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 1.2, December 2, 1996

Table of Contents

System Architecture 1

Topics:

1.1 Overview

The capability to store and retrieve Java objects is essential to building all but the most transient applications. The key to storing and retrieving objects is representing the state of objects in a serialized form 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 object's contents 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 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 straight forward 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 OutputStream. 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 2.2, “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`, strings, and arrays. Other objects must implement either Serializable or Externalizable interfaces 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`. All primitive data is written to the stream in block-data records prefixed by a marker and the length. Putting the data in records allows it to be skipped if necessary.

`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 is equally straight forward:

```
// 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. More 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 `DataOutput` interface, such as `readInt`, `readFloat`, or `readUTF`. Individual bytes and arrays of bytes are read with the methods of `InputStream`. All 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. Object Serialization has been designed to provide a rich set of features for Java classes. Other container formats such as OLE or OpenDoc have different stream or file system representations.

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

- Provide a stream to write to and read from,
- Handle requests to write primitive types and objects to 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 transient (not 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 is saved and restored.

1.5 The ObjectOutput Interface

The `ObjectOutput` interface provides an abstract stream based interface to object storage. It extends `DataOutput` so those methods may be used for writing primitive data types. Objects implementing 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, refer to the object implementing this interface for details.

1.6 The ObjectInput Interface

The `ObjectInput` interface provides an abstract stream based interface to object retrieval. It extends `DataInput` 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, refer to the object implementing this interface for details.

1.7 The Serializable Interface

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

```
package java.io;

public interface Serializable {};
```

A `Serializable` object:

- Must implement the `java.io.Serializable` interface.
- Must mark its fields that are not to be persistent with the transient keyword.
- Can implement a `writeObject` method to control what information is saved or to append additional information to the stream.
- Can implement a `readObject` method so it can read the information written by the corresponding `writeObject` method or to update the state of the object after it has been restored.

ObjectOutputStream and ObjectInputStream are designed and implemented to allow the Serializable classes they operate on to evolve, that is, to allow changes to the classes that are compatible with the earlier versions of the classes. Details of the mechanism to allow compatible changes can be found in Section 5.5, “Compatible Java Type Evolution.

1.8 The Externalizable Interface

For Externalizable objects only the identity of class of the object is saved by the container and it is the responsibility of the class to save and restore the contents. The interface Externalizable is defined as:

```
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;
}
```

An Externalizable Object:

- Must implement the java.io.Externalizable interface.
- Must implement a writeExternal method to save the state of the object. It must explicitly coordinate with its supertype to save its state.
- Must 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.
- If writing an externally defined format the writeExternal and readExternal methods are solely responsible for that format.

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 would not present a security risk.

1.9 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 re-verified 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 and static fields are not serialized or deserialized. 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 class's transient fields are safe.

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

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 Interfaces 2

Topics:

- The ObjectOutputStream Class
- The writeObject Method
- The writeExternal 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 final void defaultWriteObject();
        throws IOException, NotActiveException;
    public void reset() throws IOException;

    protected void annotateClass(Class cl) throws IOException;

    protected Object replaceObject(Object obj) throws IOException;

    protected final 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;

    public 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;
}
```

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. 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 aborted and left in an unknown and unusable state. Objects are serialized as follows:

1. If there is data in the block-data buffer it is written to the stream and the buffer reset.
2. If the object is null, null is put in the stream and writeObject returns.
3. If the object has already been written to the stream, its handle is written to the stream and writeObject returns. If the object has been already been replaced the handle for the previously written replacement object is written to the stream.
4. If the object is a Class, the corresponding ObjectStreamClass is written to the stream, a handle assigned for the class and writeObject returns.
5. 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.
6. If the object is a `java.lang.String` the string is written in Universal Transfer Format (UTF) format, a handle assigned to the string and writeObject returns.
7. 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.
8. If enabled by calling `enableReplaceObject`, the `replaceObject` method is called to allow subclasses to substitute an object. If the object is replaced the mapping from the original object to the replacement is stored for later use in step 3 and steps 2 through 7 are repeated on the new object. If the replacement object is not one of the types covered by steps 2 through 7 processing resumes using the replacement object at step 9.

9. For regular objects, the ObjectStreamClass for the object's class 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.

10. The contents of the object are written to the stream.
    - 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 non-static and non-transient fields to the stream. If the class does have a `writeObject` method it is called. It may call `defaultWriteObject` to save the state of the object and then it can write other information to the stream.
    - If the object is `Externalizable`, the writeExternal method of the object is called.
    - If the object is neither Serializable or Externalizable, the NotSerializableException is thrown.

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 non-static and non-transient fields of the current class to the stream. If called from outside the `writeObject` method the NotActiveException is thrown.

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 having already 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, 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.

The `replaceObject` method is used by trusted subclasses to allow objects within the graph to be replaced or monitored during serialization. Replacing objects must explicitly be enabled 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. A subclass's implementation 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 replaced by the substitute object.

When objects are being replaced the subclass must insure 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 is used 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 to do replacement can be trusted. Every reference to objects is passed to `replaceObject`. To insure that the private state of objects is not unintentionally exposed only trusted streams may use `replaceObject`. Trusted classes are those classes with a class loader equals null.

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.

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 blockdata 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.

2.2 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 saving 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.

2.3 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;
```

Object Input Interfaces 3

Topics:

- The ObjectInputStream Class
- The ObjectInputValidation Interface
- The readObject Method
- The readExternal 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 final void defaultReadObject()
        throws IOException, ClassNotFoundException,
            NotActiveException;

    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 final 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;
}
```

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 a blockdata record occurs in the stream, throw a BlockDataException with the number of available bytes.
2. If the object in the stream is null, return null.
3. If the object in the stream is a handle to a previous object, return the object.
4. 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 return the String.
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 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, assign it to the array, and return the array.

8. 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.

9. An instance of the class is allocated. The instance and its handle are added to the set of known objects. The contents restored appropriately:
   - For Serializable objects, the no-arg constructor for the non-serializable supertype is run and then each class's fields are restored by calling class specific `readObject` methods or if not defined the `defaultReadObject` method is called. 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.
   - 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.

10. If previously enabled by `enableResolveObject` the `resolveObject` method is called just before the object is returned. This allows subclasses to replace it if desired. The value of the call to `resolveObject` is returned from `readObject`.

All of the methods for reading primitives types only consume bytes from the blockdata 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 blockdata 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 in above.

The `defaultReadObject` method is used to read the fields of an object from the stream. It uses the class descriptor in the stream to read the fields 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 `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, 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.

The `resolveObject` method is used by trusted subclasses to enable the monitoring or substitution of one object for another during deserialization. Resolving objects must explicitly be enabled 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. A subclass's implementation 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 to the original object or else a `ClassCastException` is thrown. All assignments are type checked. All references in the stream to the original object will replaced by references to the substitute object.

The `enableResolveObject` method is used 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 that the stream requesting to do replacement can be trusted. Every reference to deserialized objects is passed to the resolveObject method. To insure 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 equals null.

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

3.2 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 ObjectInvalidException;
}
```

3.3 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;
```

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 class's readObject method, if implemented, is responsible for restoring the state of the class. The `defaultReadObject` method should be called before reading any optional data written by the corresponding `writeObject` method. If an attempt is made 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.

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

3.4 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.

Class Descriptors 4

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 streamVersionUID 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 long getSerialVersionUID();

    public String toString();
}
```

The `lookup` method returns the `ObjectStreamClass` descriptor for the specified class in the Java VM. If the class has defined serialVersionUID it is retrieved from the class. If not defined by the class it is computed from the class's definition in the Java Virtual Machine. NULL is returned if the specified class is not Serializable or Externalizable. Only class descriptions for classes that implement the java.io.Serializable or java.io.Externalizable interfaces can be written to a stream.

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.

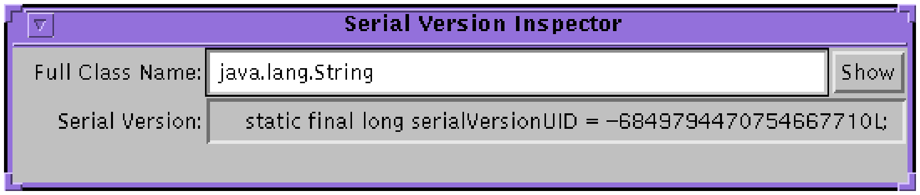
The `forClass` method returns the Class in the local Virtual Machine if one is known. Otherwise, it returns null.

The `getSerialVersionUID` method returns the serialVersionUID of this class. Refer to the Stream Unique Identifiers below. 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 Standard.

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

4.1 *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 -show it puts up a simple user interface. To find if a class is serializable and its serialVersionUID enter its full class name and press either <enter> or the Show button. The string printed can be copied and pasted into the evolved class.



Full Class Name: java.lang.String

Serial Version: static final long serialVersionUID = -6849794470754667710L;

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

4.2 *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:

```
    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. Classes do not need to anticipate versioning.

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 compute a signature for the stream. The first two 32-bit quantities 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 item in the stream is as follows:

1. The class name written using UTF encoding
2. The class modifiers written as a 32-bit integer
3. The name of each interface sorted by name written using UTF encoding.
4. For each field of the class sorted by field name except private static and private transient fields.
   a. The name of the field in UTF encoding
   b. The modifiers of the field written as an 32-bit integer
   c. The descriptor of the field in UTF encoding
5. For each method including constructors sorted by method name and signature, except private methods and constructors.
   a. The name of the method in UTF encoding
   b. The modifiers of the method written as an 32-bit integer
   c. The descriptor of the method in UTF encoding
6. The SHA-1 algorithm is executed on the stream of bytes produced by DataOutputStream and produces 5 32-bit values sha[0..4].
7. The hash value is assembled from the first and second 32 bits values.
   long hash = sha[1] << 32 + sha[0];

Versioning of Serializable Objects 5

Topics:

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 identity of a class, including what constitutes a compatible change, that is a change that does not affect the contract between the class and its callers.

This note 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

- Support bidirectional communication between different versions of a class operating in different virtual machines.
  - Define a mechanism that allows Java classes to read streams written by older versions of the same class.
  - Define 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.
- For RMI to use serialization it must perform well and produce compact streams in simple cases.
- Be able to identify and load classes that match the exact class used to write the stream.
- Keep the overhead low for non-versioned 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.

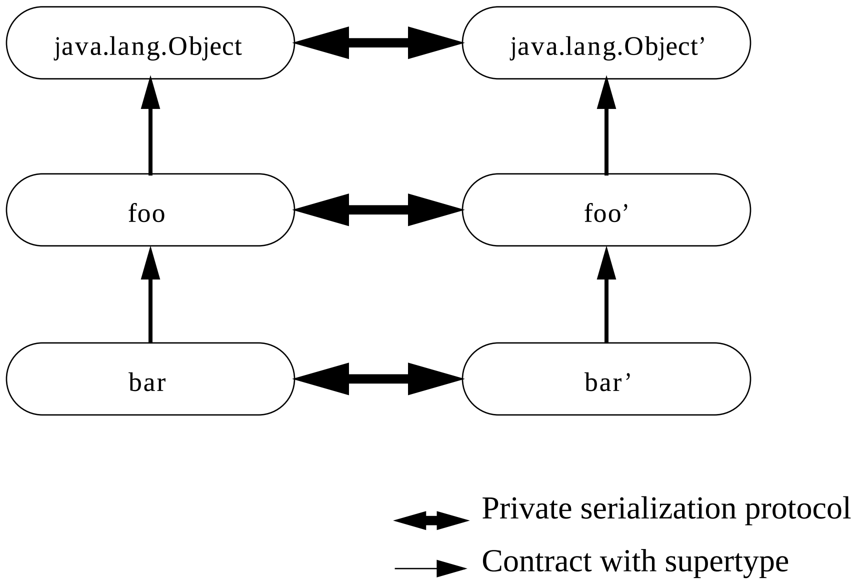
5.3 Assumptions

- 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 do 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 non-evolved class. This takes two forms. First, it must itself 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 it must provide sufficient and equivalent information to allow the earlier version to continue to satisfy the non-evolved 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

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, etc.

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 non-static and non-transient 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 non-static and non-transient 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 otherwise 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 non-static field to static or a non-transient field to transient - This 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 in the deleting a field case above, the field of the earlier version will be initialized to the default value but that 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.

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 non-default 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 is fields will be initialized to default values. The support for subclassing non-serializable 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 NotSerializableException is thrown.
- Removing java.io.Serializable so that it is no longer Serializable - This is equivalent to removing the class, and it can be dealt with by reading and discarding data for the class.
- 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 non-static or transient to non-transient - This 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:

- Overview
- Stream Elements
- Grammar for the Stream Format
- Example

6.1 Overview

The stream format is designed to satisfy the following goals:

- Be compact and structured for efficient reading.
- Allow skipping through the stream using only the knowledge of the structure and format of the stream. Do not require any per class code to be invoked.
- Require 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 zero. The handles restart at zero when the stream is reset.

A class object is represented by:

- its ObjectStreamClass object.

An ObjectStreamClass object is represented by:

- The Stream Unique Identifier (SUID) of compatible classes.
- A flag indicating if the class had writeObject/readObject methods.
- The number of non-static and non-transient 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.

Arrays are represented by

- Their ObjectStreamClass object.
- The number of elements.
- The sequence of values (the type of the values is implicit in the type of the array. e.g. the values of a byte array are of type byte).

New objects in the stream are represented by:

- 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 default serialized fields (those fields not marked static or transient as described in the corresponding ObjectStreamClass.

- 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 whether the data is written to the stream within a `writeObject` method or written directly to the stream from outside a `writeObject` method. This data may only be read by the corresponding `readObject` methods or directly from the stream. Objects written by `writeObject` terminate any previous block-data record and are written 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 Grammar for the Stream Format

The table below contains the grammar. Non-terminal symbols are show in italics, Terminal symbols in a fixed width font. Definitions of non-terminals are followed by a ":". The definition is followed by one or more alternatives each on a separate line. The notation in the table is as follows:

- *(datatype)* This token has the data type specified, e.g. (byte).
- token[n] A predefined number of occurrences of the token, i.e. 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.

6.3.1 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

newClass:
    TC_CLASS classDesc newHandle

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

superClassDesc:
    classDesc

newClassDesc:
    TC_CLASSDESC className serialVersionUIDnewHandle classDescInfo

classDescInfo:
    classDescFlags fields classAnnotation superClassDesc

className:
    (utf)

serialVersionUID:
    (long)

classDescFlags:
    (byte)

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

```
fieldDesc:
    primitiveDesc
    objectDesc

primitiveDesc:
    prim_typecode fieldName modifiers

objectDesc:
    obj_typecode fieldName modifiers className

fieldName:
    (utf)

modifiers:
    (short)

className:
    (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_WRRD_METHOD & !classDescFlags
    wrclass objectAnnotation     // SC_WRRD_METHOD & classDescFlags

nowrclass:
    values                       // fields in order of class descriptor

wrclass:
    nowrclass

objectAnnotation:
    endBlockData
    contents endBlockData        // contents written by writeObject

blockdata:
    TC_BLOCKDATA (byte)<size> (byte)[size]

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

endBlockData:
    TC_ENDBLOCKDATA

newString:
    TC_STRING newHandle (utf)

prevObject:
    TC_REFERENCE (int)handle

nullReference:
    TC_NULL

exception:
    TC_EXCEPTION  (Throwable)object// A java.lang.Throwable object

resetContext:
    TC_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

6.3.2 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 byte SC_WRRD_METHODS = 0x01;
final static byte SC_SERIALIZABLE = 0x02;
final static byte SC_EXTERNALIZABLE = 0x04;
```

6.4 Example

Suppose an original class and two instances in a linked list:

```
class List {
    int value;
    List next;
    public static void main(String[] args) {
        list list1 = new list();
        list list2 = new list();
        list1.value = 17;
        list1.next = list2;
        list2.value = 19;
        list2.next = null;

        ObjectOutputStream out = ...;
        out.writeObject(list1);
        out.writeObject(list2);
}
```

The resulting stream contains:

```
00: ac ed 00 05 73 72 00 04 4c 69 73 74 2f 0b 17 f6 >....sr..List/...<
10: 5a 0f bc a7 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 Byte Stream

A.1 Overview

The object serialization system allows a bytestream to be produced from a graph of objects, sent out of the Java environment (e.g. 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 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 appendix is to address these security concerns.

A.2 Design Goals

The goal for object serialization is to have the simplest serialization system 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, for example file handles, are just 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 were 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 one is 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 Byte Stream

Another way of protecting a bytestream outside the Java 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`.

- `ObjectStreamException` - superclass of all serialization exceptions
- `InvalidClassException` - when a class cannot be used to restore objects.
  - The class does not match the serial version of the class in the stream
  - The class contains fields with invalid primitive data types.
  - The class is not public.
  - The class does not have an accessible no-arg constructor.
- `NotSerializableException` - thrown by a `readObject` or `writeObject` method to terminate serialization or deserialization.
- `StreamCorruptedException` - thrown when the stream header is invalid or when control information in the stream is not found or found to be invalid.
- `NotActiveException` - thrown if `registerValidation` is not called during `readObject`.
- `InvalidObjectException` - thrown when a restored object cannot be made valid.
- `OptionalDataException` - thrown by readObject when there is primitive data in the stream when an object is expected. The length field of the exception supplied 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.