Java Serialization

by Andrew Downs

Adding object persistence to Java applications

Introduction

This article discusses and demonstrates how to incorporate object persistence into a Java application using the serialization mechanism in Java 1.1. This article assumes a general familiarity with Java. The code in this article was developed using the Apple Macintosh Runtime for Java (MRJ) version 2.0 and the MRJ SDK.

Overview

Serialization involves saving the current state of an object to a stream, and restoring an equivalent object from that stream. The stream functions as a container for the object. Its contents include a partial representation of the object's internal structure, including variable types, names, and values. The container may be transient (RAM-based) or persistent (disk-based). A transient container may be used to prepare an object for transmission from one computer to another. A persistent container, such as a file on disk, allows storage of the object after the current session is finished. In both cases the information stored in the container can later be used to construct an equivalent object containing the same data as the original. The example code in this article will focus on persistence.

Since Java applets do not have direct access to a local disk, it may be impossible for an applet to find a suitable container for persistent storage of a serialized object. Therefore, the code in this article focuses on Java applications.

Implementation

For an object to be serialized, it must be an instance of a class that implements either the Serializable or Externalizable interface. Both interfaces only permit the saving of data associated with an object's variables. They depend on the class definition being available to the Java Virtual Machine at reconstruction time in order to construct the object.

The Serializable interface relies on the Java runtime default mechanism to save an object's state. Writing an object is done via the writeObject() method in the ObjectOutputStream class (or the ObjectOutput interface). Writing a primitive value may be done through the appropriate write<datatype>() method. Reading the serialized object is accomplished using the readObject() method of the ObjectInputStream class, and primitives may be read using the various read<datatype>() methods.

What about other objects that may be referred to by the object we are serializing? For instance, what if our object is a Frame containing a set of (AWT) Panel and TextArea instance variables? Using the Serializable interface, these references (and their associated data) also are converted and written to the stream. All state information necessary to reconstruct our Frame object and any objects that it references gets stored together.

If those other objects or their formats weren't stored, our reconstructed Frame would contain null object references, and the content of those Panels and TextAreas would be gone. Plus, any methods that rely on the existence of the Panels or TextAreas would throw exceptions.

The Externalizable interface specifies that the implementing class will handle the serialization on its own, instead of relying on the default runtime mechanism. This includes which fields get written (and read), and in what order. The class must define a writeExternal() method to write out the stream, and a corresponding readExternal() method to read the stream. Inside of these methods the class calls ObjectOutputStream writeObject(), ObjectInputStream readObject(), and any necessary write<datatype>() and read<datatype>() methods, for the desired fields.

Hiding Data

Sometimes you may wish to prevent certain fields from being stored in the serialized object. The Serializable interface allows the implementing class to specify that some of its fields do not get saved or restored. This is accomplished by placing the keyword transient before the data type in the variable declaration. For example, you may have some data which is confidential and can be re-read from a master file later (as opposed to saving it with the serialized object). Or you decide (wisely) to preserve the privacy of file references by declaring any such variables as transient. Otherwise, all fields automatically get written without any additional effort by the class.

In addition to those fields declared as transient, static fields are not serialized (written out), and so cannot be deserialized (read back in).

Another way to use Serializable, and control which fields get written, is to override the writeObject() method of the Serializable interface. Inside of this method, you are responsible for writing out the appropriate fields. If you take this approach, you will want to override readObject() as well, to control the restoration process. This is similar to using Externalizable, except that interface requires writeExternal() and readExternal().

For the Externalizable interface, since both writeExternal() and readExternal() must be declared public, this increases the risk that a rogue object could use them to determine the format of the serialized object. For this reason, you should be careful when saving object data with this interface.

It is worth considering the amount of security you need for any objects that you serialize. When reading them back in, all of the normal Java security checks (such as the bytecode verifier) are in effect. You can define certain values within the class that should remain intact in serialized objects. Perhaps they should contain a specific value, or a value within a particular range. You can easily check the value of any numeric variable read in from a serialized object, especially if you know that only a portion of the available range for that data type is used by your variable.

You can also encrypt the outgoing data stream. The implementation is up to you, and don't forget to decrypt the object format when reading it back in.

Versioning

The ability to save and restore objects leads to an interesting question: what happens when an object has been stored for so long, that upon restoration it finds that its format has been superceded by a new, different version of the class?

The stream reading the serialized representation is responsible for accounting for any differences. The intent is that a newer version of a Java class should be able to interoperate with older representations of the same class, as long as there have not been certain changes in the class structure. The same does not necessarily hold true for an older version of the class, which may not be able to effectively deal with a newer representation.

So, we need some way to determine at runtime (or more appropriately, deserialization-time) whether we have the necessary backward compatibility.

In Java 1.1, changes to classes may be specified using a version number. A specific class variable, serialVersionUID (representing the Stream Unique Identifier, or SUID), may be used to specify the earliest version of the class that can be deserialized. The SUID is declared as follows:

```
  static final long serialVersionUID = 2L;
```

This particular declaration and assignment specifies that version 2 is as far back as this class can go. It is not compatible with an object written by version 1 of the class, and it cannot write a version 1 object. If it encounters a version 1 object in a stream (such as when restoring from a file), an InvalidClassException will be thrown.

The SUID is a measure of backward compatibility. The same SUID can be used for multiple representations of a class, as long as newer versions can still read the older versions.

If you do not explicitly assign a SUID, a default value will be assigned when the object gets serialized. This default SUID is a hash, or unique numeric value, which is computed using the class name, interfaces, methods, and fields. The exact algorithm is defined by the Secure Hash Algorithm (SHA). Refer to the Sun Java documentation for details.

The JDK (MRJ) utility program serialver will display the default (hash) SUID for a class. You can then paste this value in any subsequent, compatible versions of the class. (It is not required in the initial version of the class.) As of this writing the serialver program has not been included in the MRJ SDK, but hopefully will be in the future.

How can you obtain the SUID for a class at runtime to determine compatibility? First, query the Virtual Machine for information about the class represented in the stream, using methods of the class ObjectStreamClass. Here is how we can get the SUID of the current version of the class named MyClass, as known to the Virtual Machine:

```
ObjectStreamClass myObject = ObjectStreamClass.lookup(
Class.forName( "MyClass" ) );
long theSUID = myObject.getSerialVersionUID();
```

Now when we restore an Externalizable object, we can compare its SUID to the class SUID just obtained. If there is a mismatch, we should take appropriate action. This may involve telling the user that we cannot handle the restoration, or we may have to assign and use some default values.

If we are restoring a Serializable object, the runtime will check the SUID for us when it attempts to read values from the stream. If you override readObject(), you will want to compare the SUIDs there.

How do you determine what changes between class versions are acceptable? For an earlier version, which may contain fewer fields, trying to read a serialized object from a later version of the same class may cause problems. There is a tendency to add fields to a class as that class evolves, which means that the earlier version does not know about the newer fields. In contrast, since a newer version of a class may look for fields that are not present in the older version, it assigns default values to those fields.

This can be seen in the example code when we add a new field to the MyVersionObject class, but don't update the SUID. The new class can still read the older stream representation, even though no values exist in that stream for the new fields. It assigns 0 to the new int, and null to the new String, but doesn't throw any exceptions. If we then increment the SUID (from 1 to 2) to indicate that we do not consider older class versions compatible with this version, we throw an InvalidClassException when attempting to read a version 1 object from the stream.

The Sun documentation lists the various class format changes that can adversely affect the restoration of an object. A few of these include:

- Deleting a field, or changing it from non-static or non-transient to static or transient, respectively.
- Changing the position of classes in a hierarchy.
- Changing the data type of a primitive field.
- Changing the interface for a class from Serializable to Externalizable (or vice-versa).

On the other hand, not every change will have a negative effect. Here are some changes to class versions that do not have a detrimental effect on object behavior:

- Adding fields, which will result in default values (based on data type) being assigned to the new fields upon restoration.
- Adding classes will still allow an object of the added class to be created, since the class structure information is included in the stream. However, its fields will be set to the default values.
- Adding or removing the writeObject() or readObject() methods.
- Changing the access modifier (public, private, etc.) for a field, since it is still possible to assign a value to the field.
- Changing a field from static or transient to to non-static or non-transient, respectively.

Format of a Serialized Object

The format for the default structure of a serialized object is similar, but not identical, to the structure of a class file. The Sun documentation describes in detail the format of the Object Serialization Stream. The example code writes files that may be opened with a text editor, so you can inspect the serialized objects.

Example Code

The following code illustrates the writing and reading of Serializable and Externalizable classes. ObjectReaderWriter is the primary application class. At runtime it displays a "Save As..." FileDialog, allowing you to specify an output file to receive the stream containing the serialized objects. (All the sample objects are written to the same file.) It then prompts for an input file from which to read a stream.

This arrangement of the sample code allows you to write out the serialized data to one file, make changes to the class format for one or more of the data classes, recompile and rerun, and attempt to read one of the older versions back in.

The class MySerialObject contains a reference to an instance of the class MyInternalObject, to demonstrate the saving of nested object references in the stream. MySerialObject also contains a field (of type int) that is marked transient, and upon restoration you will find that the default value 0 gets assigned to that variable.

The class MyVersionObject demonstrates the use of versioning with a programmer-specified SUID. You only need to change the SUID when you make changes to the class structure that render it incompatible with older versions of that same class, and whose serialized instances have previously been written to disk.

You can compile the .java (source) files using the javac (Java compiler) tool included in the MRJ SDK Tools folder, or using the Java compiler in CodeWarrior or Visual Cafe. You can then optionally create a .jar (Java archive) file containing the resulting .class (output) files.

The archive for this article includes the .java and .class files, and a .jar file containing the .class files. To run the program, drag either the file ObjectReaderWriter.class or ObjectReaderWriter.jar onto the JBindery application icon, which is located in the MRJ SDK JBindery folder. Once JBindery launches, it will display ObjectReaderWriter in the "class name" field. (This field specifies the name of the class to run at application startup; that class must contain a main() method.) Click OK to run the program.

Listing 1: ObjectReaderWriter.java

ObjectReaderWriter.java
The class that will read and write serialized and externalized objects.

```
import java.awt.*;
import java.io.*;

public class ObjectReaderWriter {
  String filePath;

  public static void main( String args[] ) {
    ObjectReaderWriter orw = new ObjectReaderWriter();
  }

  ObjectReaderWriter() {
    try {
      //  Create instances of each data class to be serialized.
      MySerialObject serialObject = new MySerialObject();

      MyExternObject externObject = new MyExternObject();

      MyVersionObject versionObject = new MyVersionObject();

      //  Allow the user to specify an output file.
      FileDialog fd = new FileDialog( new Frame(),
        "Save As...", FileDialog.SAVE );
      fd.show();
      filePath = new String( fd.getDirectory() + fd.getFile() );

      //  Create a stream for writing.
      FileOutputStream fos = new FileOutputStream( filePath );

      //  Next, create an object that can write to that file.
      ObjectOutputStream outStream =
        new ObjectOutputStream( fos );

      //  Save each object.
      outStream.writeObject( serialObject );

      externObject.writeExternal( outStream );
      outStream.writeObject( versionObject );

      //  Finally, we call the flush() method for our object, which
          forces the data to
      //  get written to the stream:
      outStream.flush();

      //  Allow the user to specify an input file.
      fd = new FileDialog( new Frame(), "Open...",
          FileDialog.LOAD );
      fd.show();
      filePath = new String( fd.getDirectory() + fd.getFile() );

      //  Create a stream for reading.
      FileInputStream fis = new FileInputStream( filePath );

      //  Next, create an object that can read from that file.
      ObjectInputStream inStream = new ObjectInputStream( fis );

      // Retrieve the Serializable object.
      serialObject = ( MySerialObject )inStream.readObject();

      //  Display what we retrieved:
      System.out.println( serialObject.getS() );
      System.out.println( "i = " + serialObject.getI() );
      serialObject.displayInternalObjectAttrs();

      // Retrieve the Externalizable object.
      externObject.readExternal( inStream );

      //  Display what we retrieved:
      System.out.println( externObject.getS() );
      System.out.println( "i = " + externObject.getI() );

      // Retrieve the versioned object.
      versionObject = ( MyVersionObject )
        inStream.readObject();
      //  Display what we retrieved:
      System.out.println( versionObject.getS() );
      System.out.println( "i = " + versionObject.getI() );

      // Display the SUID of the versioned class in the VM,
      // not necessarily the serialized object.
      ObjectStreamClass myObject = ObjectStreamClass.lookup(
          Class.forName( "MyVersionObject" ) );
      long theSUID = myObject.getSerialVersionUID();

      System.out.println
        ( "The SUID of class MyVersionObject = " + theSUID );
    }
    catch ( InvalidClassException e ) {
      System.out.println( "InvalidClassException..." );
    }
    catch ( ClassNotFoundException e ) {
      System.out.println( "ClassNotFoundException..." );
    }
    catch ( OptionalDataException e ) {
      System.out.println( "OptionalDataException..." );
    }
    catch ( FileNotFoundException e ) {
      System.out.println( "FileNotFoundException..." );
    }
```

```
    catch ( IOException e ) {
      System.out.println( "IOException..." );
    }
  }
}
```

Listing 2: MySerialObject.java

MySerialObject.java
The serializable data class.

```
import java.io.*;

public class MySerialObject implements Serializable {
  private transient int i;
  private String s;
  MyInternalObject mio;

  MySerialObject() {
    i = 64;
    s = new String( "Instance of MySerialObject..." );
    mio = new MyInternalObject();
  }

  public int getI() {
    return i;
  }

  public String getS() {
    return s;
  }

  public void displayInternalObjectAttrs() {
    System.out.println( mio.getS() );
    System.out.println( "i = " + mio.getI() );
  }
}
```

Listing 3: MyInternalObject.java

MyInternalObject.java
The nested data class.

```
import java.io.*;

public class MyInternalObject implements Serializable {
  private int i;
  private String s;

  MyInternalObject() {
    i = 128;
    s = new String( "Instance of MyInternalObject..." );
  }
  public int getI() {
    return i;
  }

  public String getS() {
    return s;
  }
}
```

Listing 4: MyExternObject.java

MyExternObject.java
The externalizable data class.

```
import java.io.*;

public class MyExternObject implements Externalizable {
  private int i;
  private String s;

  MyExternObject() {
    i = 256;
    s = new String( "Instance of MyExternObject..." );
  }

  public int getI() {
    return i;
  }

  public String getS() {
    return s;
  }

  public void writeExternal( ObjectOutput out ) throws
      IOException {
    out.writeInt( this.i );
    out.writeObject( this.s );
  }

  public void readExternal( ObjectInput in ) throws
      IOException, ClassNotFoundException {
    this.i = in.readInt();
    this.s = ( String )in.readObject();
  }
}
```

Listing 5: MyVersionObject.java

MyVersionObject.java
The versioned data class.

```
import java.io.*;

public class MyVersionObject implements Serializable {
  static final long serialVersionUID = 1L;
  private int i;
  private String s;

  //  Uncomment the next two lines to verify that default values will be substituted
if
  //  the value is not present in the stream at deserialization time.
  //  private int i2 = -1; private String s2 = "This is the new String field";

  MyVersionObject() {
    i = 512;
    s = new String( "Instance of MyVersionObject..." );
  }
  public int getI() {
    return i;
  }

  public String getS() {
    return s;
  }
}
```

Conclusion

Adding object persistence to Java applications using serialization is easy. Serialization allows you to save the current state of an object to a container, typically a file. At some later time, you can retrieve the saved data values and create an equivalent object. Depending on which interface you implement, you can choose to have the object and all its referenced objects saved and restored automatically, or you can specify which fields should be saved and restored. Java also provides several ways of protecting sensitive data in a serialized object, so objects loaded from a serialized representation should prove no less secure than those classes loaded at application startup. Versioning provides a measure of the backward compatibility of class versions. The code needed to add serialization to your application is simple and flexible.

References

Developing Java Beans, Robert Englander, O'Reilly & Associates, Inc., 1997.

URLs

- http://www.javasoft.com/products/jdk/1.1/docs/guide/serialization/index.html.
- http://www.apple.com/macos/java/.

Andrew Downs is a Senior Software Engineer for Template Software in New Orleans, LA, designing and building enterprise apps. He also teaches C and Java programming at Tulane University College. Andrew wrote the Macintosh freeware program Recent Additions, and the Java application UDPing. You can reach him at andrew.downs@template.com.