Java™ Object Serialization Specification

Object serialization in the Java™ system is the process of creating a serialized representation of objects or a graph of objects. Object values and types are serialized with sufficient information to insure that the equivalent typed object can be recreated. Deserialization is the symmetric process of recreating the object or graph of objects from the serialized representation. Different versions of a class can write and read compatible streams.

Revision 1.43
JDK™ 1.2,   November 30, 1998

Table of Contents

Change History

Sept. 30, 1998 **Updates for JDK™ 1.2 Beta4 RC1**

- Documentation corrections only.

June 22, 1998 **Updates for JDK™ 1.2 Beta4**

- Eliminated JDK™ 1.2 `java.io` interfaces, `Replaceable` and `Resolvable`. References to either of these classes as an interface should be replaced with `java.io.Serializable`. Serialization will use reflection to invoke the methods, `writeReplace` and `readResolve`, if the Serializable class defines these methods. See Section 2.5, "The writeReplace Method" and Section 3.6, "The readResolve Method."
- New javadoc tags `@serial`, `@serialField`, and `@serialData` provide a way to document the Serialized Form of a Serializable class. Javadoc generates a serialization specification based on the contents of these tags. See Section 1.6, "Documenting Serializable Fields and Data for a Class."
- Special Serializable class member, `serialPersistentFields`, must be declared private. See Section 1.5, "Defining Serializable Fields for a Class."
- Clarified the steps involved in computing the `serialVersionUID` in Section 4.4, "Stream Unique Identifiers."

Feb. 6, 1998 **Updates for JDK™ 1.2 Beta 3**

- Introduced the concept of STREAM_PROTOCOL versions. Added the `STREAM_PROTOCOL_2` version to indicate a new format for `Externalizable` objects that enable skipping by an `Externalizable` object within the stream, even when the object's class is not available in the local Virtual Machine. Compatibility issues are discussed in Section 6.3, "Stream Protocol Versions."
- `The ObjectInputStream.resolveClass` method can return a local class in a different package than the name of the class within the stream. This capability enables renaming of packages between releases. The `serialVersionUID` and the base class name must be the same in the stream and in the local version of the class. See Section 3.1, "The ObjectInputStream Class."
- Allow substitution of `String` or `array` objects when writing them to or reading them from the stream. See Section 2.1, "The ObjectOutputStream Class" and Section 3.1, "The ObjectInputStream Class."

Sept. 4, 1997 **Updates for JDK™ 1.2 Beta1**

- Separated the `Replaceable` interface into two interfaces: `Replaceable` and `Resolvable`. The `Replaceable` interface allows a class to nominate its own replacement just before serializing the object to the stream. The `Resolvable` interface allows a class to nominate its own replacement when reading an object from the stream.
- Modified serialization to use the JDK™ 1.2 security model. There is a check for `SerializablePermission "enableSubstitution"` within the `ObjectInputStream.enableReplace` and `ObjectOutputStream.enableResolve` methods. See Section 2.1, "The ObjectOutputStream Class" and Section 3.1, "The ObjectInputStream Class."
- Updated `writeObject`'s exception handler to write handled `IOExceptions` into the stream. See Section 2.1, "The ObjectOutputStream Class."

July 3, 1997 **Updates for JDK™ 1.2 Alpha**

- Documented the requirements for specifying the serialized state of classes. See Section 1.5, “Defining Serializable Fields for a Class.”
- Added the Serializable Fields API to allow classes more flexibility in accessing the serialized fields of a class. The stream protocol is unchanged. See Section 1.7, “Accessing Serializable Fields of a Class,” Section 2.2, “The ObjectOutputStream.PutField Class,” and Section 3.2, “The ObjectInputStream.GetField Class.”
- Clarified that field descriptors and data are written to and read from the stream in canonical order. See Section 4.1, “The ObjectStreamClass Class.”

System Architecture 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
- Protecting Sensitive Information

1.1 Overview

The ability to store and retrieve Java™ 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™ objects, the serialized form must be able to identify and verify the Java™ 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™ objects are to:

- Have a simple yet extensible mechanism.
- Maintain the Java™ 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™ 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 objects of type `Class`, `ObjectStreamClass`, `String`, and arrays. 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.

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.

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
{
    public Object readObject()
        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™ 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 3.6, “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™ Type Evolution” for information about the mechanism which is used to allow compatible changes.

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
- Have a public no-arg constructor

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.

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 3.6, "The readResolve Method" for additional information.)

1.12 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.

Object Output Classes 2

Topics:

- The ObjectOutputStream Class
- The ObjectOutputStream.PutField Class
- The writeObject Method
- The writeExternal Method
- The writeReplace Method
- The useProtocolVersion Method

2.1 The ObjectOutputStream Class

Class `ObjectOutputStream` implements object serialization. It maintains the state of the stream including the set of objects already serialized. Its methods control the traversal of objects to be serialized to save the specified objects and the objects to which they refer.

```
package java.io;

public class ObjectOutputStream
    extends OutputStream
    implements ObjectOutput, ObjectStreamConstants
{
    public ObjectOutputStream(OutputStream out)
        throws IOException;

    public final void writeObject(Object obj)
        throws IOException;
```

```
public void defaultWriteObject();
    throws IOException, NotActiveException;

public PutField putFields()
    throws IOException;

public writeFields()
    throws IOException;

public void reset() throws IOException;

protected void annotateClass(Class cl) throws IOException;

protected Object replaceObject(Object obj) throws IOException;

protected boolean enableReplaceObject(boolean enable)
    throws SecurityException;

protected void writeStreamHeader() throws IOException;

public void write(int data) 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;

protected void drain() throws IOException;

public void close() throws IOException;

public void writeBoolean(boolean data) throws IOException;

public void writeByte(int data) throws IOException;

public void writeShort(int data) throws IOException;

public void writeChar(int data) throws IOException;

public void writeInt(int data) throws IOException;

public void writeLong(long data) throws IOException;

public void writeFloat(float data) throws IOException;
```

```
public void writeDouble(double data) throws IOException;

public void writeBytes(String data) throws IOException;

public void writeChars(String data) throws IOException;

public void writeUTF(String data) throws IOException;

// Inner class to provide access to serializable fields.
abstract static public class PutField
{
    public void put(String name, boolean value)
        throws IOException, IllegalArgumentException;

    public void put(String name, char data)
        throws IOException, IllegalArgumentException;

    public void put(String name, byte data)
        throws IOException, IllegalArgumentException;

    public void put(String name, short data)
        throws IOException, IllegalArgumentException;

    public void put(String name, int data)
        throws IOException, IllegalArgumentException;

    public void put(String name, long data)
        throws IOException, IllegalArgumentException;

    public void put(String name, float data)
        throws IOException, IllegalArgumentException;

    public void put(String name, double data)
        throws IOException, IllegalArgumentException;

    public void put(String name, Object data)
        throws IOException, IllegalArgumentException;
}

public void useProtocolVersion(int version) throws IOException;

protected ObjectOutputStream()
    throws IOException;
```

```
    protected writeObjectOverride()
        throws NotActiveException, IOException;
}
```

The `ObjectOutputStream` constructor requires an `OutputStream`. The constructor calls `writeStreamHeader` to write a magic number and version to the stream, that will be read and verified by the corresponding `readStreamHeader` in the `ObjectInputStream` constructor.

The `writeObject` method is used to serialize an object to the stream. An object is serialized as follows:

1. **If a subclass is overriding the implementation, call the `writeObjectOverride` method and return. Overriding the implementation is described at the end of this section.**

2. **If there is data in the block-data buffer, the data is written to the stream and the buffer is reset.**

3. **If the object is null, null is put in the stream and `writeObject` returns.**

4. **If the object has been previously replaced, as described in Step 8, write the handle of the replacement to the stream and `writeObject` returns.**

5. **If the object has already been written to the stream, its handle is written to the stream and `writeObject` returns.**

6. **If the object is a `Class`, the corresponding `ObjectStreamClass` is written to the stream, a handle is assigned for the class, and `writeObject` returns.**

7. **If the object is an `ObjectStreamClass`, a descriptor for the class is written to the stream including its name, `serialVersionUID`, and the list of fields by name and type. A handle is assigned for the descriptor. The `annotateClass` subclass method is called before `writeObject` returns.**

8. **Process potential substitutions by the class of the object and/or by a subclass of `ObjectInputStream`.**

    a. **If the class of an object defines the appropriate `writeReplace` method, the method is called. Optionally, it can return a substitute object to be serialized.**

b. **Then, if enabled by calling the `enableReplaceObject` method, the `replaceObject` method is called to allow subclasses of `ObjectOutputStream` to substitute for the object being serialized. If the original object was replaced in the previous step, the `replaceObject` method is called with the replacement object.**

If the original object was replaced by either one or both steps above, the mapping from the original object to the replacement is recorded for later use in Step 4. Then, Steps 3 through 7 are repeated on the new object.

If the replacement object is not one of the types covered by Steps 3 through 7, processing resumes using the replacement object at Step 10.

9. **If the object is a `java.lang.String`, the string is written in Universal Transfer Format (UTF) format, a handle is assigned to the string, and `writeObject` returns.**

10. **If the object is an array, `writeObject` is called recursively to write the `ObjectStreamClass` of the array. The handle for the array is assigned. It is followed by the length of the array. Each element of the array is then written to the stream, after which `writeObject` returns.**

11. **For regular objects, the `ObjectStreamClass` for the class of the object is written by recursively calling `writeObject`. It will appear in the stream only the first time it is referenced. A handle is assigned for this object.**

12. **The contents of the object is written to the stream.**

    a. **If the object is serializable, the highest serializable class is located. For that class, and each derived class, that class's fields are written. If the class does not have a `writeObject` method, the `defaultWriteObject` method is called to write the serializable fields to the stream. If the class does have a `writeObject` method, it is called. It may call `defaultWriteObject` or `putFields` and `writeFields` to save the state of the object, and then it can write other information to the stream.**

    b. **If the object is externalizable, the `writeExternal` method of the object is called.**

    c. **If the object is neither serializable or externalizable, the `NotSerializableException` is thrown.**

Exceptions may occur during the traversal or may occur in the underlying stream. For any subclass of `IOException`, the exception is written to the stream using the exception protocol and the stream state is discarded. If a second `IOException` is thrown while attempting to write the first exception into the stream, the stream is left in an unknown state and `StreamCorruptedException` is thrown from `writeObject`. For other exceptions, the stream is aborted and left in an unknown and unusable state.

The `defaultWriteObject` method implements the default serialization mechanism for the current class. This method may be called only from a class's `writeObject` method. The method writes all of the serializable fields of the current class to the stream. If called from outside the `writeObject` method, the `NotActiveException` is thrown.

The `putFields` method returns a `PutField` object the caller uses to set the values of the serializable fields in the stream. The fields may be set in any order. After all of the fields have been set, `writeFields` must be called to write the field values in the canonical order to the stream. If a field is not set, the default value appropriate for its type will be written to the stream. This method may only be called from within the `writeObject` method of a serializable class. It may not be called more than once or if `defaultWriteObject` has been called. Only after `writeFields` has been called can other data be written to the stream.

The `reset` method resets the stream state to be the same as if it had just been constructed. `Reset` will discard the state of any objects already written to the stream. The current point in the stream is marked as reset, so the corresponding `ObjectInputStream` will reset at the same point. Objects previously written to the stream will not be remembered as already having been written to the stream. They will be written to the stream again. This is useful when the contents of an object or objects must be sent again. `Reset` may not be called while objects are being serialized. If called inappropriately, an `IOException` is thrown.

The `annotateClass` method is called while a `Class` is being serialized, and after the class descriptor has been written to the stream. Subclasses may extend this method and write other information to the stream about the class. This information must be read by the `resolveClass` method in a corresponding `ObjectInputStream` subclass.

An `ObjectOutputStream` subclass can implement the `replaceObject` method to monitor or replace objects during serialization. Replacing objects must be enabled explicitly by calling `enableReplaceObject` before calling `writeObject` with the first object to be replaced. Once enabled, `replaceObject` is called for each object just prior to serializing the object for the first time. Note that the `replaceObject` method is not called for objects of the specially handled classes, `Class` and `ObjectStreamClass`. An implementation of a subclass may return a substitute object that will be serialized instead of the original. The substitute object must be serializable. All references in the stream to the original object will be replaced by the substitute object.

When objects are being replaced, the subclass must ensure that the substituted object is compatible with every field where the reference will be stored, or that a complementary substitution will be made during deserialization. Objects, whose type is not a subclass of the type of the field or array element, will later abort the deserialization by raising a `ClassCastException` and the reference will not be stored.

The `enableReplaceObject` method can be called by trusted subclasses of `ObjectOutputStream` to enable the substitution of one object for another during serialization. Replacing objects is disabled until `enableReplaceObject` is called with a `true` value. It may thereafter be disabled by setting it to `false`. The previous setting is returned. The `enableReplaceObject` method checks that the stream requesting the replacement can be trusted. To ensure that the private state of objects is not unintentionally exposed, only trusted stream subclasses may use `replaceObject`. Trusted classes are those classes that belong to a security protection domain with permission to enable Serializable substitution.

If the subclass of `ObjectOutputStream` is not considered part of the system domain, `SerializablePermission "enableSubstitution"` must be added to the security policy file. `AccessControlException` is thrown if the protection domain of the subclass of `ObjectInputStream` does not have permission to `"enableSubstitution"` by calling `enableReplaceObject`. See the document *Java™ Security Architecture (JDK1.2)* for additional information about the security model.

The `writeStreamHeader` method writes the magic number and version to the stream. This information must be read by the `readStreamHeader` method of `ObjectInputStream`. Subclasses may need to implement this method to identify the stream's unique format.

The `flush` method is used to empty any buffers being held by the stream and to forward the flush to the underlying stream. The `drain` method may be used by subclassers to empty only the `ObjectOutputStream`'s buffers without forcing the underlying stream to be flushed.

All of the write methods for primitive types encode their values using a `DataOutputStream` to put them in the standard stream format. The bytes are buffered into block data records so they can be distinguished from the encoding of objects. This buffering allows primitive data to be skipped if necessary for class versioning. It also allows the stream to be parsed without invoking class-specific methods.

To override the implementation of serialization, the subclass of `ObjectOutputStream` should call the protected no-arg `ObjectOutputStream`, constructor. There is a security check within the no-arg constructor for `SerializablePermission "enableSubclassImplementation"` to ensure that only trusted classes are allowed to override the default implementation. This constructor does not allocate any private data for `ObjectOutputStream` and sets a flag that indicates that the final `writeObject` method should invoke the `writeObjectOverride` method and return. All other `ObjectOutputStream` methods are not final and can be directly overridden by the subclass.

2.2 The ObjectOutputStream.PutField Class

Class `PutField` provides the API for setting values of the serializable fields for a class when the class does not use default serialization. Each method puts the specified named value into the stream. I/O exceptions will be thrown if the underlying stream throws an exception. An `IllegalArgumentException` is thrown if the name does not match the name of a field declared for this object's `ObjectStreamClass` or if the type of the value does not match the declared type of the serializable field.

2.3 The writeObject Method

For serializable objects, the `writeObject` method allows a class to control the serialization of its own fields. Here is its signature:

```
private void writeObject(ObjectOutputStream stream)
    throws IOException;
```

Each subclass of a serializable object may define its own `writeObject` method. If a class does not implement the method, the default serialization provided by `defaultWriteObject` will be used. When implemented, the class is only responsible for having its own fields, not those of its supertypes or subtypes.

The class's `writeObject` method, if implemented, is responsible for saving the state of the class. The `defaultWriteObject` method should be called before writing any optional data that will be needed by the corresponding `readObject` method to restore the state of the object. The responsibility for the format, structure, and versioning of the optional data lies completely with the class.

2.4 The writeExternal Method

Objects implementing `java.io.Externalizable` must implement the `writeExternal` method to save the entire state of the object. It must coordinate with its superclasses to save their state. All of the methods of `ObjectOutput` are available to save the object's primitive typed fields and object fields.

```
public void writeExternal(ObjectOutput stream)
    throws IOException;
```

A new default format for writing Externalizable data has been introduced in JDK™ 1.2. The new format specifies that primitive data will be written in block data mode by `writeExternal` methods. Additionally, a tag denoting the end of the External object is appended to the stream after the `writeExternal` method returns. The benefits of this format change are discussed in Section 3.5, "The readExternal Method." Compatibility issues caused by this change are discussed in Section 2.6, "The useProtocolVersion Method."

2.5 The writeReplace Method

For Serializable and Externalizable classes, the `writeReplace` method allows a class of an object to nominate its own replacement in the stream before the object is written. By implementing the `writeReplace` method, a class can directly control the types and instances of its own instances being serialized.

The method is defined as follows:

```
ANY-ACCESS-MODIFIER Object writeReplace() {
     throws ObjectStreamException;
```

The `writeReplace` method is called when `ObjectOutputStream` is preparing to write the object to the stream. The `ObjectOutputStream` checks whether the class defines the `writeReplace` method. If the method is defined, the `writeReplace` method is called to allow the object to designate its replacement in the stream. The object returned should be either of the same type as the object passed in or an object that when read and resolved will result in an object of a type that is compatible with all references to the object. If it is not, a `ClassCastException` will occur when the type mismatch is discovered.

2.6 The useProtocolVersion Method

Due to a stream protocol change that was not backwards compatible, a mechanism has been added to enable the current Virtual Machine to write a serialization stream that is readable by a previous release. Of course, the problems that are corrected by the new stream format will exist when using the backwards compatible protocol.

Stream protocol versions are discussed in Section 6.3, “Stream Protocol Versions.”

Object Input Classes 3

Topics:

- The ObjectInputStream Class
- The ObjectInputStream.GetField Class
- The ObjectInputValidation Interface
- The readObject Method
- The readExternal Method
- The readResolve Method

3.1 The ObjectInputStream Class

Class `ObjectInputStream` implements object deserialization. It maintains the state of the stream including the set of objects already deserialized. Its methods allow primitive types and objects to be read from a stream written by `ObjectOutputStream`. It manages restoration of the object and the objects that it refers to from the stream.

```
package java.io;

public class ObjectInputStream
    extends InputStream
    implements ObjectInput, ObjectStreamConstants
{
    public ObjectInputStream(InputStream in)
        throws StreamCorruptedException, IOException;
```

```
public final Object readObject()
    throws OptionalDataException, ClassNotFoundException,
        IOException;

public void defaultReadObject()
    throws IOException, ClassNotFoundException,
        NotActiveException;

public GetField readFields()
    throws IOException;

public synchronized void registerValidation(
    ObjectInputValidation obj, int prio)
    throws NotActiveException, InvalidObjectException;

protected Class resolveClass(ObjectStreamClass v)
    throws IOException, ClassNotFoundException;

protected Object resolveObject(Object obj)
    throws IOException;

protected boolean enableResolveObject(boolean enable)
    throws SecurityException;

protected void readStreamHeader()
    throws IOException, StreamCorruptedException;

public int read() throws IOException;

public int read(byte[] data, int offset, int length)
    throws IOException

public int available() throws IOException;

public void close() throws IOException;

public boolean readBoolean() throws IOException;

public byte readByte() throws IOException;

public int readUnsignedByte() throws IOException;

public short readShort() throws IOException;

public int readUnsignedShort() throws IOException;
```

```
public char readChar() throws IOException;

public int readInt() throws IOException;

public long readLong() throws IOException;

public float readFloat() throws IOException;

public double readDouble() throws IOException;

public void readFully(byte[] data) throws IOException;

public void readFully(byte[] data, int offset, int size)
    throws IOException;

public int skipBytes(int len) throws IOException;

public String readLine() throws IOException;

public String readUTF() throws IOException;

// Class to provide access to serializable fields.
static abstract public class GetField
{
    public ObjectStreamClass getObjectStreamClass();

    public boolean defaulted(String name)
        throws IOException, IllegalArgumentException;

    public char get(String name, char default)
        throws IOException, IllegalArgumentException;

    public boolean get(String name, boolean default)
        throws IOException, IllegalArgumentException;

    public byte get(String name, byte default)
        throws IOException, IllegalArgumentException;

    public short get(String name, short default)
        throws IOException, IllegalArgumentException;

    public int get(String name, int default)
        throws IOException, IllegalArgumentException;

    public long get(String name, long default)
        throws IOException, IllegalArgumentException;
```

```
        public float get(String name, float default)
            throws IOException, IllegalArgumentException;

        public double get(String name, double default)
            throws IOException, IllegalArgumentException;

        public Object get(String name, Object default)
            throws IOException, IllegalArgumentException;
    }

    protected ObjectInputStream()
        throws StreamCorruptedException, IOException;

    protected readObjectOverride()
        throws OptionalDataException, ClassNotFoundException,
            IOException;
}
```

The `ObjectInputStream` constructor requires an `InputStream`. The constructor calls `readStreamHeader` to read and verifies the header and version written by the corresponding `ObjectOutputStream.writeStreamHeader` method.

The `readObject` method is used to deserialize an object from the stream. It reads from the stream to reconstruct an object.

1. **If the `ObjectInputStream` subclass is overriding the implementation, call the `readObjectOverride` method and return. Reimplementation is described at the end of this section.**
2. **If a block data record occurs in the stream, throw a `BlockDataException` with the number of available bytes.**
3. **If the object in the stream is null, return null.**
4. **If the object in the stream is a handle to a previous object, return the object.**
5. **If the object in the stream is a `Class`, read its `ObjectStreamClass` descriptor, add it and its handle to the set of known objects, and return the corresponding `Class` object.**

6. If the object in the stream is an `ObjectStreamClass`, read its name, `serialVersionUID`, and fields. Add it and its handle to the set of known objects. Call the `resolveClass` method on the stream to get the local class for this descriptor, and throw an exception if the class cannot be found. Return the `ObjectStreamClass` object.

7. If the object in the stream is a String, read its UTF encoding, add it and its handle to the set of known objects, and proceed to Step 11.

8. If the object in the stream is an array, read its `ObjectStreamClass` and the length of the array. Allocate the array, and add it and its handle in the set of known objects. Read each element using the appropriate method for its type and assign it to the array. Proceed to Step 11.

9. For all other objects, the `ObjectStreamClass` of the object is read from the stream. The local class for that `ObjectStreamClass` is retrieved. The class must be serializable or externalizable.

10. An instance of the class is allocated. The instance and its handle are added to the set of known objects. The contents restored appropriately:

    a. For serializable objects, the no-arg constructor for the first non-serializable supertype is run. For serializable classes, the fields are initialized to the default value appropriate for its type. Then the fields of each class are restored by calling class-specific `readObject` methods, or if these are not defined, by calling the `defaultReadObject` method. Note that field initializers and constructors are not executed for serializable classes during deserialization. In the normal case, the version of the class that wrote the stream will be the same as the class reading the stream. In this case, all of the supertypes of the object in the stream will match the supertypes in the currently-loaded class. If the version of the class that wrote the stream had different supertypes than the loaded class, the `ObjectInputStream` must be more careful about restoring or initializing the state of the differing classes. It must step through the classes, matching the available data in the stream with the classes of the object being restored. Data for classes that occur in the stream, but do not occur in the object, is discarded. For classes that occur in the object, but not in the stream, the class fields are set to default values by default serialization.

    b. For externalizable objects, the no-arg constructor for the class is run and then the `readExternal` method is called to restore the contents of the object.

11. **Process potential substitutions by the class of the object and/or by a subclass of** `ObjectInputStream`:

    a. **If the class of the object defines the appropriate** `readResolve` **method, the method is called to allow the object to replace itself.**

    b. **Then if previously enabled by** `enableResolveObject`**, the** `resolveObject` **method is called to allow subclasses of the stream to examine and replace the object. If the previous step did replace the original object, the** `resolveObject` **method is called with the replacement object.**

    If a replacement took place, the table of known objects is updated so the replacement object is associated with the handle. The replacement object is then returned from `readObject`.

All of the methods for reading primitives types only consume bytes from the block data records in the stream. If a read for primitive data occurs when the next item in the stream is an object, the read methods return `-1` or the `EOFException` as appropriate. The value of a primitive type is read by a `DataInputStream` from the block data record.

The exceptions thrown reflect errors during the traversal or exceptions that occur on the underlying stream. If any exception is thrown, the underlying stream is left in an unknown and unusable state.

When the reset token occurs in the stream, all of the state of the stream is discarded. The set of known objects is cleared.

When the exception token occurs in the stream, the exception is read and a new `WriteAbortedException` is thrown with the terminating exception as an argument. The stream context is reset as described earlier.

The `defaultReadObject` method is used to read the fields and object from the stream. It uses the class descriptor in the stream to read the fields in the canonical order by name and type from the stream. The values are assigned to the matching fields by name in the current class. Details of the versioning mechanism can be found in Section 5.5, "Compatible Java™ Type Evolution." Any field of the object that does not appear in the stream is set to its default value. Values that appear in the stream, but not in the object, are discarded. This occurs primarily when a later version of a class has written additional fields that do not occur in the earlier version. This method may only be called from the `readObject` method while restoring the fields of a class. When called at any other time, the `NotActiveException` is thrown.

The `readFields` method reads the values of the serializable fields from the stream and makes them available via the `GetField` class. The `readFields` method is only callable from within the `readObject` method of a serializable class. It cannot be called more than once or if `defaultReadObject` has been called. The `GetFields` object uses the current object's `ObjectStreamClass` to verify the fields that can be retrieved for this class. The `GetFields` object returned by `readFields` is only valid during this call to the classes `readObject` method. The fields may be retrieved in any order. Additional data may only be read directly from stream after `readFields` has been called.

The `registerValidation` method can be called to request a callback when the entire graph has been restored but before the object is returned to the original caller of `readObject`. The order of validate callbacks can be controlled using the priority. Callbacks registered with higher values are called before those with lower values. The object to be validated must support the `ObjectInputValidation` interface and implement the `validateObject` method. It is only correct to register validations during a call to a class's `readObject` method. Otherwise, a `NotActiveException` is thrown. If the callback object supplied to `registerValidation` is null, an `InvalidObjectException` is thrown.

The `resolveClass` method is called while a class is being deserialized, and after the class descriptor has been read. Subclasses may extend this method to read other information about the class written by the corresponding subclass of `ObjectOutputStream`. The method must find and return the class with the given name and `serialVersionUID`. The default implementation locates the class by calling the class loader of the closest caller of `readObject` that has a class loader. If the class cannot be found `ClassNotFoundException` should be thrown. Prior to JDK™ 1.1.6, the `resolveClass` method was required to return the same fully qualified class name as the class name in the stream. In order to accommodate package renaming across releases, `method resolveClass` only needs to return a class with the same base class name and `SerialVersionUID` in JDK™ 1.1.6 and later versions.

The `resolveObject` method is used by trusted subclasses to monitor or substitute one object for another during deserialization. Resolving objects must be enabled explicitly by calling `enableResolveObject` before calling `readObject` for the first object to be resolved. Once enabled, `resolveObject` is called once for each serializable object just prior to the first time it is being returned from `readObject`. Note that the `resolveObject` method is not called for objects of the specially handled classes, `Class`, `ObjectStreamClass`, `String`, and arrays. A subclass's implementation of `resolveObject` may return a substitute object that will be assigned or returned instead of the original. The object returned must be of a type that is consistent and assignable to every reference of the original object or else a `ClassCastException` will be thrown. All assignments are type-checked. All references in the stream to the original object will be replaced by references to the substitute object.

The `enableResolveObject` method is called by trusted subclasses of `ObjectOutputStream` to enable the monitoring or substitution of one object for another during deserialization. Replacing objects is disabled until `enableResolveObject` is called with a *`true`* value. It may thereafter be disabled by setting it to *`false`*. The previous setting is returned. The `enableResolveObject` method checks if the stream has permission to request substitution during serialization. To ensure that the private state of objects is not unintentionally exposed, only trusted streams may use `resolveObject`. Trusted classes are those classes with a class loader equal to null or belong to a security protection domain that provides permission to enable substitution.

If the subclass of `ObjectInputStream` is not considered part of the system domain, a line has to be added to the security policy file to provide to a subclass of `ObjectInputStream` permission to call `enableResolveObject`. The `SerializablePermission` to add is `"enableSubstitution"`. `AccessControlException` is thrown if the protection domain of the subclass of `ObjectStreamClass` does not have permission to `"enableSubstitution"` by calling `enableResolveObject`. See the document *Java™ Security Architecture (JDK™ 1.2)* for additional information about the security model.

The `readStreamHeader` method reads and verifies the magic number and version of the stream. If they do not match, the `StreamCorruptedMismatch` is thrown.

To override the implementation of deserialization, a subclass of `ObjectInputStream` should call the protected no-arg `ObjectInputStream`, constructor. There is a security check within the no-arg constructor for `SerializablePermission "enableSubclassImplementation"` to ensure that only trusted classes are allowed to override the default implementation. This constructor does not allocate any private data for `ObjectInputStream` and sets a flag that indicates that the final `readObject` method should invoke the `readObjectOverride` method and return. All other `ObjectInputStream` methods are not final and can be directly overridden by the subclass.

3.2 The ObjectInputStream.GetField Class

The class `ObjectInputStream.GetField` provides the API for getting the values of serializable fields. The protocol of the stream is the same as used by `defaultReadObject.` Using `readFields` to access the serializable fields does not change the format of the stream. It only provides an alternate API to access the values which does not require the class to have the corresponding non-transient and non-static fields for each named serializable field. The serializable fields are those declared using `serialPersistentFields` or if it is not declared the non-transient and non-static fields of the object. When the stream is read the available serializable fields are those written to the stream when the object was serialized. If the class that wrote the stream is a different version not all fields will correspond to the serializable fields of the current class. The available fields can be retrieved from the `ObjectStreamClass` of the `GetField` object.

The `getObjectStreamClass` method returns an `ObjectStreamClass` object representing the class in the stream. It contains the list of serializable fields.

The `defaulted` method returns *true* if the field is not present in the stream. An `IllegalArgumentException` is thrown if the requested field is not a serializable field of the current class.

Each `get` method returns the specified serializable field from the stream. I/O exceptions will be thrown if the underlying stream throws an exception. An `IllegalArgumentException` is thrown if the name or type does not match the name and type of an field serializable field of the current class. The default value is returned if the stream does not contain an explicit value for the field.

3.3 The ObjectInputValidation Interface

This interface allows an object to be called when a complete graph of objects has been deserialized. If the object cannot be made valid, it should throw the `ObjectInvalidException`. Any exception that occurs during a call to `validateObject` will terminate the validation process, and the `InvalidObjectException` will be thrown.

```
package java.io;

public interface ObjectInputValidation
{
```

```
    public void validateObject()
        throws InvalidObjectException;
}
```

3.4 The readObject Method

For serializable objects, the `readObject` method allows a class to control the deserialization of its own fields. Here is its signature:

```
  private void readObject(ObjectInputStream stream)
        throws IOException, ClassNotFoundException;
```

Each subclass of a serializable object may define its own `readObject` method. If a class does not implement the method, the default serialization provided by `defaultReadObject` will be used. When implemented, the class is only responsible for restoring its own fields, not those of its supertypes or subtypes.

The `readObject` method of the class, if implemented, is responsible for restoring the state of the class. The values of every field of the object whether transient or not, static or not are set to the default value for the fields type. The `defaultReadObject` method should be called before reading any optional data written by the corresponding `writeObject` method. If the `readObject` method of the class attempts to read more data than is present in the optional part of the stream for this class, the stream will throw an `EOFException`. The responsibility for the format, structure, and versioning of the optional data lies completely with the class. The `@serialData` javadoc tag within the javadoc comment for the `readObject` method should be used to document the format and structure of the optional data.

If the class being restored is not present in the stream being read, its fields are initialized to the appropriate default values.

Reading an object from the `ObjectInputStream` is analogous to creating a new object. Just as a new object's constructors are invoked in the order from the superclass to the subclass, an object being read from a stream is deserialized from superclass to subclass. The `readObject` or `defaultReadObject` method is called instead of the constructor for each `Serializable` subclass during deserialization.

One last similarity between a constructor and a `readObject` method is that both provide the opportunity to invoke a method on an object that is not fully constructed. Any non-final method called while an object is being constructed can potentially be overridden by a subclass. Methods called during the construction phase of an object are resolved by the actual type of the object, not the type currently being initialized by either its constructor or `readObject` method. This situation results in the overriding method being invoked on an object that is not fully constructed yet.

3.5 The readExternal Method

Objects implementing `java.io.Externalizable` must implement the `readExternal` method to restore the entire state of the object. It must coordinate with its superclasses to restore their state. All of the methods of `ObjectInput` are available to restore the object's primitive typed fields and object fields.

```
public void readExternal(ObjectInput stream)
    throws IOException;
```

Note – The `readExternal` method is public, and it raises the risk of a client being able to overwrite an existing object from a stream. The class may add its own checks to insure that this is only called when appropriate.

A new stream protocol version has been introduced in JDK™ 1.2 to correct a problem with `Externalizable` objects. The old definition of `Externalizable` objects required the local virtual machine to find a `readExternal` method to be able to properly read an `Externalizable` object from the stream. The new format adds enough information to the stream protocol so serialization can skip an `Externalizable` object when the local `readExternal` method is not available. Due to class evolution rules, serialization must be able to skip an `Externalizable` object in the input stream if there is not a mapping for the object using the local classes.

An additional benefit of the new `Externalizable` stream format is the exception `EOFException` is thrown if a `readExternal` method attempts to read beyond the end of an `Externalizable` object. Also, serialization is able to skip by any External data that is not read by a `readExternal` method.

Due to the format change, JDK™ 1.1.6 and earlier releases are not able to read the new format. `StreamCorruptedException` is thrown when JDK™ 1.1.6 or earlier attempts to read an `Externalizable` object from a stream written in `PROTOCOL_VERSION_2`. Compatibility issues are discussed in more detail in Section 6.3, "Stream Protocol Versions."

3.6 The readResolve Method

For Serializable and Externalizable classes, the `readResolve` method allows a class to replace/resolve the object read from the stream before it is returned to the caller. By implementing the `readResolve` method, a class can directly control the types and instances of its own instances being deserialized. The method is defined as follows:

```
ANY-ACCESS-MODIFIER Object readResolve()
    throws ObjectStreamException;
```

The `readResolve` method is called when `ObjectInputStream` has read an object from the stream and is preparing to return it to the caller. `ObjectInputStream` checks whether the class of the object defines the `readResolve` method. If the method is defined, the `readResolve` method is called to allow the object in the stream to designate the object to be returned. The object returned should be of a type that is compatible with all uses. If it is not compatible, a `ClassCastException` will be thrown when the type mismatch is discovered.

For example, a `Symbol` class could be created for which only a single instance of each symbol binding existed within a virtual machine. The `readResolve` method would be implemented to determine if that symbol was already defined and substitute the preexisting equivalent `Symbol` object to maintain the identity constraint. In this way the uniqueness of `Symbol` objects can be maintained across serialization.

Class Descriptors 4

Topics:

- The ObjectStreamClass Class
- The ObjectStreamField Class
- Inspecting Serializable Classes
- Stream Unique Identifiers

4.1 The ObjectStreamClass Class

The `ObjectStreamClass` provides information about classes that are saved in a Serialization stream. The descriptor provides the fully-qualified name of the class and its serialization version UID. A `SerialVersionUID` identifies the unique original class version for which this class is capable of writing streams and from which it can read.

```
package java.io;

public class ObjectStreamClass
{
    public static ObjectStreamClass lookup(Class cl);

    public String getName();

    public Class forClass();

    public ObjectStreamField[] getFields();

    public long getSerialVersionUID();
```

```
    public String toString();
}
```

The `lookup` method returns the `ObjectStreamClass` descriptor for the specified class in the virtual machine. If the class has defined `serialVersionUID` it is retrieved from the class. If the `serialVersionUID` is not defined by the class, it is computed from the definition of the class in the virtual machine. If the specified class is not serializable or externalizable, `null` is returned.

The `getName` method returns the fully-qualified name of the class. The class name is saved in the stream and is used when the class must be loaded.

The `forClass` method returns the `Class` in the local virtual machine if one was found by `ObjectInputStream.resolveClass` method. Otherwise, it returns `null`.

The `getFields` method returns an array of `ObjectStreamField` objects that represent the serializable fields of this class.

The `getSerialVersionUID` method returns the `serialVersionUID` of this class. Refer to Section 4.4, "Stream Unique Identifiers." If not specified by the class, the value returned is a hash computed from the class's name, interfaces, methods, and fields using the Secure Hash Algorithm (SHA) as defined by the National Institute of Standards.

The `toString` method returns a printable representation of the class descriptor including the name of the class and the `serialVersionUID`.

When an `ObjectStreamClass` instance is written to the stream, it writes the class name and `serialVersionUID`, flags, and the number of fields. Depending on the class, additional information may be written:

- For non-serializable classes, the number of fields is always zero. Neither the serializable nor the externalizable flag bits are set.
- For serializable classes, the serializable flag is set, the number of fields counts the number of serializable fields and is followed by a descriptor for each serializable field. The descriptors are written in canonical order. The descriptors for primitive typed fields are written first sorted by field name followed by descriptors for the object typed fields sorted by field name. The names are sorted using `String.compareTo`. The protocol describes the format.
- For externalizable classes, flags includes the externalizable flag, and the number of fields is always zero.

4.2 The ObjectStreamField Class

An `ObjectStreamField` represents a serializable field of a serializable class. The serializable fields of a class can be retrieved from the `ObjectStreamClass`.

The special static Serializable field, serialPersistentFields, an array of `ObjectStreamField` components is used to override the default serializable fields.

```
package java.io;

public class ObjectStreamField {

    public ObjectStreamField(String fieldName, Class fieldType);

    public String getName();

    public Class getType() throws ClassNotFoundException;

    public String toString();
}
```

The `ObjectStreamField` constructor is used to create a new instance of an `ObjectStreamField`. The argument is the type of the serializable field. For example, `Integer.TYPE` or `java.lang.Hashtable.class`. `ObjectStreamField` objects are used to specify the serializable fields of a class or to describe the fields present in a stream.

The `getName` method returns the name of the serializable field.

The `getType` method returns the type of the field.

The `toString` method returns a printable representation with name and type.

4.3 Inspecting Serializable Classes

The program `serialver` can be used to find out if a class is serializable and to get its `serialVersionUID`. When invoked with the `-show` option, it puts up a simple user interface. To find out if a class is serializable and to find out its `serialVersionUID`, enter its full class name, then press either the Enter or the Show button. The string printed can be copied and pasted into the evolved class.

When invoked on the command line with one or more class names, `serialver` prints the `serialVersionUID` for each class in a form suitable for copying into an evolving class. When invoked with no arguments, `serialver` prints a usage line.

4.4 Stream Unique Identifiers

Each versioned class must identify the original class version for which it is capable of writing streams and from which it can read. For example, a versioned class must declare:

```
private static final long serialVersionUID = 3487495895819393L;
```

The stream-unique identifier is a 64-bit hash of the class name, interface class names, methods, and fields. The value must be declared in all versions of a class except the first. It may be declared in the original class but is not required. The value is fixed for all compatible classes. If the SUID is not declared for a class, the value defaults to the hash for that class. `Serializable` classes do not need to anticipate versioning; however, `Externalizable` classes do. The initial version of an `Externalizable` class must output a stream data format that is extensible in the future. The initial version of the method `readExternal` has to be able to read the output format of all future versions of the method `writeExternal`.

The `serialVersionUID` is computed using the signature of a stream of bytes that reflect the class definition. The National Institute of Standards and Technology (NIST) Secure Hash Algorithm (SHA-1) is used to compute a signature for the stream. The first 8 bytes of the signature are used to form a 64-bit hash. A `java.lang.DataOutputStream` is used to convert primitive data types to a sequence of bytes. The values input to the stream are defined by the Java™ Virtual Machine (VM) specification for classes.

The sequence of items in the stream is as follows:

1. Class name, written using UTF encoding.
2. Class modifiers, written as a 32-bit integer.

3. The name of each interface, sorted by name and written using UTF encoding.
4. For each field of the class sorted by field name (except private static and private transient fields):
   a. Field name, in UTF encoding.
   b. Field modifiers, written as a 32-bit integer.
   c. Field descriptor, in UTF encoding.
5. If a class initializer exists, write out the following:
   a. Method name, `<clinit>`, in UTF encoding.
   b. Method modifier, `java.lang.reflect.Modifier.STATIC`, written as a 32-bit integer.
   c. Method descriptor, `()V`, in UTF encoding.
6. For each non-private constructor sorted by method name and signature:
   a. Method name, `<init>`, in UTF encoding.
   b. Method modifiers, written as a 32-bit integer.
   c. Method descriptor, in UTF encoding.
7. For each non-private method sorted by method name and signature:
   a. Method name, in UTF encoding.
   b. Method modifiers, written as a 32-bit integer.
   c. Method descriptor, in UTF encoding.
8. The SHA-1 algorithm is executed on the stream of bytes produced by `DataOutputStream` and produces 20 byte values, `sha[0..19]`.
9. The hash value is assembled from the first 8 byte values of `sha` array.

```
long h = 0;
for (int i = 0; i < Math.min(8, sha.length); i++) {
    h += (long)(sha[i] & 255) << (i * 8);
}
```

Versioning of Serializable Objects 5

Topics:

- Overview
- Goals
- Assumptions
- Who's Responsible for Versioning of Streams
- Compatible Java™ Type Evolution
- Type Changes Affecting Serialization

5.1 Overview

When Java™ objects use serialization to save state in files, or as blobs in databases, the potential arises that the version of a class reading the data is different than the version that wrote the data.

Versioning raises some fundamental questions about the identity of a class, including what constitutes a compatible change. A *compatible change* is a change that does not affect the contract between the class and its callers.

This section describes the goals, assumptions, and a solution that attempts to address this problem by restricting the kinds of changes allowed and by carefully choosing the mechanisms.

The proposed solution provides a mechanism for “automatic” handling of classes that evolve by adding fields and adding classes. Serialization will handle versioning without class-specific methods to be implemented for each version. The stream format can be traversed without invoking class-specific methods.

5.2 Goals

The goals are to:

- Support bidirectional communication between different versions of a class operating in different virtual machines by:
  - Defining a mechanism that allows Java™ classes to read streams written by older versions of the same class.
  - Defining a mechanism that allows Java™ classes to write streams intended to be read by older versions of the same class.
- Provide default serialization for persistence and for RMI.
- Perform well and produce compact streams in simple cases, so that RMI can use serialization.
- Be able to identify and load classes that match the exact class used to write the stream.
- Keep the overhead low for nonversioned classes.
- Use a stream format that allows the traversal of the stream without having to invoke methods specific to the objects saved in the stream.

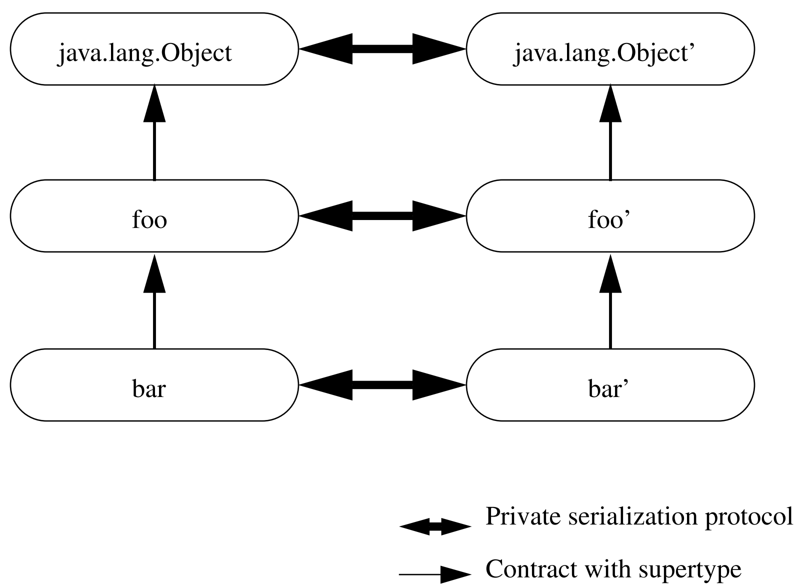
5.3 Assumptions

The assumptions are that:

- Versioning will only apply to serializable classes since it must control the stream format to achieve it goals. Externalizable classes will be responsible for their own versioning which is tied to the external format.
- All data and objects must be read from, or skipped in, the stream in the same order as they were written.
- Classes evolve individually as well as in concert with supertypes and subtypes.
- Classes are identified by name. Two classes with the same name may be different versions or completely different classes that can be distinguished only by comparing their interfaces or by comparing hashes of the interfaces.
- Default serialization will not perform any type conversions.
- The stream format only needs to support a linear sequence of type changes, not arbitrary branching of a type.

5.4 *Who's Responsible for Versioning of Streams*

In the evolution of classes, it is the responsibility of the evolved (later version) class to maintain the contract established by the nonevolved class. This takes two forms. First, the evolved class must not break the existing assumptions about the interface provided by the original version, so that the evolved class can be used in place of the original. Secondly, when communicating with the original (or previous) versions, the evolved class must provide sufficient and equivalent information to allow the earlier version to continue to satisfy the nonevolved contract.



For the purposes of the discussion here, each class implements and extends the interface or contract defined by its supertype. New versions of a class, for example `foo'`, must continue to satisfy the contract for `foo` and may extend the interface or modify its implementation.

Communication between objects via serialization is not part of the contract defined by these interfaces. Serialization is a private protocol between the implementations. It is the responsibility of the implementations to communicate sufficiently to allow each implementation to continue to satisfy the contract expected by its clients.

5.5 Compatible Java™ Type Evolution

In the *Java™ Language Specification*, Chapter 13 discusses binary compatibility of Java™ classes as those classes evolve. Most of the flexibility of binary compatibility comes from the use of late binding of symbolic references for the names of classes, interfaces, fields, methods, and so on.

The following are the principle aspects of the design for versioning of serialized object streams.

- The default serialization mechanism will use a symbolic model for binding the fields in the stream to the fields in the corresponding class in the virtual machine.
- Each class referenced in the stream will uniquely identify itself, its supertype, and the types and names of each serializable field written to the stream. The fields are ordered with the primitive types first sorted by field name, followed by the object fields sorted by field name.
- Two types of data may occur in the stream for each class: required data (corresponding directly to the serializable fields of the object); and optional data (consisting of an arbitrary sequence of primitives and objects). The stream format defines how the required and optional data occur in the stream so that the whole class, the required, or the optional parts can be skipped if necessary.
  - The required data consists of the fields of the object in the order defined by the class descriptor.
  - The optional data is written to the stream and does not correspond directly to fields of the class. The class itself is responsible for the length, types, and versioning of this optional information.
- If defined for a class, the `writeObject`/`readObject` methods supersede the default mechanism to write/read the state of the class. These methods write and read the optional data for a class. The required data is written by calling `defaultWriteObject` and read by calling `defaultReadObject`.
- The stream format of each class is identified by the use of a Stream Unique Identifier (SUID). By default, this is the hash of the class. All later versions of the class must declare the Stream Unique Identifier (SUID) that they are compatible with. This guards against classes with the same name that might inadvertently be identified as being versions of a single class.

- Subtypes of `ObjectOutputStream` and `ObjectInputStream` may include their own information identifying the class using the `annotateClass` method; for example, `MarshalOutputStream` embeds the URL of the class.

5.6 Type Changes Affecting Serialization

With these concepts, we can now describe how the design will cope with the different cases of an evolving class. The cases are described in terms of a stream written by some version of a class. When the stream is read back by the same version of the class, there is no loss of information or functionality. The stream is the only source of information about the original class. Its class descriptions, while a subset of the original class description, are sufficient to match up the data in the stream with the version of the class being reconstituted.

The descriptions are from the perspective of the stream being read in order to reconstitute either an earlier or later version of the class. In the parlance of RPC systems, this is a "receiver makes right" system. The writer writes its data in the most suitable form and the receiver must interpret that information to extract the parts it needs and to fill in the parts that are not available.

5.6.1 Incompatible Changes

Incompatible changes to classes are those changes for which the guarantee of interoperability cannot be maintained. The incompatible changes that may occur while evolving a class are:

- Deleting fields - If a field is deleted in a class, the stream written will not contain its value. When the stream is read by an earlier class, the value of the field will be set to the default value because no value is available in the stream. However, this default value may adversely impair the ability of the earlier version to fulfill its contract.
- Moving classes up or down the hierarchy - This cannot be allowed since the data in the stream appears in the wrong sequence.
- Changing a nonstatic field to static or a nontransient field to transient - When relying on default serialization, this change is equivalent to deleting a field from the class. This version of the class will not write that data to the stream, so it will not be available to be read by earlier versions of the class. As when deleting a field, the field of the earlier version will be initialized to the default value, which can cause the class to fail in unexpected ways.

- Changing the declared type of a primitive field - Each version of the class writes the data with its declared type. Earlier versions of the class attempting to read the field will fail because the type of the data in the stream does not match the type of the field.
- Changing the `writeObject` or `readObject` method so that it no longer writes or reads the default field data or changing it so that it attempts to write it or read it when the previous version did not. The default field data must consistently either appear or not appear in the stream.
- Changing a class from `Serializable` to `Externalizable` or visa-versa is an incompatible change since the stream will contain data that is incompatible with the implementation in the available class.
- Removing either `Serializable` or `Externalizable` is an incompatible change since when written it will no longer supply the fields needed by older versions of the class.
- Adding the `writeReplace` or `readResolve` method to a class is incompatible if the behavior would produce an object that is incompatible with any older version of the class.

5.6.2 Compatible Changes

The compatible changes to a class are handled as follows:

- Adding fields - When the class being reconstituted has a field that does not occur in the stream, that field in the object will be initialized to the default value for its type. If class-specific initialization is needed, the class may provide a readObject method that can initialize the field to nondefault values.
- Adding classes - The stream will contain the type hierarchy of each object in the stream. Comparing this hierarchy in the stream with the current class can detect additional classes. Since there is no information in the stream from which to initialize the object, the class's fields will be initialized to the default values.
- Removing classes - Comparing the class hierarchy in the stream with that of the current class can detect that a class has been deleted. In this case, the fields and objects corresponding to that class are read from the stream. Primitive fields are discarded, but the objects referenced by the deleted class are created, since they may be referred to later in the stream. They will be garbage-collected when the stream is garbage-collected or reset.
- Adding `writeObject/readObject` methods - If the version reading the stream has these methods then `readObject` is expected, as usual, to read the required data written to the stream by the default serialization. It should call

`defaultReadObject` first before reading any optional data. The `writeObject` method is expected as usual to call `defaultWriteObject` to write the required data and then may write optional data.

- Removing `writeObject`/`readObject` methods - If the class reading the stream does not have these methods, the required data will be read by default serialization, and the optional data will be discarded.
- Adding `java.io.Serializable` - This is equivalent to adding types. There will be no values in the stream for this class so its fields will be initialized to default values. The support for subclassing nonserializable classes requires that the class's supertype have a no-arg constructor and the class itself will be initialized to default values. If the no-arg constructor is not available, the `InvalidClassException` is thrown.
- Changing the access to a field - The access modifiers public, package, protected, and private have no effect on the ability of serialization to assign values to the fields.
- Changing a field from static to nonstatic or transient to nontransient - When relying on default serialization to compute the serializable fields, this change is equivalent to adding a field to the class. The new field will be written to the stream but earlier classes will ignore the value since serialization will not assign values to static or transient fields.

Object Serialization Stream Protocol 6

Topics:

6.1 Overview

The stream format satisfies the following design goals:

- Is compact and is structured for efficient reading.
- Allows skipping through the stream using only the knowledge of the structure and format of the stream. Does not require invoking any per class code.
- Requires only stream access to the data.

6.2 Stream Elements

A basic structure is needed to represent objects in a stream. Each attribute of the object needs to be represented: its classes, its fields, and data written and later read by class-specific methods. The representation of objects in the stream can be described with a grammar. There are special representations for null objects, new objects, classes, arrays, strings, and back references to any object already in the stream. Each object written to the stream is assigned a handle that is used to refer back to the object. Handles are assigned sequentially starting from 0x7E0000. The handles restart at 0x7E0000 when the stream is reset.

A class object is represented by the following:

- Its `ObjectStreamClass` object.

An `ObjectStreamClass` object is represented by the following:

- The Stream Unique Identifier (SUID) of compatible classes.
- A flag indicating if the class had `writeObject`/`readObject` methods
- The number of serializable fields
- The array of fields of the class that are serialized by the default mechanism For arrays and object fields, the type of the field is included as a string.
- Optional block-data records or objects written by the `annotateClass` method
- The `ObjectStreamClass` of its supertype (null if the superclass is not serializable)

Strings are represented by their UTF encoding. Note that the current specification and implementation of the modified UTF restricts the total length of the encoded string to 65535 characters.

Arrays are represented by the following:

- Their `ObjectStreamClass` object.
- The number of elements.
- The sequence of values. The type of the values is implicit in the type of the array. for example the values of a byte array are of type byte.

New objects in the stream are represented by the following:

- The most derived class of the object.
- Data for each serializable class of the object, with the highest superclass first. For each class the stream contains the following:
  - The serializable fields. See Section 1.5, "Defining Serializable Fields for a Class."
  - If the class has `writeObject`/`readObject` methods, there may be optional objects and/or block-data records of primitive types written by the `writeObject` method followed by an `endBlockData` code.

All primitive data written by classes is buffered and wrapped in block-data records, regardless if the data is written to the stream within a `writeObject` method or written directly to the stream from outside a `writeObject` method. This data can only be read by the corresponding `readObject` methods or be read directly from the stream. Objects written by the `writeObject` method terminate any previous block-data record and are written either as regular objects or null or back references, as appropriate. The block-data records allow error recovery to discard any optional data. When called from within a class, the stream can discard any data or objects until the `endBlockData`.

6.3 Stream Protocol Versions

It was necessary to make a change to the serialization stream format in JDK™ 1.2 that is not backwards compatible to all minor releases of JDK™ 1.1. To provide for cases where backwards compatibility is required, a capability has been added to indicate what `PROTOCOL_VERSION` to use when writing a serialization stream. The method `ObjectOutputStream.useProtocolVersion` takes as a parameter the protocol version to use to write the serialization stream.

The Stream Protocol Versions are as follows:

- `ObjectStreamConstants.PROTOCOL_VERSION_1`
  Indicates the initial stream format.
- `ObjectStreamConstants.PROTOCOL_VERSION_2`
  Indicates the new external data format. Primitive data is written in block data mode and is terminated with `TC_ENDBLOCKDATA`.

  Block data boundaries have been standardized. Primitive data written in block data mode is normalized to not exceed 1024 byte chunks. The benefit of this change was to tighten the specification of serialized data format within the stream. This change is fully backward and forward compatible.

JDK™ 1.2 defaults to writing `PROTOCOL_VERSION_2`.

JDK™ 1.1 defaults to writing `PROTOCOL_VERSION_1`.

JDK™ 1.1.7 and greater can read both versions.

Releases prior to JDK™ 1.1.7 can only read `PROTOCOL_VERSION_1`.

6.4 Grammar for the Stream Format

The table below contains the grammar for the stream format. Nonterminal symbols are shown in *italics*. Terminal symbols in a `fixed width font`. Definitions of nonterminals are followed by a ":". The definition is followed by one or more alternatives, each on a separate line. The following table describes the notation:

Notation	Meaning
(*datatype*)	This token has the data type specified, such as byte.
`token[n]`	A predefined number of occurrences of the token, that is an array.
`x0001`	A literal value expressed in hexadecimal. The number of hex digits reflects the size of the value.
<*xxx*>	A value read from the stream used to indicate the length of an array.

6.4.1 Rules of the Grammar

A Serialized stream is represented by any stream satisfying the *stream* rule.

```
stream:
  magic version contents

contents:
  content
  contents content

content:
  object
  blockdata

object:
  newObject
  newClass
  newArray
  newString
  newClassDesc
  prevObject
  nullReference
  exception
  TC_RESET

newClass:
  TC_CLASS classDesc newHandle
```

```
classDesc:
  newClassDesc
  nullReference
  (ClassDesc)prevObject      // an object required to be of type
                               // ClassDesc

superClassDesc:
  classDesc

newClassDesc:
  TC_CLASSDESC className serialVersionUID newHandle classDescInfo

classDescInfo:
  classDescFlags fields classAnnotation superClassDesc

className:
  (utf)

serialVersionUID:
  (long)

classDescFlags:
  (byte)                   // Defined in Terminal Symbols and
                               // Constants

fields:
  (short)<count>  fieldDesc[count]

fieldDesc:
  primitiveDesc
  objectDesc

primitiveDesc:
  prim_typecode fieldName

objectDesc:
  obj_typecode fieldName className1

fieldName:
  (utf)

className1:
  (String)object             // String containing the field's type

classAnnotation:
  endBlockData
  contents endBlockData      // contents written by annotateClass
```

```
prim_typecode:
   'B'// byte
   'C'// char
   'D'// double
   'F'// float
   'I'// integer
   'J'// long
   'S'// short
   'Z'// boolean

obj_typecode:
   '['// array
   'L'// object

newArray:
  TC_ARRAY classDesc newHandle (int)<size> values[size]

newObject:
  TC_OBJECT classDesc newHandle classdata[]  // data for each class

classdata:
  nowrclass                  // SC_SERIALIZABLE & classDescFlag &&
                             // !(SC_WRITE_METHOD & classDescFlags)
  wrclass objectAnnotation   // SC_SERIALIZABLE & classDescFlag &&
                             // SC_WRITE_METHOD & classDescFlags
  externalContents           // SC_EXTERNALIZABLE & classDescFlag &&
                             // !(SC_BLOCKDATA  & classDescFlags
  objectAnnotation           // SC_EXTERNALIZABLE & classDescFlag&&
                             // SC_BLOCKDATA & classDescFlags

nowrclass:
  values                     // fields in order of class descriptor

wrclass:
  nowrclass

objectAnnotation:
  endBlockData
  contents endBlockData      // contents written by writeObject
                             // or writeExternal PROTOCOL_VERSION_2.

blockdata:
  blockdatashort
  blockdatalong

blockdatashort:
  TC_BLOCKDATA (unsigned byte)<size> (byte)[size]

blockdatalong:
  TC_BLOCKDATALONG (int)<size> (byte)[size]

endBlockData:
  TC_ENDBLOCKDATA
```

```
externalContent:          // Only parseable by readExternal
  ( bytes)                // primitive data
    object

externalContents:         // externalContent written by
  externalContent         // writeExternal in PROTOCOL_VERSION_1.
  externalContents externalContent

newString:
  TC_STRING newHandle (utf)

prevObject
  TC_REFERENCE (int)handle

nullReference
  TC_NULL

exception:
  TC_EXCEPTION reset (Throwable)object reset

magic:
  STREAM_MAGIC

version
  STREAM_VERSION

values:           // The size and types are described by the
                  // classDesc for the current object

newHandle:        // The next number in sequence is assigned
                  // to the object being serialized or deserialized

reset:            // The set of known objects is discarded
                  // so the objects of the exception do not
                  // overlap with the previously sent objects
                  // or with objects that may be sent after
                  // the exception
```

6.4.2 Terminal Symbols and Constants

The following symbols in `java.io.ObjectStreamConstants` define the terminal and constant values expected in a stream.

```
final static short STREAM_MAGIC = (short)0xaced;
final static short STREAM_VERSION = 5;
final static byte TC_NULL = (byte)0x70;
final static byte TC_REFERENCE = (byte)0x71;
final static byte TC_CLASSDESC = (byte)0x72;
final static byte TC_OBJECT = (byte)0x73;
final static byte TC_STRING = (byte)0x74;
final static byte TC_ARRAY = (byte)0x75;
```

```
    final static byte TC_CLASS = (byte)0x76;
    final static byte TC_BLOCKDATA = (byte)0x77;
    final static byte TC_ENDBLOCKDATA = (byte)0x78;
    final static byte TC_RESET = (byte)0x79;
    final static byte TC_BLOCKDATALONG = (byte)0x7A;
    final static byte TC_EXCEPTION = (byte)0x7B;
    final static  int   baseWireHandle = 0x7E0000;
```

The flag byte `classDescFlags` may include values of

```
    final static byte SC_WRITE_METHOD = 0x01; //if SC_SERIALIZABLE
    final static byte SC_BLOCK_DATA = 0x08;    //if SC_EXTERNALIZABLE
    final static byte SC_SERIALIZABLE = 0x02;
    final static byte SC_EXTERNALIZABLE = 0x04;
```

The flag `SC_WRITE_METHOD` is set if the Serializable class writing the stream had a `writeObject` method that may have written additional data to the stream. In this case a `TC_ENDBLOCKDATA` marker is always expected to terminate the data for that class.

The flag `SC_BLOCKDATA` is set if the `Externalizable` class is written into the stream using `STREAM_PROTOCOL_2`. By default, this is the protocol used to write `Externalizable` objects into the stream in JDK™ 1.2. JDK™ 1.1 writes `STREAM_PROTOCOL_1`.

The flag `SC_SERIALIZABLE` is set if the class that wrote the stream extended `java.io.Serializable` but not `java.io.Externalizable`, the class reading the stream must also extend `java.io.Serializable` and the default serialization mechanism is to be used.

The flag `SC_EXTERNALIZABLE` is set if the class that wrote the stream extended `java.io.Externalizable`, the class reading the data must also extend `Externalizable` and the data will be read using its `writeExternal` and `readExternal` methods.

Example

Consider the case of an original class and two instances in a linked list:

```
class List implements java.io.Serializable {
    int value;
    List next;
    public static void main(String[] args) {
        try {
            List list1 = new List();
            List list2 = new List();
```

```
            list1.value = 17;
            list1.next = list2;
            list2.value = 19;
            list2.next = null;

            ByteArrayOutputStream o = new ByteArrayOutputStream();
            ObjectOutputStream out = new ObjectOutputStream(o);
            out.writeObject(list1);
            out.writeObject(list2);
            out.flush();
            ...
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

The resulting stream contains:

```
00: ac ed 00 05 73 72 00 04 4c 69 73 74 69 c8 8a 15 >....sr..Listi...<
10: 40 16 ae 68 02 00 02 49 00 05 76 61 6c 75 65 4c >Z......I..valueL<
20: 00 04 6e 65 78 74 74 00 06 4c 4c 69 73 74 3b 78 >..nextt..LList;x<
30: 70 00 00 00 11 73 71 00 7e 00 00 00 00 00 13 70 >p....sq.~......p<
40: 71 00 7e 00 03                                  >q.~..<
```

Security in Object Serialization A

Topics:

- Overview
- Design Goals
- Using transient to Protect Important System Resources
- Writing Class-Specific Serializing Methods
- Encrypting a Bytestream

A.1 Overview

The object serialization system allows a bytestream to be produced from a graph of objects, sent out of the Java™ environment (either saved to disk or sent over the network) and then used to recreate an equivalent set of new objects with the same state.

What happens to the state of the objects outside of the environment is outside of the control of the Java™ system (by definition), and therefore is outside the control of the security provided by the system. The question then arises, once an object has been serialized, can the resulting byte array be examined and changed, perhaps injecting viruses into Java™ programs? The intent of this section is to address these security concerns.

A.2 Design Goals

The goal for object serialization is to be as simple as possible and yet still be consistent with known security restrictions; the simpler the system is, the more likely it is to be secure. The following points summarize how security in object serialization has been implemented:

- Only objects implementing the `java.io.Serializable` or `java.io.Externalizable` interfaces can be serialized. there are mechanisms for not serializing certain fields and certain classes.
- The serialization package cannot be used to recreate the *same* object, and no object is ever overwritten by a deserialize operation. All that can be done with the serialization package is to create *new* objects, initialized in a particular fashion.
- While deserializing an object might cause code for the class of the object to be loaded, that code loading is protected by all of the usual Java™ code verification and security management guarantees. Classes loaded because of deserialization are no more or less secure than those loaded in any other fashion.
- Externalizable objects expose themselves to being overwritten because the `readExternal` method is public.

A.3 Using transient to Protect Important System Resources

Direct handles to system resources, such as file handles, are the kind of information that is relative to an address space and should not be written out as part of an object's persistent state. Therefore, fields that contain this kind of information should be declared `transient`, which prevents them from being serialized. Note that this is not a new or overloaded meaning for the `transient` keyword.

If a resource, like a file handle, was not declared `transient`, the object could be altered while in its serialized state, enabling it to have improper access to resources after it is deserialized.

A.4 Writing Class-Specific Serializing Methods

To guarantee that a deserialized object does not have state which violates some set of invariants that need to be guaranteed, a class can define its own serializing and deserializing methods. If there is some set of invariants that need to be maintained between the data members of a class, only the class can know about these invariants, and it is up to the class writer to provide a deserialization method that checks these invariants.

This is important even if you are not worried about security; it is possible that disk files can be corrupted and serialized data be invalid. So checking such invariants is more than just a security measure; it is a validity measure. However, the only place it can be done is in the code for the particular class, since there is no way the serialization package can determine what invariants should be maintained or checked.

A.5 Encrypting a Bytestream

Another way of protecting a bytestream outside the virtual machine is to encrypt the stream produced by the serialization package. Encrypting the bytestream prevents the decoding and the reading of a serialized object's private state.

The implementation allows encryption, both by allowing the classes to have their own special methods for serialization/deserialization and by using the stream abstraction for serialization, so the output can be fed into some other stream or filter.

Exceptions In Object Serialization B

All exceptions thrown by serialization classes are subclasses of `ObjectStreamException` which is a subclass of `IOException`.

Exception	Description
`ObjectStreamException`	Superclass of all serialization exceptions.
`InvalidClassException`	Thrown when a class cannot be used to restore objects for any of these reasons: • The class does not match the serial version of the class in the stream. • The class contains fields with invalid primitive data types. • The `Externalizable` class does not have a public no-arg constructor. • The `Serializable` class can not access the no-arg constructor of its closest non-Serializable superclass.
`NotSerializableException`	Thrown by a `readObject` or `writeObject` method to terminate serialization or deserialization.
`StreamCorruptedException`	Thrown: • If the stream header is invalid. • If control information not found. • If control information is invalid. • JDK™ 1.1.5 or less attempts to call `readExternal` on a `PROTOCOL_VERSION_2` stream.

Exception	Description
`NotActiveException`	Thrown if `writeObject` state is invalid within the following `ObjectOutputStream` methods: • `defaultWriteObject` • `putFields` • `writeFields` Thrown if `readObject` state is invalid within the following `ObjectInputStream` methods: • `defaultReadObject` • `readFields` • `registerValidation`
`InvalidObjectException`	Thrown when a restored object cannot be made valid.
`OptionalDataException`	Thrown by `readObject` when there is primitive data in the stream and an object is expected. The length field of the exception indicates the number of bytes that are available in the current block.
`WriteAbortedException`	Thrown when reading a stream terminated by an exception that occurred while the stream was being written.

Example of Serializable Fields C

Topics:

- Example Alternate Implementation of java.io.File

C.1 *Example Alternate Implementation of java.io.File*

This appendix provides a brief example of how an existing class could be specified and implemented to interoperate with the existing implementation but without requiring the same assumptions about the representation of the file name as a `String`.

The system class `java.io.File` represents a filename and has methods for parsing, manipulating files and directories by name. It has a single private field that contains the current file name. The semantics of the methods that parse paths depend on the current path separator which is held in a static field. This path separator is part of the serialized state of a file so that file name can be adjusted when read.

The serialized state of a `File` object is defined as the serializable fields and the sequence of data values for the file. In this case, there is one of each.

```
Serializable Fields:
    String path;     // path name with embedded separators
Serializable Data:
    char            // path name separator for path name
An alternate implementation might be defined as follows:
class File implements java.io.Serializable {
    ...
```

```
    private String[] pathcomponents;
    // Define serializable fields with the ObjectStreamClass

    /**
     * @serialField path String
     *              Path components separated by separator.
     */
    private static final ObjectStreamField[] serialPersistentFields
=   }
            new ObjectStreamField("path", String.class)
    };

    ...
    /**
   * @serialData Default fields followed by separator character.
    */
    private void writeObject(ObjectOutputStream s)
        throws IOException
    {
        ObjectOutputStream.PutField fields = s.putFields();
        StringBuffer str = new StringBuffer();
        for(int i = 0; i < pathcomponents; i++) {
            str.append(separator);
            str.append(pathcomponents[i]);
        }
        fields.put("path", str.toString());
        s.writeFields();
        s.writeChar(separatorChar); // Add the separator character
    }
    ...
    private void readObject(ObjectInputStream s)
        throws IOException
    {
        ObjectInputStream.GetField fields = s.readFields();
        String path = (String)fields.get("path", null);
        ...
       char sep = s.readChar(); // read the previous separator char
        // parse path into components using the separator
        // and store into pathcomponents array.
    }
}
```