Java Object Serialization Specification
version 6.0

System Architecture

CHAPTER 1

Topics:

- Overview
- Writing to an Object Stream
- Reading from an Object Stream
- Object Streams as Containers
- Defining Serializable Fields for a Class
- Documenting Serializable Fields and Data for a Class
- Accessing Serializable Fields of a Class
- The ObjectOutput Interface
- The ObjectInput Interface
- The Serializable Interface
- The Externalizable Interface
- Serialization of Enum Constants
- Protecting Sensitive Information

1.1 Overview

The ability to store and retrieve Java^{TM} objects is essential to building all but the most transient applications. The key to storing and retrieving objects in a serialized form is representing the state of objects sufficient to reconstruct the object(s). Objects to be saved in the stream may support either the `Serializable` or the `Externalizable` interface. For Java^{TM} objects, the serialized form must be able to identify and verify the Java^{TM} class from which the contents of the object were saved and to restore the contents to a new instance. For serializable objects, the stream includes sufficient information to restore the fields in the stream to a compatible version of the class. For Externalizable objects, the class is solely responsible for the external format of its contents.

Objects to be stored and retrieved frequently refer to other objects. Those other objects must be stored and retrieved at the same time to maintain the relationships between the objects. When an object is stored, all of the objects that are reachable from that object are stored as well.

The goals for serializing Java^{TM} objects are to:

- Have a simple yet extensible mechanism.
- Maintain the Java^{TM} object type and safety properties in the serialized form.
- Be extensible to support marshaling and unmarshaling as needed for remote objects.
- Be extensible to support simple persistence of Java^{TM} objects.

- Require per class implementation only for customization.
- Allow the object to define its external format.

1.2 Writing to an Object Stream

Writing objects and primitives to a stream is a straightforward process. For example:

```
// Serialize today's date to a file.
    FileOutputStream f = new FileOutputStream("tmp");
    ObjectOutput s = new ObjectOutputStream(f);
    s.writeObject("Today");
    s.writeObject(new Date());
    s.flush();
```

First an `OutputStream`, in this case a `FileOutputStream`, is needed to receive the bytes. Then an `ObjectOutputStream` is created that writes to the `FileOutputStream`. Next, the string "Today" and a Date object are written to the stream. More generally, objects are written with the `writeObject` method and primitives are written to the stream with the methods of `DataOutput`.

The `writeObject` method (see Section 2.3, "The writeObject Method") serializes the specified object and traverses its references to other objects in the object graph recursively to create a complete serialized representation of the graph. Within a stream, the first reference to any object results in the object being serialized or externalized and the assignment of a handle for that object. Subsequent references to that object are encoded as the handle. Using object handles preserves sharing and circular references that occur naturally in object graphs. Subsequent references to an object use only the handle allowing a very compact representation.

Special handling is required for arrays, enum constants, and objects of type `Class`, `ObjectStreamClass`, and `String`. Other objects must implement either the `Serializable` or the `Externalizable` interface to be saved in or restored from a stream.

Primitive data types are written to the stream with the methods in the `DataOutput` interface, such as `writeInt`, `writeFloat`, or `writeUTF`. Individual bytes and arrays of bytes are written with the methods of `OutputStream`. Except for serializable fields, primitive data is written to the stream in block-data records, with each record prefixed by a marker and an indication of the number of bytes in the record.

`ObjectOutputStream` can be extended to customize the information about classes in the stream or to replace objects to be serialized. Refer to the `annotateClass` and `replaceObject` method descriptions for details.

1.3 Reading from an Object Stream

Reading an object from a stream, like writing, is straightforward:

```
// Deserialize a string and date from a file.
    FileInputStream in = new FileInputStream("tmp");
    ObjectInputStream s = new ObjectInputStream(in);
    String today = (String)s.readObject();
    Date date = (Date)s.readObject();
```

First an `InputStream`, in this case a `FileInputStream`, is needed as the source stream. Then an `ObjectInputStream` is created that reads from the `InputStream`. Next, the string "Today" and a Date object are read from the stream. Generally, objects are read with the `readObject` method and primitives are read from the stream with the methods of `DataInput`.

The `readObject` method deserializes the next object in the stream and traverses its references to other objects recursively to create the complete graph of objects serialized.

Primitive data types are read from the stream with the methods in the `DataInput` interface, such as `readInt`, `readFloat`, or `readUTF`. Individual bytes and arrays of bytes are read with the methods of `InputStream`. Except for serializable fields, primitive data is read from block-data records.

`ObjectInputStream` can be extended to utilize customized information in the stream about classes or to replace objects that have been deserialized. Refer to the `resolveClass` and `resolveObject` method descriptions for details.

1.4 Object Streams as Containers

Object Serialization produces and consumes a stream of bytes that contain one or more primitives and objects. The objects written to the stream, in turn, refer to other objects, which are also represented in the stream. Object Serialization produces just one stream format that encodes and stores the contained objects.

Each object that acts as a container implements an interface which allows primitives and objects to be stored in or retrieved from it. These interfaces are the `ObjectOutput` and `ObjectInput` interfaces which:

- Provide a stream to write to and to read from
- Handle requests to write primitive types and objects to the stream
- Handle requests to read primitive types and objects from the stream

Each object which is to be stored in a stream must explicitly allow itself to be stored and must implement the protocols needed to save and restore its state. Object Serialization defines two such protocols. The protocols allow the container to ask the object to write and read its state.

To be stored in an Object Stream, each object must implement either the `Serializable` or the `Externalizable` interface:

- For a `Serializable` class, Object Serialization can automatically save and restore fields of each class of an object and automatically handle classes that evolve by adding fields or supertypes. A serializable class can declare which of its fields are saved or restored, and write and read optional values and objects.
- For an `Externalizable` class, Object Serialization delegates to the class complete control over its external format and how the state of the supertype(s) is saved and restored.

1.5 Defining Serializable Fields for a Class

The serializable fields of a class can be defined two different ways. Default serializable fields of a class are defined to be the non-transient and non-static fields. This default computation can be overridden by declaring a special field in the `Serializable` class, `serialPersistentFields`. This field must be initialized with an array of `ObjectStreamField` objects that list the names and types of the serializable fields. The modifiers for the field are required to be private, static, and final. If the field's value is null or is otherwise not an instance of `ObjectStreamField[]`, or if the field does not have the required modifiers, then the behavior is as if the field were not declared at all.

For example, the following declaration duplicates the default behavior.

```
class List implements Serializable {
    List next;

    private static final ObjectStreamField[] serialPersistentFields
                 = {new ObjectStreamField("next", List.class)};

}
```

By using `serialPersistentFields` to define the Serializable fields for a class, there no longer is a limitation that a serializable field must be a field within the current definition of the `Serializable` class. The `writeObject` and `readObject` methods of the `Serializable` class can map the current implementation of the class to the serializable fields of the class using the interface that is described in Section 1.7, "Accessing Serializable Fields of a Class." Therefore, the fields for a `Serializable` class can change in a later release, as long as it maintains the mapping back to its Serializable fields that must remain compatible across release boundaries.

Note - There is, however, a limitation to the use of this mechanism to specify serializable fields for inner classes. Inner classes can only contain final static fields that are initialized to constants or expressions built up from constants. Consequently, it is not possible to set `serialPersistentFields` for an inner class (though it is possible to set it for static member classes). For other restrictions pertaining to serialization of inner class instances, see section Section 1.10, "The Serializable Interface".

1.6 Documenting Serializable Fields and Data for a Class

It is important to document the serializable state of a class to enable interoperability with alternative implementations of a Serializable class and to document class evolution. Documenting a serializable field gives one a final opportunity to review whether or not the field should be serializable. The serialization javadoc tags, `@serial`, `@serialField`, and `@serialData`, provide a way to document the serialized form for a Serializable class within the source code.

- The `@serial` tag should be placed in the javadoc comment for a default serializable field. The syntax is as follows: `@serial` *field-description* The optional *field-description* describes the meaning of the field and its acceptable values. The *field-description* can span multiple lines. When a field is added after the initial release, a *@since* tag indicates the version the field was added. The *field-description* for `@serial` provides serialization-specific documentation and is appended to the javadoc comment for the field within the serialized form documentation.
- The `@serialField` tag is used to document an `ObjectStreamField` component of a `serialPersistentFields` array. One of these tags should be used for each `ObjectStreamField` component. The syntax is as follows: `@serialField` *field-name field-type field-description*
- The `@serialData` tag describes the sequences and types of data written or read. The tag describes the sequence and type of optional data written by `writeObject` or all data written by the `Externalizable.writeExternal` method. The syntax is as follows: `@serialData` *data-description*

The javadoc application recognizes the serialization javadoc tags and generates a specification for each Serializable and Externalizable class. See Section C.1, "Example Alternate Implementation of java.io.File" for an example that uses these tags.

When a class is declared Serializable, the serializable state of the object is defined by serializable fields (by name and type) plus optional data. Optional data can only be written explicitly by the `writeObject` method of a `Serializable` class. Optional data can be read by the `Serializable` class' `readObject` method or serialization will skip unread optional data.

When a class is declared Externalizable, the data that is written to the stream by the class itself defines the serialized state. The class must specify the order, types, and meaning of each datum that is written to the stream. The class must handle its own evolution, so that it can continue to read data written by and write data that can be read by previous versions. The class must coordinate with the superclass when saving and restoring data. The location of the superclasses data in the stream must be specified.

The designer of a Serializable class must ensure that the information saved for the class is appropriate for persistence and follows the serialization-specified rules for interoperability and evolution. Class evolution is explained in greater detail in Chapter 5, "Versioning of Serializable Objects."

1.7 Accessing Serializable Fields of a Class

Serialization provides two mechanisms for accessing the serializable fields in a stream:

- The default mechanism requires no customization
- The Serializable Fields API allows a class to explicitly access/set the serializable fields by name and type

The default mechanism is used automatically when reading or writing objects that implement the `Serializable` interface and do no further customization. The serializable fields are mapped to the corresponding fields of the class and values are either written to the stream from those fields or are read in and assigned respectively. If the class provides `writeObject` and `readObject` methods, the default mechanism can be invoked by calling `defaultWriteObject` and `defaultReadObject`. When the `writeObject` and `readObject` methods are implemented, the class has an opportunity to modify the serializable field values before they are written or after they are read.

When the default mechanism cannot be used, the serializable class can use the `putFields` method of `ObjectOutputStream` to put the values for the serializable fields into the stream. The `writeFields` method of `ObjectOutputStream` puts the values in the correct order, then writes them to the stream using the existing protocol for serialization. Correspondingly, the `readFields` method of `ObjectInputStream` reads the values from the stream and makes them available to the class by name in any order. See Section 2.2, "The ObjectOutputStream.PutField Class" and Section 3.2, "The ObjectInputStream.GetField Class." for a detailed description of the Serializable Fields API.

1.8 The ObjectOutput Interface

The `ObjectOutput` interface provides an abstract, stream-based interface to object storage. It extends the DataOutput interface so those methods can be used for writing primitive data types. Objects that implement this interface can be used to store primitives and objects.

```
package java.io;

public interface ObjectOutput extends DataOutput
{
    public void writeObject(Object obj) throws IOException;
    public void write(int b) throws IOException;
    public void write(byte b[]) throws IOException;
     public void write(byte b[], int off, int len) throws IOException;
    public void flush() throws IOException;
    public void close() throws IOException;
}
```

`The writeObject` method is used to write an object. The exceptions thrown reflect errors while accessing the object or its fields, or exceptions that occur in writing to storage. If any exception is thrown, the underlying storage may be corrupted. If this occurs, refer to the object that is implementing this interface for more information.

1.9 The ObjectInput Interface

The `ObjectInput` interface provides an abstract stream based interface to object retrieval. It extends the `DataInput` interface so those methods for reading primitive data types are accessible in this interface.

```
package java.io;

public interface ObjectInput extends DataInput
{
        throws ClassNotFoundException, IOException;
    public int read() throws IOException;
    public int read(byte b[]) throws IOException;
    public int read(byte b[], int off, int len) throws IOException;
    public long skip(long n) throws IOException;
    public int available() throws IOException;
    public void close() throws IOException;
}
```

The `readObject` method is used to read and return an object. The exceptions thrown reflect errors while accessing the objects or its fields or exceptions that occur in reading from the storage. If any exception is thrown, the underlying storage may be corrupted. If this occurs, refer to the object implementing this interface for additional information.

1.10 The Serializable Interface

Object Serialization produces a stream with information about the Java^{TM} classes for the objects which are being saved. For serializable objects, sufficient information is kept to restore those objects even if a different (but compatible) version of the implementation of the class is present. The `Serializable` interface is defined to identify classes which implement the serializable protocol:

```
package java.io;

public interface Serializable {};
```

A Serializable class must do the following:

- Implement the `java.io.Serializable` interface
- Identify the fields that should be serializable

  (Use the `serialPersistentFields` member to explicitly declare them serializable or use the transient keyword to denote nonserializable fields.)

- Have access to the no-arg constructor of its first nonserializable superclass

The class can optionally define the following methods:

- A `writeObject` method to control what information is saved or to append additional information to the stream
- A `readObject` method either to read the information written by the corresponding `writeObject` method or to update the state of the object after it has been restored
- A `writeReplace` method to allow a class to nominate a replacement object to be written to the stream

  (See Section 2.5, "The writeReplace Method" for additional information.)

- A `readResolve` method to allow a class to designate a replacement object for the object just read from the stream

  (See Section 3.7, "The readResolve Method" for additional information.)

`ObjectOutputStream` and `ObjectInputStream` allow the serializable classes on which they operate to evolve (allow changes to the classes that are compatible with the earlier versions of the classes). See Section 5.5, "Compatible Java^{TM} Type Evolution" for information about the mechanism which is used to allow compatible changes.

Note - Serialization of inner classes (i.e., nested classes that are not static member classes), including local and anonymous classes, is strongly discouraged for several reasons. Because inner classes declared in non-static contexts contain implicit non-transient references to enclosing class instances, serializing such an inner class instance will result in serialization of its associated outer class instance as well. Synthetic fields generated by `javac` (or other Java^{TM} compilers) to implement inner classes are implementation dependent and may vary between compilers; differences in such fields can disrupt compatibility as well as result in conflicting default `serialVersionUID` values. The names assigned to local and anonymous inner classes are also implementation dependent and may differ between compilers. Since inner classes cannot declare static members other than compile-time constant fields, they cannot use the `serialPersistentFields` mechanism to designate serializable fields. Finally, because inner classes associated with outer instances do not have zero-argument constructors (constructors of such inner classes implicitly accept the enclosing instance as a prepended parameter), they cannot implement `Externalizable`. None of the issues listed above, however, apply to static member classes.

1.11 The Externalizable Interface

For Externalizable objects, only the identity of the class of the object is saved by the container; the class must save and restore the contents. The `Externalizable` interface is defined as follows:

```
package java.io;

public interface Externalizable extends Serializable
{
    public void writeExternal(ObjectOutput out)
        throws IOException;

    public void readExternal(ObjectInput in)
        throws IOException, java.lang.ClassNotFoundException;
}
```

The class of an Externalizable object must do the following:

- Implement the `java.io.Externalizable` interface
- Implement a `writeExternal` method to save the state of the object

  (It must explicitly coordinate with its supertype to save its state.)

- Implement a `readExternal` method to read the data written by the `writeExternal` method from the stream and restore the state of the object

  (It must explicitly coordinate with the supertype to save its state.)

- Have the `writeExternal` and `readExternal` methods be solely responsible for the format, if an externally defined format is written

---

Note - The `writeExternal` and `readExternal` methods are public and raise the risk that a client may be able to write or read information in the object other than by using its methods and fields. These methods must be used only when the information held by the object is not sensitive or when exposing it does not present a security risk.

- Have a public no-arg constructor

Note - Inner classes associated with enclosing instances cannot have no-arg constructors, since constructors of such classes implicitly accept the enclosing instance as a prepended parameter. Consequently the `Externalizable` interface mechanism cannot be used for inner classes and they should implement the `Serializable` interface, if they must be serialized. Several limitations exist for serializable inner classes as well, however; see Section 1.10, "The Serializable Interface", for a full enumeration.

An Externalizable class can optionally define the following methods:

- A `writeReplace` method to allow a class to nominate a replacement object to be written to the stream

(See Section 2.5, "The writeReplace Method" for additional information.)

- A `readResolve` method to allow a class to designate a replacement object for the object just read from the stream

  (See Section 3.7, "The readResolve Method" for additional information.)

1.12 Serialization of Enum Constants

Enum constants are serialized differently than ordinary serializable or externalizable objects. The serialized form of an enum constant consists solely of its name; field values of the constant are not present in the form. To serialize an enum constant, `ObjectOutputStream` writes the value returned by the enum constant's `name` method. To deserialize an enum constant, `ObjectInputStream` reads the constant name from the stream; the deserialized constant is then obtained by calling the `java.lang.Enum.valueOf` method, passing the constant's enum type along with the received constant name as arguments. Like other serializable or externalizable objects, enum constants can function as the targets of back references appearing subsequently in the serialization stream.

The process by which enum constants are serialized cannot be customized: any class-specific `writeObject`, `readObject`, `readObjectNoData`, `writeReplace`, and `readResolve` methods defined by enum types are ignored during serialization and deserialization. Similarly, any `serialPersistentFields` or `serialVersionUID` field declarations are also ignored--all enum types have a fixed `serialVersionUID` of `0L`. Documenting serializable fields and data for enum types is unnecessary, since there is no variation in the type of data sent.

1.13 Protecting Sensitive Information

When developing a class that provides controlled access to resources, care must be taken to protect sensitive information and functions. During deserialization, the private state of the object is restored. For example, a file descriptor contains a handle that provides access to an operating system resource. Being able to forge a file descriptor would allow some forms of illegal access, since restoring state is done from a stream. Therefore, the serializing runtime must take the conservative approach and not trust the stream to contain only valid representations of objects. To avoid compromising a class, the sensitive state of an object must not be restored from the stream, or it must be reverified by the class. Several techniques are available to protect sensitive data in classes.

The easiest technique is to mark fields that contain sensitive data as *private transient*. Transient fields are not persistent and will not be saved by any persistence mechanism. Marking the field will prevent the state from appearing in the stream and from being restored during deserialization. Since writing and reading (of private fields) cannot be superseded outside of the class, the transient fields of the class are safe.

Particularly sensitive classes should not be serialized at all. To accomplish this, the object should not implement either the `Serializable` or the `Externalizable` interface.

Some classes may find it beneficial to allow writing and reading but specifically handle and revalidate the state as it is deserialized. The class should implement `writeObject` and `readObject` methods to save and restore only the appropriate state. If access should be denied, throwing a `NotSerializableException` will prevent further access.

CONTENTS | PREV | NEXT