible to pass an instance of any value type (see 7.2.3) created by one DDS implementation to a method implemented by another. For example, the method `DataWriter.write` optionally accepts an argument of type `InstanceHandle`; this object may have been created by the same DDS implementation that created the `DataWriter` or by another DDS implementation.
- It shall be possible to read or take samples from a `DataReader` provided by one DDS implementation and immediately write them using a `DataWriter` provided by another DDS implementation, provided that the samples are of a DDS type compatible with that `DataWriter`.

Note that passing an object from one implementation to another may incur a performance cost, as the "receiving" implementation may have to copy the object in question before operating on it.

Otherwise, unless elsewhere noted in this specification, a Service implementation may raise an exception or behave in an undefined way if it encounters a concrete type defined by another party. For example, the concrete `WaitSet` implementation provided by one DDS vendor need not support the attachment of `Condition` implementations provided by another DDS vendor.

7.2.3 Resource Management

The use of interfaces instead of classes requires the introduction of an explicit factory pattern for the construction of objects of all DDS types. For some types (Entities in particular), this pattern is already explicit in the DDS PIM. For other types (such as QoS policies), it is a property solely of the PIM-to-PSM mapping. These latter types—those without PIM-defined factory construction methods—serve as their own factories. Each is represented as an abstract class with one or more static factory methods. These methods are named according to the convention `new<ClassName>` in order to resemble constructor invocations and are amenable to use with the Java 5 static import facility.

This PSM maps the factory deletion methods of the DDS PIM (e.g., `DomainParticipant.delete_publisher`) to `close` methods on the "product" interfaces themselves (e.g., `Publisher.close`). Closing an Entity implicitly closes all of its contained objects, if any. For example, closing a `Publisher` also closes all of its contained `DataWriters`.

Design Rationale (non-normative)

> The `close` destruction design pattern is intended to be familiar to those developers who have used `java.io` stream APIs and/or [JMS] and eliminates the possibility that an object could be deleted using a factory other than the one that created it.

Users of this PSM are recommended to call `close` once they are finished using such heavyweight objects. In addition, implementations *may automatically* close objects that the JRE deems to be no longer in use—for example, they may call `close()` in an `Object.finalize()` override—subject to the following restrictions:

- Any object to which the application has a direct reference is still in use.

- Any entity with a non-null listener is still in use.

- Any object that has been explicitly retained is still in use.

- The creator of any object that is still in use is itself still in use.

7.2.4 Concurrency and Reentrancy

It is expected that most Service implementations will be used frequently in multi-threaded environments. Therefore, for the sake of portability, this PSM constrains the level of thread safety that applications may expect:

- All `DataReader` and `DataWriter` operations shall be reentrant.

- All `Topic` (and other `TopicDescription` extension interfaces), `Publisher`, `Subscriber`, and `DomainParticipant` operations shall be reentrant with the exception that `close` may not be called on a given object concurrently with any other call of any method on that object or on any contained object.

- All `ServiceEnvironment` and `DomainParticipantFactory` operations shall be reentrant with the exception that `DomainParticipantFactory.close` may not be called on a given object concurrently with any other call of any method on that object or on any contained object.

- All `WaitSet` and `Condition` (including `Condition` extension interfaces) operations shall be reentrant with the exception that their `close` methods may not be called on a given object concurrently with any other call of any method on that object.

- Code within a DDS listener callback may not safely call any method on any DDS Entity but the one on which the status change occurred.

- Any method of any value type may be non-reentrant.

A vendor may choose to provide stronger guarantees than the rules above, but if so, those guarantees are unspecified.

Design Rationale (non-normative)
- Objects that are likely to "own" mutexes within their implementation need not permit `close` invocations currently with other method invocations. This is to allow implementations to dispose of these mutexes within the `close` method without creating a race condition or requiring an additional level of locking.

- Method invocations are restricted within listener callbacks in order to avoid deadlocks, especially in Service implementations that invoke callbacks within Service-managed threads.

7.2.5 Method Signature Conventions

This PSM maps the underscore-formatted names of the DDS PIM and IDL PSM (such as `get_qos`) into conventional Java "camel-case" names (such as `getQos`). This mapping makes the API look more familiar to Java developers and makes it interoperate better with Java reflective technologies that expect this naming convention.

Properties defined by the DDS PIM are expressed as sets of accessor and mutator methods. The signatures of these methods conform to the following convention:

- Mutators are named `set<PropertyName>` (for example, the mutator for a property "Foo" would be named `setFoo`.) They take a single argument—the new value of the property—and return the enclosing object in order to facilitate method chaining.

- Accessors for properties that are either of unmodifiable objects (such as those of primitive types, primitive box types, or strings) or pointers to the internal state of an object are named `get<PropertyName>`. (For example, the accessor for an integer property "Foo" would be named `getFoo`.) They take no arguments.

- Accessors for properties that are of mutable types, and that may change asynchronously after they are retrieved, are named `get<PropertyName>`. They take a pre-allocated object of the property type as their first argument, the contents of which shall be overwritten by the method. To facilitate method chaining, these methods also return a reference to this argument. This pattern forces the caller to make a copy, thereby avoiding unexpected changes to the property. An Entity's status is an example of a property of this kind.

7.2.6 API Extensibility

Implementation-specific extensions to the types specified by this PSM are by definition unspecified. However, implementations may provide such a capability by providing extended implementation-specific interfaces and returning instances of these interfaces from the specified factory methods.

Implementations shall not place their extensions, if any, in any interface or class in the package `org.omg.dds` or in any other package whose name begins with that prefix.

7.3   Infrastructure Module

This PSM realizes the Infrastructure Module from the DDS specification with two packages: `org.omg.dds.core` and `org.omg.dds.core.policy`. The latter contains all QoS policy classes, since a given QoS policy may apply to multiple DDS Entity types. The former contains all other Infrastructure types, including for example `Entity` and `Condition` base interfaces.

Design Rationale (non-normative)

> These two packages have been made distinct from one another for two reasons: First, the QoS policies constitute a significant proportion of the total set of types in the Infrastructure Module, and the contents of the module are thus easier to understand when they are divided along this line. Second, a dedicated package for QoS policies makes the code completion features of modern programming environments easier to use, because it allows users to narrow the set of classes through which they must search in order to find the one they're looking for. The term "core" has been preferred to "infrastructure" for the sake of brevity (such as when using fully qualified names) and for consistency with the C++ PSM for DDS, which uses the term "core" as well.

7.3.1   `ServiceEnvironment` Class

A `ServiceEnvironment` object represents an instantiation of a Service implementation within a JVM. It is the "root" for all other DDS objects and assists in their creation by means of an internal service-provider interface. All stateful types in this PSM implement an interface `DDSObject`, through a `getServiceEnvironment` method on which they can provide access to the `ServiceEnvironment` from which they are ultimately derived. (`ServiceEnvironment` itself implements this interface; a `ServiceEnvironment` always returns `this` from its `getServiceEnvironment` operation.)

The `ServiceEnvironment` class allows implementations to avoid the presence of static state, if desired. It also allows multiple DDS implementations—or multiple versions of the "same" implementation—to potentially coexist within the same Java run-time environment. A DDS application's first step is to instantiate a `ServiceEnvironment`, which represents the DDS implementation that it will use. From there, it can create all of its additional DDS objects.

The `ServiceEnvironment` class is abstract. To avoid compile-time dependencies on concrete `ServiceEnvironment` implementations, an application can instantiate a `ServiceEnvironment` by means of a static `createInstance` method on the `ServiceEnvironment` class. This method looks up a concrete `ServiceEnvironment` subclass using a Java system property containing the name of that subclass. This subclass must be provided by implementers and will therefore have an implementation-specific name.

ServiceEnvironment provides factory methods for the following objects: *DynamicTypeFactory, WaitSet, GuardCondition, TypeSupport, Time, Duration*, and *InstanceHandle*. It also provides helper functions allStatuses and noStatuses to create special instances of Status objects.

7.3.2   Error Handling and Exceptions

The PSM maps the `ReturnCode_t` type from the DDS PIM into a combination of standard Java exceptions (where their semantics match those expressed in the PIM) and new exception classes defined by this PSM. This mapping is as follows:

- With the exception of `java.util.concurrent.TimeoutException`, all exceptions are unchecked (that is, they extend `java.lang.RuntimeException` directly or indirectly).

- The exception classes defined by this PSM extend the base class `DDSException`. All of the PSM-defined exception classes are defined in the package `org.omg.dds.core`. All of these classes are abstract so as not to specify the representation of state; implementations shall provide concrete implementations.

Table 7.1 - `ReturnCode_t` → exception mapping

`ReturnCode_t` Value	Exception Class
RETCODE_OK	Normal return; no exception.
RETCODE_NO_DATA	An informational state (e.g., a Boolean result) attached to a normal return; no exception.
RETCODE_ERROR	`DDSException`
RETCODE_BAD_PARAMETER	`java.lang.IllegalArgumentException`
RETCODE_TIMEOUT	`java.util.concurrent.TimeoutException`
RETCODE_UNSUPPORTED	`java.lang.UnsupportedOperationException`
RETCODE_ALREADY_DELETED	`AlreadyClosedException`
RETCODE_ILLEGAL_OPERATION	`IllegalOperationException`
RETCODE_NOT_ENABLED	`NotEnabledException`
RETCODE_PRECONDITION_NOT_MET	`PreconditionNotMetException`
RETCODE_IMMUTABLE_POLICY	`ImmutablePolicyException`
RETCODE_INCONSISTENT_POLICY	`InconsistentPolicyException`
RETCODE_OUT_OF_RESOURCES	`OutOfResourcesException`

In addition, this PSM permits implementations to throw exceptions to indicate errors in operations that in the PIM return an object reference. The PIM uses the convention of modeling failure conditions as operation return results, making it impossible to provide finer failure-detection granularity than a simple nil/non-nil result check in the case of methods that must return something other than a return code. The Java language, with built-in exception support, eliminates that restriction, and this PSM takes advantage of that fact.

Design Rationale (non-normative)
- This PSM uses checked and unchecked exceptions according to the following rationale: Where the exception represents a *fault*—a design flaw, implementation mistake, or runtime failure—it is unchecked. Where it represents a *contingency*—an uncommon-but-expected return scenario, for which the caller is expected to have a coping strategy—it is checked[1].
- Most exceptions in the DDS API represent faults, not contingencies.
- Within each category, this PSM reuses existing JRE exception classes when they are available and appropriate.

---

1. The fault/contingency model of Java exceptions was first described by Barry Ruzek, then of BEA, in late 2006 or early 2007 in the article *Effective Java Exceptions*. This article was originally published at http://dev2dev.bea.com/pub/a/2006/11/effective-exceptions.html and is now available at http://crmondemand.oracle.com/technetwork/articles/entarch/effective-exceptions-092345.html.

7.3.3 Value Types

All DDS types with value semantics implement the interface `org.omg.dds.core.Value`[2]. These include QoS, QoS policy, status, time, and other types.

The `Value` interface extends the standard Java SE interfaces `java.lang.Cloneable` and `java.io.Serializable`, allowing objects of implementing types to be copied by value as well as serialized and deserialized using built-in Java mechanisms.

It also defines a small number of additional methods. It defines a method `copyFrom` that accepts a source object of the same type as the object itself. This method overwrites the state of the target object ("`this`") with the state of the argument object. It is similar to `clone` but does not require allocating a new object. `Value` implementers are also expected to override their inherited implementations of `Object.equals` and `Object.hashCode` in order to enforce value semantics.

QoS policy objects are immutable. New policy objects can be created from existing policy objects by using the QoS DSL described in sub clause 7.3.5.3.

7.3.4 Time and Duration

This PSM maps the DDS `Time_t` and `Duration_t` types into the value types `Time` and `Duration` respectively. These classes can provide their magnitude using a variety of units (expressed using `java.util.concurrent.TimeUnit`).

Design Rationale (non-normative)

> The names of these types omit the underscore and 't' characters from the ends of their names. That naming convention, while common among C POSIX programmers, is not conventional in Java.

7.3.5 QoS and QoS Policies

The DDS PIM: individual QoS policies (such as reliability) and the collections of policies that apply to a particular DDS Entity type. This PSM represents the former with the base interface `org.omg.dds.core.policy.QosPolicy` and the latter with the base interface `org.omg.dds.core.EntityQos`.

7.3.5.1 QoS Policies

The DDS PIM represents each QoS policy in three ways; this PSM maps them as follows.

---

2. The term "value type" refers to any data type for which object identity is considered to be established solely based on the state of the objects of that type. Such types generally provide deep copy and comparison operations. (For example, integers are an example of a value type: every occurrence of the quantity 42 is considered to refer to the same number as every other.) The term should not be confused with an IDL `valuetype` as defined by the CORBA specification.

Table 7.2 - QoS Policy Representation

DDS PIM	Java 5 PSM
QoS policy structure containing the state of an instance of that policy.	QoS policy interface extending `org.omg.dds.core.policy.QosPolicy`. Each policy provides Java Bean-style properties.
Unique QoS policy ID, represented by an instance of the enumeration `QosPolicyId_t`.	Java platform provides "Class object," which uniquely identifies a QoS policy. The id will be represented by an object of `Class<? extends QosPolicy>` (for example, `Class<Reliability>`).
Unique QoS policy name, represented by a string property `QosPolicy.name`.	Java reflection provides the necessary capability to obtain name of a QoSPolicy class.

The `org.omg.dds.core.policy.PolicyFactory` interface allows creation of new default-initiated policy objects. The default state of the newly created policy objects via the `PolicyFactory` interface is unspecified.

Design Rationale (non-normative)

> In general, to ensure portability, programmers are encouraged to explicitly specify policy parameters using the QoS DSL.

7.3.5.2 Entity QoS

Each Entity QoS (e.g., `DataReaderQos`) is an interface extending `org.omg.dds.core.EntityQos`. These sub-interfaces provide direct access to their policies as in the IDL PSM. However, the base interface also provides for generic access using the `java.util.Map` interface. This interface allows applications to look up policies by ID and to iterate over them in a generic way, including vendor-specific extension policies, without introducing compile-time dependencies on vendor-specific APIs.

QoS objects cannot be created directly. They can be either retrieved from an entity (e.g., `DataReader`) using the `getQoS` method or looked up using a string identifier using the `QoSProvider` interface. QoS objects can be cloned from pre-existing state maintained by the Service implementation.

QoS objects as returned by Entities and QoSProvider shall be immutable; applications shall never observe them to change. Applications that wish to modify QoS values must use the "QoS DSL" supported by the QoS and Policy classes.

7.3.5.3 QoS DSL

Modifying QoS objects and their constituent policies is disallowed but a QoS DSL shall support creation of new QoS objects and policies from the existing objects using Java fluent interface design. QoS classes shall provide `withPolicy` and `withPolicies` methods that accept one or more policy objects to create a new QoS object. Policy classes shall provide *with* methods to specify policy parameters and to create new policy objects from the existing ones. Each *with* method call will create a new policy object because the target object of the method call is immutable. The *with* methods shall support method chaining (QoS DSL).

Example (non-normative)

PolicyFactory pf = … // object policy factory reference

ResourceLimits rl = pf.ResourceLimits().withMaxSamples(P).withMaxInstances(Q);

Design Rationale (non-normative)

- The `getQos` operation can operate maximally efficiently: it need not allocate any memory or perform any copies.

- The immutable result of `getQos` can be used safely concurrently from multiple threads.

- The `getQos` and `setQos` methods form a conventional Java-Bean-style property.

7.3.5.4   QoS Libraries and Profiles

The *DDS for Lightweight CCM* specification [DDS-CCM] defines a format for QoS libraries and profiles. These libraries and profiles provide a mechanism for entity QoS configuration administration. This PSM provides the following APIs for accessing these administered QoS configurations:

- The `org.omg.dds.core.QosProvider` interface allows Entity's Qos to be obtained from the names of QoS library and profile. The Qos library source is provided as a uniform resource identifier (URI). Conforming implementation must support "file://" prefix. For instance, "file:///path/to/qos/library."

  An instance of QosProvider is obtained from the ServiceEnvironment. For example,
  `serviceEnv.newQosProvider(String uri, String profile);`

  The uri parameter uses the standard uri syntax. The profile parameter identifies a uniquely identified profile in the document referred by the uri.

- Each Entity factory interface `DomainParticipantFactory`, `DomainParticipant`, `Publisher`, and `Subscriber` provides methods to create new "product" Entities and to set their default QoS based on QoS objects created programmatically or obtained through `QosProvider`.

7.3.6   Entity Base Interfaces

As in the DDS PIM, all Entity interfaces extend—directly or indirectly—the interface `Entity`. In this PSM, this interface is generic; it is parameterized by the Entity's QoS and listener types. These parameters allow applications to call common operations like `getQos` or `getListener` in a type-safe way while still working with Entities polymorphically. The `Entity` interface extends `java.io.Closeable` interface to support specific new language constructs (e.g., Java 7 try-with-resources) for dealing with all Entities polymorphically.

Also as in the DDS PIM, Entities other than `DomainParticipant` extend the interface `DomainEntity`. These Entities provide operations to get the creating parent Entity; in this PSM, this operation is the polymorphic `DomainEntity.getParent`.

7.3.7   Entity Status Changes

This sub clause describes the objects pertaining to the status changes of DDS Entities: the Status types themselves, listeners, conditions, and wait sets.

7.3.7.1 Status Classes

This PSM represents each status identified by the DDS PIM as an abstract class extending `org.omg.dds.core.Status`, which in turn extends `java.util.EventObject`.

The DDS PIM also identifies statuses using a "status kind;" these are composed into a mask that is used when setting listeners and at other times. This PSM represents status kinds using the `java.lang.Class` instances of the corresponding status classes and status masks as `java.util.Sets` of such status classes.

Status objects passed to listeners in callbacks may be pooled and reused by the implementation. Therefore, applications that wish to retain these objects—or any objects found within them, such as instance handles—for later use outside of the callback are responsible for copying them.

7.3.7.2 Listeners

This PSM maps the Listener interface from the DDS PIM to the empty marker interface `java.util.EventListener` interface defined by the Java SE standard library.

For each listener sub-interface (e.g., `DataWriterListener`), this PSM provides a concrete implementation of that interface in which all methods have empty implementations. These concrete classes are named like the listener interfaces they implement, but with the word "Listener" replaced by "Adapter."

In the DDS PIM, each listener callback receives two arguments: the Entity, the status of which has changed, and the new value of that status. In this PSM, the former is unnecessary and is omitted: it is available through the read-only `Source` property of the status object.

Design Rationale (non-normative)

> The listener + adapter design pattern is consistent with that used in the standard AWT and Swing UI libraries and elsewhere. It allows applications that are only interested in a subset of the callbacks provided by an interface to override only those methods and ignore the others.

This PSM distinguishes between lower-level listener interfaces, the implementations of which are likely to do type-specific things, and higher-level listener interfaces, the implementations of which are likely to do type-agnostic things.

- The former category includes `TopicListener`, `DataReaderListener`, and `DataWriterListener`. These classes are generic; their type parameters match that of the Entities on which they are set. This convention allows applications to read and write data within the context of a callback in a statically type-safe way.

- The latter category includes `PublisherListener`, `SubscriberListener`, and `DomainParticipantListener`. The `Topics`, `DataReaders`, and `DataWriters` passed to these listeners' callbacks are parameterized with the generic wildcard '?'. Because of this difference between these listeners and those in the former category, there are no inheritance relationships between these categories, unlike in the PIM.

7.3.7.3 Conditions

Conditions extend the base interface `org.omg.dds.core.Condition`.

The interface `StatusCondition`, which extends `Condition`, is a generic interface with a type parameter that is the type of the Entity to which it belongs. This type parameter allows its `getParent` method to be both polymorphic and type safe.

7.3.7.4 Wait Sets

Wait sets extend the base interface `org.omg.dds.core.WaitSet`.

In the DDS PIM, an application indicates its intention to wait for a condition to be triggered by invoking the operation `WaitSet.wait`. However, in Java, this operation overloads unintentionally with the inherited method `Object.wait`. This inherited method has a different meaning; the overload is inappropriate. Therefore, this PSM maps the DDS PIM `wait` operation to the more explicit method name `waitForConditions`.

7.4 Domain Module

This PSM realizes the Domain Module from the DDS specification with the package `org.omg.dds.domain`. This package contains `DomainParticipant`, `DomainParticipantFactory`, and so forth.

7.4.1 `DomainParticipantFactory` Interface

The `DomainParticipantFactory` is a per-`ServiceEnvironment` singleton. An instance of this interface can be obtained by passing that `ServiceEnvironment` to the factory's `getInstance` method.

7.4.2 `DomainParticipant` Interface

This PSM represents the `DomainParticipant` classifier from the DDS PIM with the interface `org.omg.dds.domain.DomainParticipant`.

7.5 Topic Module

This PSM realizes the Topic Module from the DDS specification with the packages `org.omg.dds.type` and `org.omg.dds.topic`.

7.5.1 Type Support

As in the DDS PIM, each type to be published or subscribed with DDS is represented by a class extending `org.omg.dds.type.TypeSupport`. Applications obtain instances of these interfaces by calling the static base class operation `newTypeSupport`, passing this method the Java `Class` object of the type they wish to support and optionally a name under which that type should be registered. If no such name is provided, the type shall be registered under the fully qualified name of the provided `Class`.

This PSM modifies slightly the capability for type registration provided by the DDS PIM. In the PIM, types are registered by invoking a `TypeSupport.register_type` operation. Subsequently, applications provide the registered type name to the `DomainParticipant.create_topic` operation in order to refer to the registered type. This PSM instead asks applications to instantiate each `TypeSupport` object with a name and then provide that `TypeSupport` itself to the `create_topic` method.

Design Rationale (non-normative)

> By requiring the application to pass an instance of the generic `TypeSupport` interface to the `createTopic` method, this PSM maintains unbroken static type safety all the way from type registration to data publication or reception. A pattern of type access based on the name strings would require a type cast.

7.5.2 `Topic` Interface

The `Topic` interface adds only a single operation to the set of those it inherits from its `TopicDescription` and `DomainEntity` super-types: an accessor for the inconsistent topic status. `Topic`—like all `TopicDescriptions`, and like `DataReader` and `DataWriter`—is a generic interface with a type parameter that identifies the type of the

data with which it is associated. Although `Topic` provides no type-specific operations, its type parameter preserves type safety from `Topic` creation (actually all the way from type registration) through data publication and/or subscription. `TopicDescription` interface extends `java.io.Closeable` to support specific new language constructs (e.g., Java try-with-resources) for dealing with this interface.

7.5.3 `ContentFilteredTopic` and `MultiTopic` Interfaces

`ContentFilteredTopic` and `MultiTopic` are generic interfaces with type parameters that identify the types of the data with which they are associated.

Note that the type parameter of a `ContentFilteredTopic` does not need to match that of its related `Topic` exactly; it can be any supertype. For example, if the user-defined type `Bar` extends the user-defined type `Foo`, a `ContentFilteredTopic<Foo>` can wrap a `Topic<Bar>`.

7.5.4 Discovery Interfaces

The data types pertaining to the DDS built-in discovery topics are contained in the package `org.omg.dds.topic` as well. These types provide only accessors for their state, not mutators, to reflect the read-only (from an application's point of view) nature of discovery.

7.6 Publication Module

This PSM realizes the Publication Module from the DDS specification with the package `org.omg.dds.pub`.

Design Rationale (non-normative)
> The term "pub" has been preferred to the longer "publication" for the sake of brevity (such as when using fully qualified names) and for consistency with the C++ PSM for DDS, which uses the term "pub" as well.

7.6.1 `Publisher` Interface

`Publishers` are represented by instances of the `org.omg.dds.pub.Publisher` interface.

In addition to the methods defined for this interface by [DDS], it additionally provides a `lookupDataWriter` overload that acts on the basis of a `Topic` object rather than solely on the topic's name. This overload is provided for the sake of additional static type safety.

7.6.2 `DataWriter` Interface

`DataWriters` are represented by instances of the `org.omg.dds.pub.DataWriter` interface. This is a generic interface, parameterized by the type of the data samples to be written by a given writer. The DDS PIM distinguishes between a type-specific `DataWriter` (`FooDataWriter`) and one whose type is not statically known (`DataWriter` itself); these are related by an inheritance relationship. This PSM makes no such distinction: Java's generic wildcard syntax (`DataWriter<?>`) makes it possible to express all type-specific `DataWriter` operations on the `DataWriter` interface itself; there is no `FooDataWriter`.

For most type-specific operations, the DDS PIM provides variants that accept an explicit timestamp (to allow applications to manage the passage of time themselves) and variants that do not (indicating that the Service implementation should provide this); these two sets of operations use different naming conventions. In addition, the PIM includes an instance handle parameter in the signatures of these operations, despite the fact that not all types are keyed and therefore have any use for instance handles. These design choices reflect the existence of the IDL PSM: IDL does not support method

overloading. Java does; therefore, the provision of timestamps and/or instance handles is optional and is handled by means of method overloads. For example, the `write` method provides the following overloads: one accepting a data sample only, another accepting a sample and an instance handle, and another accepting both of these as well as a timestamp. Users of `DataWriters` of unkeyed types may choose to call the overloads that accept instance handle arguments; if they do, the handle argument must be a nil handle (as explained in the DDS PIM).

7.7 Subscription Module

This PSM realizes the Subscription Module from the DDS specification with the package `org.omg.dds.sub`.

<u>Design Rationale (non-normative)</u>

> The term "sub" has been preferred to the longer "subscription" for the sake of brevity (such as when using fully qualified names) and for consistency with the C++ PSM for DDS, which uses the term "sub" as well.

7.7.1 `Subscriber` Interface

`Subscribers` are represented by instances of the `org.omg.dds.sub.Subscriber` interface.

In addition to the methods defined for this interface by [DDS], it additionally provides a `lookupDataReader` overload that acts on the basis of a `TopicDescription` object rather than solely on the topic description's name. This overload is provided for the sake of additional static type safety.

7.7.2 `Sample` Interface

This PSM follows the guidance of the DDS PIM rather than of the IDL PSM: it represents data samples as single objects that incorporate both data and metadata. Each sample is represented by an instance of the `org.omg.dds.sub.Sample` interface. It provides its data via a `getData` method; if there is no valid data (corresponding to a false value for `SampleInfo.valid_data` in the IDL PSM), this operation returns `null`. It provides its metadata (corresponding to the other `SampleInfo` properties in the IDL PSM) as read-only Java-Bean-style properties.

The `Sample` interface also defines a nested interface: `Sample.Iterator`, an iterator that extends `java.util.ListIterator`. An iterator of this type provides read-only access to an ordered series of samples of a single type; such iterators are used by the `DataReader` `read` and `take` methods (see below).

7.7.3 `DataReader` Interface

`DataReaders` are represented by instances of the `org.omg.dds.sub.DataReader` interface. This is a generic interface, parameterized by the type of the data samples to be read by a given reader. The DDS PIM distinguishes between a type-specific `DataReader` (`FooDataReader`) and one whose type is not statically known (`DataReader` itself); these are related by an inheritance relationship. This PSM makes no such distinction: Java's generic wildcard syntax (`DataReader<?>`) makes it possible to express all type-specific `DataReader` operations on the `DataReader` interface itself; there is no `FooDataReader`.

The `DataReader` interface provides a number of `read` and `take` method overloads. In addition to the distinction between read vs. take semantics (as defined in the DDS PIM), these operations come in two "flavors." One that loans samples from a Service pool and returns a `Sample.Iterator` and another that deeply copies into an application-provided `java.util.List`, as follows:

- Applications that read or take loans must eventually return those loans; this PSM maps the `return_loan` operation from the DDS PIM to an operation `returnLoan` on the `Sample.Iterator`. Moreover, the iterator implements the Java.io.Closeable interface so that try-with-resources construct can be used in Java 7.

- Applications that read or take copies may provide to the Service destination `Lists` with any number of `Samples` already in them (including empty `Lists`). Regardless of the number of `Samples` already in the list when the method is called, when it returns, the `List` shall contain the number of `Samples` requested by the application (or fewer, if fewer were available). The Service implementation may—for example, in order to avoid object allocations—elect to overwrite the contents of any `Samples` that are passed into it by invocations of these methods.

The `read` and `take` operations defined by the DDS PIM do not take advantage of overloading, because they were designed with the IDL PSM in mind, and IDL does not support overloading. Java does; therefore, this PSM both simplifies the operations' signatures as well as captures commonalities among them as follows:

- Instead of overloading several operation variants that accept large numbers of infrequently used parameters, a `DataReader.Selector` is provided to encapsulate various selection criteria (for example, sets of sample, instance, and view states). `DataReader.select` method returns a `Selector` object, which encapsulates the *default* selection criteria. For portability, the default state of the Selector object is defined as `instanceHandle=null`, `nextInstance=false`, `dataState=any`, `queryExpression=null`, and `maxSamples=unlimited`. Selector provides fluent interface to modify the default selection parameters. For convenience, Selector provides `read` and `take` methods.

7.8   Extensible and Dynamic Topic Types Module

This sub clause addresses those additions to DDS introduced by the *Extensible and Dynamic Topic Types for DDS* specification [DDS-XTypes]. The additions fall into the following categories:

- **Types pertaining to `TypeObject` Type Representations** are defined in the package `org.omg.dds.type.typeobject`.

- **Types pertaining to the Dynamic Language Binding** are defined in the package `org.omg.dds.type.dynamic`.

- **The `TypeKind` enumeration**, which pertains to both of the above, is defined in the package `org.omg.dds.type`.

- **The built-in types** are defined in the package `org.omg.dds.type.builtin`.

- **Extensions by [DDS-XTypes] to types defined by [DDS]** (such as the built-in topic data types) are contained within those types.

7.8.1   Dynamic Language Binding

The Dynamic Language Binding, as defined by [DDS-XTypes], consists of `DynamicType`, `DynamicTypeMember`, `DynamicData`, their respective factories, and several "descriptor" value types.

7.8.1.1   `DynamicTypeFactory` Interface

This abstract factory is a per-`ServiceEnvironment` singleton. The static `delete_instance` operations defined in [DDS-XTypes] have been omitted in this PSM; the Service shall manage the life cycles of the factory.

7.8.1.2   `DynamicTypeSupport` Interface

The interface `DynamicTypeSupport` defined by [DDS-XTypes] does not provide any capability beyond what the generic `TypeSupport` interface provided by this PSM already provides. Therefore, it has been omitted from this PSM.

7.8.1.3 `DynamicType` and `DynamicTypeMember` Interfaces

These interfaces are expressed in this PSM according to the mapping rules expressed elsewhere in this document. In addition, the following changes to this mapping have been made:

- Operations that provide their result as an in-out value in their first parameter and return `DDS::ReturnCode_t` have been changed such that they instead return their results directly. (This change, made for the convenience of the caller, is possible because `DDS::ReturnCode_t` is mapped to a set of exceptions in this PSM.)

- The `equals` and `clone` operations on these types have been mapped to overrides of the Java-standard `Object.equals` and `Object.clone`, respectively.

- `DynamicTypeMember` is a reference type, and instances of it are obtained from `DynamicType.addMember`. This change avoids the need to provide an additional factory method for `DynamicTypeMember` instances.

- On each type, the operations `get_annotation_count` and `get_annotation` (by index) have been unified into a single `getAnnotations` method that returns a list of annotations. The lists returned from these methods shall not be modifiable.

In addition to the methods specified by [DDS-XTypes], `DynamicTypeFactory` provides one additional factory method: `createType(Class<?>)`. This method shall inspect the given type reflectively in accordance with the Java Type Representation (see Clause 8) and instantiate an equivalent `DynamicType` object.

7.8.1.4 `DynamicData` Interface

This interface is expressed in this PSM according to the mapping rules expressed elsewhere in this document. In addition, the following changes to this mapping have been made:

- Operations that provide their result as an in-out value in their first parameter and return `DDS::ReturnCode_t` have been changed such that they instead return their results directly. (This change, made for the convenience of the caller, is possible because `DDS::ReturnCode_t` is mapped to a set of exceptions in this PSM.)

- The `equals` and `clone` operations on these types have been mapped to overrides of the Java-standard `Object.equals` and `Object.clone`, respectively.

- Methods dealing with unsigned integer types have been omitted. Applications may access unsigned data using the signed type of the same size (e.g., UInt32 becomes `Int32`), which preserves bitwise representation but not logical value, or by using the signed type one size up (e.g., UInt32 becomes `Int64`), which preserves logical value but not representation (and may therefore require additional range checking by the implementation). In the case of UInt64, the "type one size up" is `java.math.BigInteger`.

- The 128-bit `Float128` type has been represented using `java.math.BigDecimal`.

7.8.1.5 Descriptor Interfaces

This specification defines three descriptor interfaces. The instances of descriptor interfaces are immutable and therefore, provide methods to create new descriptor objects from the existing ones.

- `AnnotationDescriptor`

- `MemberDescriptor`

- `TypeDescriptor`

7.8.2 Built-in Types

[DDS-XTypes] specifies four built-in types: `DDS::String`, `DDS::Bytes`, `DDS::KeyedString`, and `DDS::KeyedBytes`.

- `DDS::String` is mapped to `java.lang.String`.

- `DDS::Bytes` is mapped to `byte[]`.

- `DDS::KeyedString` and `DDS::KeyedBytes` are mapped to modifiable value type interfaces.

The generic `DataReader` and `DataWriter` interfaces are applicable to the built-in types. `Subscriber` and `Publisher` provide generic `createDataReader` and `createDataWriter` methods to create datareader and datawriter for the built-in types, respectively.

7.8.3 Representing Types with `TypeObject`

The types in this package are expressed as modifiable value types according to the mapping rules expressed elsewhere in this document. In addition, the following changes to this mapping have been made:

- Top-level constants are moved into related interfaces, for example: `Member.MEMBER_ID_INVALID`.

Enumerations of member ID values are nested final classes within the interfaces for which they provide the member's IDs. These classes have constant integer fields, for example: `MapType.MemberId.BOUND_MAPTYPE_MEMBER_ID`.

8 Java Type Representation and Language Binding

8.1 General

The Java Type Representation defined in this clause provides a means for Java developers to publish and subscribe to DDS topics typed by plain Java objects without resorting to code generation or the reflective style of the Dynamic Language Binding.

By its very nature as an expression of the Java programming language, this Type Representation implicitly and simultaneously defines a Language Binding for DDS types. That is, a Java type necessarily defines a Java API to itself as part of its definition. Therefore, this Type Representation is intended for the run-time use of implementations of this PSM. While this specification does not preclude Service implementations from using this Type Representation for other purposes (for example, generating a Plain Language Binding in C for a DDS type represented in Java) such uses are non-normative and unspecified.

The Java platform provides a mechanism by which Java type definitions can be used to define how objects can be serialized for transmission over a network: the java.io.Serializable interface and its related types. Since the transmission of data from Java programs over DDS is a related problem, this specification builds on that mechanism. Any Java type that implements Serializable (directly or indirectly) shall be available for publishing and/or subscribing over DDS as defined below. Note that the DDS serialization of a type will not generally be the same as the JRE serialization of the same type, even if the type designer's specification of which data to serialize can be shared between these two mechanisms.

8.2 Default Mappings

The following table defines the default mappings from Java type system definitions to DDS type system ones.

Table 8.1 - Default type mappings

Java Type	DDS Type
INT, JAVA.LANG.INTEGER	INT32
SHORT, JAVA.LANG.SHORT	INT16
LONG, JAVA.LANG.LONG	INT64
FLOAT, JAVA.LANG.FLOAT	FLOAT32
DOUBLE, JAVA.LANG.DOUBLE	FLOAT64
CHAR, JAVA.LANG.CHARACTER	CHAR8
BYTE, JAVA.LANG.BYTE	BYTE
BOOLEAN, JAVA.LANG.BOOLEAN	BOOLEAN
JAVA.LANG.STRING	STRING<CHAR8>
java.util.Map	map
java.lang.Collection, array	sequence
java.lang.Object	Structure

A type designer may modify these defaults on a type-by-type and/or field-by-field basis by applying the annotation `org.omg.dds.type.SerializeAs`:

```
public @interface SerializeAs {
    public TypeKind value();
    …
}
```

8.3   Metadata

The type system metadata represented with built-in annotations in the IDL Type Representation (such as `@Key`, `@ID`) shall be represented by equivalent Java annotations unless otherwise noted. These annotations are in the package `org.omg.dds.type`.

The annotations in this package logically govern the behavior of concrete classes, not of polymorphic interfaces. As such, they may be applied to classes or to their fields, as appropriate. Interface designers wishing to document the DDS serialization of a type may additionally apply them to interfaces or to property accessor and/or mutator methods; however, they have no specified behavior in such cases.

8.4   Primitive Types

By default, Java primitive types are mapped to DDS primitive types as defined in Table 8.2. The `@SerializeAs` annotation may be used to modify these mappings as follows.

Table 8.2 - Customized primitive type mappings

DDS Type	Permitted Java Primitive Types
Int32	int, java.lang.Integer
UInt32	int, long, java.lang.Integer, java.lang.Long
Int16	short, java.lang.Short
UInt16	short, int, java.lang.Short, java.lang.Integer
Int64	long, java.lang.Long
UInt64	long, java.lang.Long, java.math.BigInteger
Float32	float, java.lang.Float
Float64	double, java.lang.Double
Float128	double, java.lang.Double, java.math.BigDecimal
Byte	byte, java.lang.Byte
Boolean	boolean, java.lang.Boolean
Char8	char, java.lang.Character
Char32	char, int, java.lang.Character, java.lang.Integer

The DDS Type System ([DDS-XTypes]) defines unsigned integer types; the Java type system does not. As a result, this Type Representation must map unsigned values to "equivalent" signed types. Type designers have two choices, reflected in the table above:

- **Preserve representation**: Map the DDS unsigned type to a Java signed type of the same size. Designers can be confident that every value in the range of the DDS type has an equivalent value in the range of the Java type. However, logical values will not be preserved in all cases: for example, large unsigned (positive) values will appear as negative values to Java applications.

- **Preserve logical value**: Map the DDS unsigned type to the next-larger Java signed type such that all values in the range of the DDS type can be reflected faithfully in the range of the Java type. However, applications must be prepared to deal with failures that may occur when data values that are logically unsigned mistakenly take a negative value that cannot be faithfully represented on the DDS network.

8.5 Collections

[DDS-XTypes] recognizes three categories of collections: strings (variable-length lists of narrow or wide characters), sequences (variable-length lists of any single element type), and maps (homogeneously typed key-value mappings).

8.5.1 Strings

DDS strings, whether of narrow or wide characters, are represented by Java `String` objects.

- If a string is to be of narrow characters (the default), each Java character shall be truncated to its least-significant byte.

- If a string is to be of wide characters (in which case it must be so marked with `@SerializeAs`), each Java code point shall become a single DDS wide character.

8.5.2 Maps

Any object whose class implements the interface `java.util.Map` shall be considered a DDS map unless marked otherwise with `@SerializeAs`.

8.5.3 Sequences and Arrays

Any object whose class implements the interface `java.util.Collection` shall be considered DDS sequences unless marked otherwise with `@SerializeAs`. If the class implements `java.util.List`, the order of the elements in the sequence shall correspond exactly to the order of the elements in the list. Otherwise, the order of the elements in the sequence shall correspond to that returned by the collection's iterator.

Objects of array types shall be considered DDS sequences unless marked otherwise with `@SerializeAs`.

Any Java collection or array may be designated as a DDS array with `@SerializeAs`.

Design Rationale (non-normative)

> Objects of array types must receive special care, because a Java array—like any Java object—is stored by reference only. Therefore, although a given array object itself is not of variable length, the reference to it may be reassigned to point to an array of a different length. Even if the reference does not change, the length of the array pointed to cannot in general be discovered by analysis of the type itself and may vary from object to object of the same type.

8.6   Aggregated Types

[DDS-XTypes] recognizes two kinds of aggregated types: structures and unions.

Any DDS type that is not a nested type (in the sense of that word defined by [DDS-XTypes], as indicated in this Type Representation by the annotation `@Nested`) must define a no-argument constructor for use by the Service implementation. Service implementations shall have the capability to invoke this constructor reflectively, even if it is not public.

The fields in the DDS structured type shall correspond to those of the Java class. Their order shall be that returned by the method `java.lang.reflect.Class.getDeclaredFields`. `Static` and/or `transient` fields shall be omitted. Service implementations shall have the capability to get and set the values of fields reflectively regardless of their declared access level (e.g., `public`, `protected`, `private`).

Service implementations need not address the following cases:

- A Java Security Manager (`java.lang.SecurityManager`) prevents privileged access to a non-`public` field or constructor.

- A field that is neither `static` nor `transient` is declared `final`, preventing its value from being modified.

- Object references form a cycle. (Cycles are not permitted by the DDS Type System.)

8.6.1   Structures

Every Java class that is not a collection or map shall be considered a structure by default.

8.6.1.1   Inheritance

Java class extension shall map to structure inheritance in the DDS Type System [DDS-XTypes], subject to the restrictions documented by the `java.io.Serializable` interface, such as those pertaining to non-`Serializable` base types.

8.6.1.2   Extensibility

The extensibility kind shall be determined in the following manner:

- `FINAL`: If the class extends `java.lang.Object` directly and is final, or if explicitly indicated.

- `EXTENSIBLE`: In all other cases, by default, or if explicitly indicated.

- `MUTABLE`: Only if explicitly indicated.

8.6.2   Unions

Any class may be annotated as a union with `@SerializeAs`.

- Such a class must annotate exactly one field to be the discriminator with `@UnionDiscriminator`.

- All other fields that are not `transient` or `static` must be annotated with `@UnionMember`, which shall identify the discriminator value associated with that field.

8.7 Enumerations and Bit Sets

By default, any Java enumeration class will be considered to be a DDS enumeration.

As in IDL, a type that is syntactically an enumeration may be annotated as a bit set type. In this case, objects of these types must also be annotated in order to be serialized correctly. A type member of type `java.util.EnumSet` or `java.util.BitSet` will be serialized as a bit set if marked with `@BitSet`.

8.8 Modules

Each segment of a Java type's package name shall correspond to a module in the DDS Type System [DDS-XTypes]. For example, a class `com.acme.project.TheClass` would be in the nested modules `com::acme::project`.

8.9 Annotations

This Type Representation ignores Java annotation types by default. Java annotations that are intended to be represented explicitly within the DDS Type System must be so annotated with `@SerializeAs`.

9 Improved Plain Language Binding for Java

9.1 Type Mapping

The type system for DDS topic types is defined by the Extensible and Dynamic Topic Types for DDS specification [DDS-XTypes].

This sub clause defines the set of rules to be used in order to map abstract DDS topic types into Java types that can be used by application programmers. Those aspects of the DDS Type System that are not addressed below are as specified in the Plain Language Binding as defined by [DDS- XTypes] (which in turn is defined in terms of an IDL-to-Java mapping [Java-MAP]).

9.1.1 Mapping Aggregation Types

DDS aggregation types shall be mapped to a final Java class. Contained attributes shall be encapsulated. Java Bean style accessors shall be provided. Special mapping rules for boolean properties are allowed. The representation of internal state shall be private.

9.1.2 Mapping Sequences and Arrays

Unbounded DDS sequences are mapped to Collection<E> interface. The state is encapsulated and getters/setters are provided through bean style property accessors. Bounded sequences and arrays are mapped to Java arrays.

9.2 Example (non-normative)

IDL	Java Representation

```
struct Point {
  long x, y;
  long z; //@optional
};


typedef sequence<octet>   plot_t;


struct RadarTrack {
    string id;
    string name;   //@optional
    Point center;
    Point vicinity[8];
    plot_t plot;   //@shared
};
```

```
public final class Point {
  Point();
  Point(int x, int y,
        java.lang.Integer z);


  int getX();
  void setX(int32 v);


  int32 getY();
  void setY(int32 v);


  java.lang.Integer getZ();
  void setZ(java.lang.Integer v);
};


public class final RadarTrack
{
  RadarTrack();
  RadarTrack(String id,
             String name,
             Point center,
             Point[] vicinity,

  java.util.Collection<byte> plot);
```

IDL	**Java Representation**
| | ```
String getId();

void setId(String id);


String getName();

void setName(String name);


Point getCenter();

void setCenter(Point center);


Point[] getVicinity();

void setVicinity(Point[] vicinity);


java.util.Collection<byte> getPlot();


void setPlot(java.util.Collection<byte>
plot);

};
``` |

Annex A
Java JAR Library File

In addition to this document, this specification includes a Java Archive (JAR) library, omgdds.jar. This library contains compiled Java *.class files for all of the classes and interfaces specified by this PSM.

This library comprises the compile-time portion of this specification: users shall be able to compile their PSM-compliant code against this library and then deploy the result against any conformant implementation.

Distributors of binary implementations of this PSM may elect to distribute the omgdds library alongside their implementation libraries or to package both the contents of omgdds.jar and their implementation into a single library.

Annex B
Java Source Code

In addition to this document, this specification includes the Java source code to all of the classes and interfaces specified by this PSM in the zip archive omgdds_src.zip. This source code, in the directory srcJava within the archive, corresponds to the binary distribution found in the library omgdds.jar and is also normative with respect to both its programming interfaces and its embedded documentation comments. (The latter have been transformed into JavaDoc HTML documentation, which is available in the zip file within the doc/ directory.)

For the convenience of both implementers and application developers, the archive contains additional files that are neither API source code nor documentation. These file are non-normative and include:

- Code examples: Short code segments, intended to be illustrative to application developers, can be found in the directory srcJavaExample within the archive.

- build.xml: A build script, compatible with version 1.6 of the Apache Ant tool[1], can be found in the top-level directory of the archive. It is capable of creating both the omgdds.jar and omg_src.zip files.

- Project files: Project definition files compatible with version 3.5 of the Eclipse IDE for Java can be found in the top-level directory of the archive.

---

1. See http://ant.apache.org