The Wonders of Java Object Serialization

by Brian T. Kurotsuchi

Introduction

With the release of Java 1.1, the Java community has gained access to a wide variety of features that were not available in Java 1.0. Some of the important features include remote method invocation (RMI), Java database connectivity (JDBC) (or more specifically the java.sql package [2]), inner classes (nested classes), reflection, Java native interfaces (JNI), JavaBeans, internationalization, JAR files, and object serialization. With all of these new features, the possibilities for Java programs expand significantly compared to the ``old days" of 1.0. This article will attempt to render a good explanation and tutorial through one of these new features: *object serialization*.

Object Serialization in 25 Words or Less

Quite simply, object serialization provides a program the ability to read or write a whole object to and from a raw byte stream. It allows Java objects and primitives to be encoded into a byte stream suitable for streaming to some type of network or to a file-system, or more generally, to a transmission medium or storage facility.

First Thoughts

At first glance, serialization may sound like a very powerful feature or a fairly insignificant one. Serialization of objects does not sound very profound on the outside, but when one considers the tasks that could be made easier and the features that can be added to the Java world with serialization, the implications can be interesting.

The real power of serialization is the ability of Java programs to easily read and write entire objects and primitive data types, without converting to/from raw bytes or parsing clumsy text data. Anyone who has taken a modern programming class taught in a programming language such as Pascal or one of the C derivatives, has surely had the opportunity to do I/O on a text file for the purpose of storing and loading data. The parsing these files always seems to reduce to a basic set of hurdles that you must get over: Did I reach the EOF marker? Is the file pointer on top of the integer or the string? What happens when I reach the EOLN mark, and what if the marker is missing? Now consider a situation where a programming language suddenly made these problems go away? Object serialization has taken a step in the direction of being able to store *objects* instead of reading and writing their state in some foreign and possibly unfriendly format.

Java Serialization from 35,000 feet

Viewed from the ease-of-use standpoint, serializing an object requires only that it meets one of two criteria. The class must either implement the `Serializable` interface (`java.io.Serializable`) which has no methods that you need to write **or** the class must implement the `Externalizable` interface which defines two methods. As long as you do not have any special requirements, making a serializable is as simple as adding the ``implements Serializable" clause. See the section titled *Controlling the Serialization* for information about these ``special requirements".

Java Serialization at the Treetops

Any object that may be subject to serialization must, at a minimum, implement the `Serializable` interface. This interface does not define any methods, but rather serves as an indication that the developer of the class has considered the effects that serialization would have on the object. Special care must be taken if the state of the object would not be appropriate to send over the network or written to a storage medium. See the section titled *Controlling the Serialization* for details on these special cases. The process of serializing an object involves traversing the graph [3] created by each object's references to other objects and primitives. Since an object's state usually consists of information stored in internal data, most objects will contain references to other objects which will need to be preserved if the original object is to regain its state when it is de-serialized.

Example: Angry Object Salad

The source code for these examples is available here

For our first example lets create a simple program that writes objects containing various data-types, and then reads them back in. The example consists of the these classes (related as shown in Figure 1):

ObjectSaladWriter
: A Java *application* that accepts a filename to write serialized data to. It then fills that file with both primitive and object data. The order that the data is written in is hard coded.

ObjectSaladReader
: A Java *application* that accepts a filename to read serialized data from, it reads the data in an order that is hard coded, and prints out the data that is read. There is a fair amount of exception handling in the code which means that we will be able to see any problems that may arise with old versions of the data, casting errors, I/O exceptions etc.

SaladComponent
: An abstract superclass of some of the components we will be serializing. We will see how this superclass interacts with the concrete subclasses.

Crouton and Tomato
: These are the classes of the two types of objects that `ObjectSaladWriter` will be writing and `ObjectSaladReader` will be re-constituting. They are both subclasses of `SaladComponent`.

TomatoSeed
: As you can see in Figure 1, this is not a subclass of a `SaladComponent`, and it is not *directly* serialized by the code we write.

The `ObjectSaladWriter` will perform the following operations: open a file stream capable of writing objects, create and write some primitives and objects, and finally, close down the streams and exit. The job of the `ObjectSaladReader` is to read the file that the `ObjectSaladWriter` created and attempt to recreate and use the resulting primitives and objects. If everything goes right there will be no error messages and the objects will print out lots of information about when they were created and some of the state that the ObjectSaladWriter provided to them. Lets examine the parts of the code that are relevant to serialization. At this point it might be a good idea to go download the source code and browse through the `ObjectSaladWriter` class.

Setting up the Streams for Serialization

Serializing to a file requires some special preparation above and beyond simply opening a file for writing. Specifically, to serialize items to a file, you will need to perform the following actions:

- Produce a valid filename to write to.
- Open a FileOutputStream .
- Attach an ObjectOutputStream to the FileOutputStream.

Examine the following code to see how this is done:

(ObjectSaladReader.java)

```
File theFile;                            // simply a file name
FileOutputStream outStream;              // generic stream to the file
ObjectOutputStream objStream;            // stream for objects to the file

// ... error checking on the args to main() ...

theFile = new File(args[0]);

// ... error checking on the filename ...

try {
   // setup a stream to a physical file on the filesystem
   outStream = new FileOutputStream(theFile);

   // attach a stream capable of writing objects to the stream that is
   // connected to the file
   objStream = new ObjectOutputStream(outStream);

   // ... other activities ...
} catch(IOException e) {
   System.err.println("Things not going as planned.");
   e.printStackTrace();
}    // catch
```

Initiating the Serialization

Preparation of the `ObjectOutputStream` is as simple as using its constructor to attach it to a class derived from `java.io.OutputStream`. The next step is to do the actual writing of objects to the object stream. The public interface to `ObjectOutputStream` provides a wide variety of methods for writing all types of data.

(ObjectSaladWriter.java)

```
// write some primitive data
objStream.writeInt(3);

// write some objects
objStream.writeObject(new Crouton(7));
objStream.writeObject(new Tomato("Mars", 11, 5))
```

When all the writing to the stream has been completed, do not forget to flush the streams and close it down. Now it is time to see if we can read the data that we wrote. The following code reads the serialized file that was written by the `ObjectSaladWriter`:

(ObjectSaladReader.java)

```
// read the primitive data
int primitive = objStream.readInt();
```

```
// read the objects
Crouton crunch = (Crouton)objStream.readObject();
Object tomato = objStream.readObject();

// close down the streams
objStream.close();
inStream.close();
```

Notice that the writing and reading was done in the same order. The storage medium is *serial*, which means that things are exactly in the same place that you left them. Try playing with the order of things and see what happens.

We have now successfully serialized and then de-serialized the following types (see figure 2 also):

- Primitives (int, float).
- An object directly (`Tomato, Crouton`).
- An object indirectly, because it was referenced by an object that we did serialize (inside Tomato and `Crouton`).
- A collection of primitives in an array (in `Crouton`).
- A collection of objects in a `Vector` (in `Tomato`).

In addition, all the knowledge that these objects are part of a class hierarchy is preserved. Notice how we can use polymorphic features after re-constitution:

(ObjectSaladReader.java):

```
SaladComponent crunch = (SaladComponent)objStream.readObject();
// some other code here, then...
crunch.showSelf();
```

The object still knows it is really a Crouton, but can still legally act like a SaladComponent. So we know that the computer is not confused about what kind of object it really is, and hopefully you are not either.

Compiling and running the example

You will need source for all six of the above classes and a Java 1.1 compatible compiler and runtime to make the examples work. In addition, the second example uses the MD5 facility that Sun provides, you may need to modify the code slightly to use a different implementation. Make sure that your classpath (however you do it on your platform) includes the directory that you unpacked the files into, which is probably directory dot `.'. Compile all the source files.

To run the program, invoke your Java interpreter on `ObjectSaladWriter` and as a parameter, supply a file name (relative or fully qualified). This will serialize the example data into the file that you specified. Next, invoke the interpreter again, but on `ObjectSaladReader` instead, and provide it with the same file name. It should print out about 25 lines of information, which should be the state that the `ObjectSaladWriter` provided to the salad objects when it wrote them to disk.

You can tweak the source files and see what happens, as we will discuss below. Remember that Java uses dynamic run-time binding, so you only need to recompile files which you change. This is a sample of the steps, as performed on a UNIX box running `tcsh` and using Sun Microsystems' Java Developer Kit (JDK):

```
~/src/serialwonder% setenv CLASSPATH .
```

```
~/src/serialwonder% javac *.java
~/src/serialwonder% java ObjectSaladWriter junk_data
~/src/serialwonder% java ObjectSaladReader junk_data
[...about 25 lines of output]
```

"What if?" Cases

There are a few situations that we should play around with to gain a better understanding of the behavior of serialization. This primarily has to do with exception conditions -- when things go seriously wrong. Try making the order of serialization and re-constitution different between the writer and reader. See what happens when you get the types mixed up when casting and when assigning from the `ObjectInputStream`.

If you change the `ObjectSaladWriter` so that two primitives are written and then the two objects, the reader will produce a java.io.OptionalDataException because it is still expecting a single integer primitive and then two objects. Trying to cast a de-serialized object to the wrong type will cause a `java.lang.ClassCastException` when you attempt to do the cast. Remember that the `readObject()` method of the ObjectInputStream only knows that it is reading something of `Object` type.

What should happen if we remove the ``implements Serializable" from the `TomatoSeed` class? Recall that serializing a `Tomato` (which contains `TomatoSeeds`) produces a graph of all objects that each individual object has references to. Any situation in which an object needs to be serialized, but does not implement `Serializable` or `Externalizable`, will result in a `java.io.NotSerializableException` exception.

Try making some modification to a class that has been serialized. If you alter and recompile the `Tomato` class for instance, and then run the `ObjectSaladReader` (without running `ObjectSaladWriter`), you will get a `java.io.InvalidClassException`. The serialization mechanism complains because the class that was originally serialized has changed format in the current runtime environment. Whether or not **you** think it is compatible, the software must assume that it is not. Look in the *Versioning* section of this article for more information.

Transient Data

An important modifier that your can use for object data is the `transient` [4] keyword. If a field in a class is marked as transient, it will automatically be skipped when serializing any objects that are instances of that class. This is very useful for references that are ``relative to an address space" [1] they were created in, such as stream objects, file references and network connections. This is a key attribute of a class that should be considered when attaching the `Serialization` or `Externalizable` interface to a class you are writing.

Controlling the Serialization

As with all general-purpose features, you must give the programmer freedom to customize. So far, we have always let the serialization be performed automatically by the code in the Object input/output stream classes, which is great for most applications. However, there are cases where you will want to control which fields of your object are serialized and which parts are not. In addition to the `Serializable` interface that we have been working with so far, the `java.io` package includes another interface called `Externalizable`.

The `Externalizable` interface is meant to be used when you need to define an object which has complete control over its serialization and re-constitution process. This ``control" consists of controlling the encoding used to send the information, which fields are serialized, and you have the option of adding any additional information you feel necessary to include in the serialized stream. Some interesting custom objects that you could create might do encryption of some or all of the serialized information, or maybe include a message digest like MD5 to check for tampering, and possibly compression to decrease bandwidth or storage requirements.

An `Externalizable` object must implement two methods which have the following signatures (return type and throws clause are not technically part of the signature [4], but you should follow the form laid out in the interface):

```
public void writeExternal(ObjectOutput) throws IOException
public void readExternal(ObjectInput) throws IOException, ClassNotFoundException
```

Implementing these methods is basically the same thing that we just did to serialize our objects that conformed to the `Serializable` interface. Recall that we used methods like `writeObject()` and `writeInt()` to make the object serialize itself. To implement `writeExternal()`, you must manually ``write'' to the `ObjectOutput` object, each piece of data that you wish to save. Keep in mind that nothing is done automatically for you, so the developer is responsible for ensuring that all superclass and subclass information is serialized as necessary.

The code to manually de-serialize your object is just a matter of undoing what your `writeExternal()` did. The `ObjectInput` has methods which are almost symmetric to those found in the `ObjectOutput` interface.

Example: Sensitive Information

Lets look at an example of implementing the `Externalizable` interface. Consider a situation where you have information from a UNIX password file, including the encrypted password itself. Your task is to make these objects serializable so they can be sent over a network or written to disk. The one restriction that you have is that you cannot let the encrypted password leave the computer's runtime environment.

Each user's information (passwd file entry) is stored in a object based on the `UserInfo` class. You must add serialization capabilities to this object, and since we need to protect the password, the `Serializable` interface will not be acceptable. The `UserInfo` object has the following data members:

```
private String login;
private int UID;
private String passwd;
private String homeDir;
```

You need to implement the `Externalizable` interface, and the first step in doing that is to write the `writeExternal()` method. Here is a possible implementation of it:

(UserInfo.java)

```
public void writeExternal(ObjectOutput out) throws IOException {
        byte[] digest = null;
   // ...compute MD5 digest in the byte array named ``digest''

   // write fields
   out.writeObject(login);
   out.writeInt(UID);
   out.writeObject(homeDir);
   // write the digest length and digest itself
   // need the length to read it in later
   out.writeInt(digest.length);
   System.out.println("Digest is "+digest.length+" bytes.");
   out.write(digest);
}
```

Next you need to re-constitute the object from the `ObjectInput`:

(UserInfo.java)

```
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
   int md5Len;

   // read fields
   this.login = (String)in.readObject();
   this.UID = in.readInt();
   this.homeDir = (String)in.readObject();

   // read the digest length, allocate byte array
   md5Len = in.readInt();
   digest = new byte[md5Len];

   // read the MD5 data
   for(int count = 0; count < md5Len; count++) {
      digest[count] = in.readByte();
   }   // for
   // ...check the MD5
}
```

One thing to remember about de-serializing an object is that the class must have a public constructor that takes no arguments. This is to allow the serialization mechanism to create an empty object and then fill in the data.

Versioning

Every piece of software, even a Java program, is subject to the standard life-cycle of software. This cycle includes the ongoing activity we call *maintenance* [6]. Maintenance includes bug fixes, adaptation to new environments, enhancement, and preventive maintenance. Considering that we now have objects that are persistent, there are now implications to changing the class definition. What is the effect of removing an instance variable from the class definition? If we add an instance variable and then want to de-serialize an older version of the class that did not contain that information, what state will that leave the object in? These are all issues that affect the programmer's ability to make updates to classes while still maintaining backwards compatibility. See the specification [1] for a discussion of the effect of the various types of class changes.

The serialization mechanism uses an identification value to keep track of all classes. This identifier, called the serialVersionUID (SUID), is computed from the structure of the class file to form a unique 64-bit identifier. It is possible to explicitly indicate which serialVersionUID your class is compatible with. In plain English, you can tell the serialization mechanism which version of your class you are compatible with.

The Solaris and Windows JDKs from Sun Microsystems provide a program called `serialver` that will compute the SUID for any given class file. Once you have this version number, you can include it in subsequent versions of the class to indicate that the new version is compatible with the version identified by the SUID. The following is the syntax used to declare the SUID that you are compatible with:

```
static final long serialVersionUID = 7110175216435025451L;
```

If you include this in the class data, the serialization mechanism will understand which past version of the class you intend to be compatible with the current implementation.

Applications

There are already numerous projects in progress that will take advantage of the serialization process, two of which have resulted in Java 1.1 features. One is remote method invocation (RMI) [5] where objects seem to be transparently transported over the network and reconstituted for use on a machine other than the one it originally was created on. RMI is the rough equivalent of the remote procedure call(RPC) [8], done ``object-oriented" style. In effect, with minimal code overhead, you can invoke methods on objects which exist on a machine which your program has access to via a network. Object serialization plays a key role in being able to pass arguments and return values over the network in a seemingly transparent manner.

A second application of serialization appears in the implementation of JavaBeans [7]. The designers wanted it to be easy and intuitive to have Beans save their state and then be recalled with that state at a later date. For this reason, all Beans must be serializable, implementing either the `Serializable` or `Externalizable` interface.

Conclusion

Object serialization is not a technology that will impress an audience during a presentation, nor will it boost product sales all by itself. Serialization is a building block that will allow the creation of potentially earth-shaking technologies for Java. We have seen some of these technologies, like JavaBeans and RMI, surface already. Clearly this technology has the potential to make things happen; it is up to us to use our ingenuity to create new applications for it. At least ask your professors to give you a class file, some JavaDoc, and a serialized object instead of an ugly text file.

References

1

Object Serialization Specification Sun Microsystems Inc. 1996. http://java.sun.com/products/jdk/1.1/docs/guide/serialization/spec/serialTOC.doc.html

2

Java Platform 1.1.3 Core API Sun Microsystems Inc. 1996. http://java.sun.com/products/jdk/1.1/docs/api/packages.html

3

Johnsonbaugh, Richard *Discrete Mathematics, Fourth Edition* Prentice Hall, Upper Saddle River, New Jersey 1997.

4

Gosling, James, Bill Joy, and Guy Steele. *The Java Language Specification* Addison-Wesley, Reading, Massachusetts, 1996.

5

Remote Method Invocation Specification. Sun Microsystems Inc. 1996. http://java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html

6
Pressman, Roger S. *Software Engineering: A Practitioner's Approach, Third Edition*, McGraw-Hill Inc., New York, New York 1982.

7
JavaBeans API Specification Sun Microsystems Inc. 1996. http://java.sun.com/beans/spec.html

8
Tanenbaum, Andrew S. *Modern Operating Systems* Prentice Hall, Upper Saddle River, New Jersey 1992.

Brian Kurotsuchi is a senior at *California Polytechnic State University, San Luis Obispo*, working towards his undergraduate degree in Computer Science. He is a member of the USENIX Association and the ACM. Within the field of Computer Science, his interests include operating systems, object-oriented design patterns and software engineering. His experience with Java includes work with servlets, threading and JDBC.