Application Development Framework
Application Express
Big Data
Business Intelligence
Cloud Computing
Communications
Database Performance & Availability
Data Warehousing
Database
.NET
Dynamic Scripting Languages
Embedded
Digital Experience
Enterprise Architecture
Enterprise Management
Identity & Security
Java
Linux
Mobile
Service-Oriented Architecture
Solaris
SQL & PL/SQL
Systems - All Articles
Virtualization

Discover the secrets of the Java Serialization API

by Todd Greanier; Reprinted from JavaWorld

Published July 2000

We all know the Java platform allows us to create reusable objects in memory. However, all of those objects exist only as long as the Java virtual machine[1] remains running. It would be nice if the objects we create could exist beyond the lifetime of the virtual machine, wouldn't it? Well, with object serialization, you can flatten your objects and reuse them in powerful ways.

Object serialization is the process of saving an object's state to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time. The Java Serialization API provides a standard mechanism for developers to handle object serialization. The API is small and easy to use, provided the classes and methods are understood.

Throughout this article, we'll examine how to persist your Java objects, starting with the basics and proceeding to the more advanced concepts. We'll learn three different ways to perform serialization -- using the default protocol, customizing the default protocol, and creating our own protocol -- and we'll investigate concerns that arise with any persistence scheme such as object caching, version control, and performance issues.

By the conclusion of this article, you should have a solid comprehension of that powerful yet sometimes poorly understood Java API.

First Things First: The Default Mechanism

Let's start with the basics. To persist an object in Java, we must have a persistent object. An object is marked serializable by implementing the `java.io.Serializable` interface, which signifies to the underlying API that the object can be flattened into bytes and subsequently inflated in the future.

Let's look at a persistent class we'll use to demonstrate the serialization mechanism:

```
10 import java.io.Serializable;
20 import java.util.Date;
30 import java.util.Calendar;
40 public class PersistentTime implements Serializable
50 {
60 private Date time;
70
80 public PersistentTime()
90 {
100       time = Calendar.getInstance().getTime();
110     }
120
130     public Date getTime()
140     {
150       return time;
160     }
170   }
```

As you can see, the only thing we had to do differently from creating a normal class is implement the `java.io.Serializable` interface on line 40. The completely empty `Serializable` is only a *marker* interface -- it simply allows the serialization mechanism to verify that the class is able to be persisted. Thus, we turn to the first rule of serialization:

Rule #1: The object to be persisted must implement the `Serializable` interface or inherit that implementation from its object hierarchy.

The next step is to actually persist the object. That is done with the `java.io.ObjectOutputStream` class. That class is a *filter stream*--it is wrapped around a lower-level byte stream (called a *node stream*) to handle the serialization protocol for us. Node streams can be used to write to file systems or even across sockets. That means we could easily transfer a flattened object across a network wire and have it be rebuilt on the other side!

Take a look at the code used to save the `PersistentTime` object:

```
10 import java.io.ObjectOutputStream;
20 import java.io.FileOutputStream;
30 import java.io.IOException;
40 public class FlattenTime
50 {
60 public static void main(String [] args)
70 {
80 String filename = "time.ser";
90 if(args.length > 0)
100      {
110        filename = args[0];
120      }
130      PersistentTime time = new PersistentTime();
140      FileOutputStream fos = null;
150      ObjectOutputStream out = null;
160      try
170      {
180        fos = new FileOutputStream(filename);
190        out = new ObjectOutputStream(fos);
200        out.writeObject(time);
210        out.close();
220      }
230      catch(IOException ex)
240      {
250        ex.printStackTrace();
260      }
270    }
280 }
```

The real work happens on line 200 when we call the `ObjectOutputStream.writeObject()` method, which kicks off the serialization mechanism and the object is flattened (in that case to a file).

To restore the file, we can employ the following code:

```
10 import java.io.ObjectInputStream;
20 import java.io.FileInputStream;
30 import java.io.IOException;
40 import java.util.Calendar;
50 public class InflateTime
60 {
70 public static void main(String [] args)
80 {
90 String filename = "time.ser";
100     if(args.length > 0)
110     {
120       filename = args[0];
130     }
140   PersistentTime time = null;
150   FileInputStream fis = null;
160   ObjectInputStream in = null;
170   try
180   {
190     fis = new FileInputStream(filename);
200     in = new ObjectInputStream(fis);
210     time = (PersistentTime)in.readObject();
220     in.close();
230   }
240   catch(IOException ex)
250   {
260     ex.printStackTrace();
270   }
280   catch(ClassNotFoundException ex)
290   {
300     ex.printStackTrace();
310   }
320   // print out restored time
330   System.out.println("Flattened time: " + time.getTime());
340   System.out.println();
350      // print out the current time
360   System.out.println("Current time: " + Calendar.getInstance().getTime());
370 }
380}
```

In the code above, the object's restoration occurs on line 210 with the `ObjectInputStream.readObject()` method call. The method call reads in the raw bytes that we previously persisted and creates a live object that is an exact replica of the original. Because `readObject()` can read any serializable object, a cast to the correct type is required. With that in mind, the class file must be accessible from the system in which the restoration occurs. In other words, the object's class file and methods are not saved; only the object's *state* is saved.

Later, on line 360, we simply call the `getTime()` method to retrieve the time that the original object flattened. The flatten time is compared to the current time to demonstrate that the mechanism indeed worked as expected.

Nonserializable Objects
The basic mechanism of Java serialization is simple to use, but there are some more things to know. As mentioned before, only objects marked `Serializable` can be persisted. The `java.lang.Object` class does not implement that interface. Therefore, not all the objects in Java can be persisted automatically. The good news is that most of them -- like AWT and Swing GUI components, strings, and arrays -- are serializable.

On the other hand, certain system-level classes such as `Thread`, `OutputStream` and its subclasses, and `Socket` are not serializable. Indeed, it would not make any sense if they were. For example, thread running in my JVM would be using my system's memory. Persisting it and trying to run it in your JVM would make no sense at all. Another important point about `java.lang.Object` not implementing the `Serializable` interface is that any class you create that extends only `Object` (and no other serializable classes) is not serializable unless you implement the interface yourself (as done with the previous example).

That situation presents a problem: what if we have a class that contains an instance of `Thread`? In that case, can we ever persist objects of that type? The answer is yes, as long as we tell the serialization mechanism our intentions by marking our class's `Thread` object as `transient`.

Let's assume we want to create a class that performs an animation. I will not actually provide the animation code here, but here is the class we'll use:

```
10 import java.io.Serializable;
20 public class PersistentAnimation implements Serializable, Runnable
30 {
40 transient private Thread animator;
50 private int animationSpeed;
60 public PersistentAnimation(int animationSpeed)
70 {
80 this.animationSpeed = animationSpeed;
90 animator = new Thread(this);
100     animator.start();
110   }
120       public void run()
130   {
140     while(true)
150     {
160       // do animation here
170     }
180   }
190 }
```

When we create an instance of the `PersistentAnimation` class, the thread `animator` will be created and started as we expect. We've marked the thread on line 40 `transient` to tell the serialization mechanism that the field should not be saved along with the rest of that object's state (in that case, the field `speed`). The bottom line: you must mark `transient` any field that either cannot be serialized or any field you do not want serialized.

Serialization does not care about access modifiers such as `private` -- all nontransient fields are considered part of an object's persistent state and are eligible for persistence.

Therefore, we have another rule to add. Here are both rules concerning persistent objects:

Rule #1: The object to be persisted must implement the `Serializable` interface or inherit that implementation from its object hierarchy
Rule #2: The object to be persisted must mark all nonserializable fields `transient`

Customize the Default Protocol

Let's move on to the second way to perform serialization: customize the default protocol. Though the animation code above demonstrates how a thread could be included as part of an object while still making that object be serializable, there is a major problem with it if we recall how Java creates objects. To wit, when we create an object with the `new` keyword, the object's constructor is called only when a new instance of a class is created. Keeping that basic fact in mind, let's revisit our animation code. First, we instantiate an object of type `PersistentAnimation`, which begins the animation thread sequence. Next, we serialize the object with that code:

```
PersistentAnimation animation = new PersistentAnimation(10);
FileOutputStream fos = ...
ObjectOutputStream out = new ObjectOutputStream(fos);
out.writeObject(animation);
```

All seems fine until we read the object back in with a call to the `readObject()` method. Remember, a constructor is called only when a new instance is created. We are not creating a new instance here, we are restoring a persisted object. The end result is the animation object will work only once, when it is first instantiated. Kind of makes it useless to persist it, huh?

Well, there is good news. We can make our object work the way we want it to; we can make the animation restart upon restoration of the object. To accomplish that, we could, for example, create a `startAnimation()` helper method that does what the constructor currently does. We could then call that method from the constructor, after which we read the object back in. Not bad, but it introduces more complexity. Now, anyone who wants to use that animation object will have to know that method has to be called following the normal deserialization process. That does not make for a seamless mechanism, something the Java Serialization API promises developers.

There is, however, a strange yet crafty solution. By using a built-in feature of the serialization mechanism, developers can enhance the normal process by providing two methods inside their class files. Those methods are:

```
private void writeObject(ObjectOutputStream out) throws IOException;
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;
```

Notice that both methods are (and must be) declared `private`, proving that neither method is inherited and overridden or overloaded. The trick here is that the virtual machine will automatically check to see if either method is declared during the corresponding method call. The virtual machine can call private methods of your class whenever it wants but no other objects can. Thus, the integrity of the class is maintained and the serialization protocol can continue to work as normal. The serialization protocol is always used the same way, by calling either `ObjectOutputStream.writeObject()` or `ObjectInputStream.readObject()`. So, even though those specialized private methods are provided, the object serialization works the same way as far as any calling object is concerned.

Considering all that, let's look at a revised version of `PersistentAnimation` that includes those private methods to allow us to have control over the deserialization process, giving us a pseudo-constructor:

```
import java.io.Serializable;
public class PersistentAnimation implements Serializable, Runnable
{
transient private Thread animator;
private int animationSpeed;
public PersistentAnimation(int animationSpeed)
{
this.animationSpeed = animationSpeed;
startAnimation();
   }
       public void run()
   {
     while(true)
     {
       // do animation here
     }
   }
   private void writeObject(ObjectOutputStream out) throws IOException
   {
     out.defaultWriteObject();
   }
   private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException
   {
     // our "pseudo-constructor"
     in.defaultReadObject();
     // now we are a "live" object again, so let's run rebuild and start
     startAnimation();

   }
   private void startAnimation()
   {
     animator = new Thread(this);
     animator.start();
   }
 }
```

Notice the first line of each of the new private methods. Those calls do what they sound like -- they perform the default writing and reading of the flattened object, which is important because we are not replacing the normal process, we are only adding to it. Those methods work because the call to `ObjectOutputStream.writeObject()` kicks off the serialization protocol. First, the object is checked to ensure it implements `Serializable` and then it is checked to see whether either of those private methods are provided. If they are provided, the stream class is passed as the parameter, giving the code control over its usage.

Those private methods can be used for any customization you need to make to the serialization process. Encryption could be added to the output and decryption to the input (note that the bytes are written and read in cleartext with no obfuscation at all). They could be used to add extra data to the stream, perhaps a company versioning code. The possibilities are truly limitless.

Stop That Serialization!
OK, we have seen quite a bit about the serialization process, now let's see some more. What if you create a class whose superclass is serializable but you do not want that new class to be serializable? You cannot unimplement an interface, so if your superclass does implement `Serializable`,

your new class implements it, too (assuming both rules listed above are met). To stop the automatic serialization, you can once again use the private methods to just throw the `NotSerializableException`. Here is how that would be done:

```
private void writeObject(ObjectOutputStream out) throws IOException
{
throw new NotSerializableException("Not today!");
}
private void readObject(ObjectInputStream in) throws IOException
{
throw new NotSerializableException("Not today!");
}
```

Any attempt to write or read that object will now always result in the exception being thrown. Remember, since those methods are declared `private`, nobody could modify your code without the source code available to them -- no overriding of those methods would be allowed by Java.

Create Your Own Protocol: the Externalizable Interface

Our discussion would be incomplete not to mention the third option for serialization: create your own protocol with the `Externalizable` interface. Instead of implementing the `Serializable` interface, you can implement `Externalizable`, which contains two methods:

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

Just override those methods to provide your own protocol. Unlike the previous two serialization variations, nothing is provided for free here, though. That is, the protocol is entirely in your hands. Although it's the more difficult scenario, it's also the most controllable. An example situation for that alternate type of serialization: read and write PDF files with a Java application. If you know how to write and read PDF (the sequence of bytes required), you could provide the PDF-specific protocol in the `writeExternal` and `readExternal` methods.

Just as before, though, there is no difference in how a class that implements `Externalizable` is used. Just call `writeObject()` or `readObject` and, voila, those externalizable methods will be called automatically.

Gotchas

There are a few things about the serialization protocol that can seem very strange to developers who are not aware. Of course, that is the purpose of the article -- to get you aware! So let's discuss a few of those gotchas and see if we can understand why they exist and how to handle them.

Caching Objects in the Stream

First, consider the situation in which an object is written to a stream and then written again later. By default, an `ObjectOutputStream` will maintain a reference to an object written to it. That means that if the state of the written object is written and then written again, the new state will not be saved! Here is a code snippet that shows that problem in action:

```
ObjectOutputStream out = new ObjectOutputStream(...);
MyObject obj = new MyObject(); // must be Serializable
obj.setState(100);
out.writeObject(obj); // saves object with state = 100
obj.setState(200);
out.writeObject(obj); // does not save new object state
```

There are two ways to control that situation. First, you could make sure to always close the stream after a write call, ensuring the new object is written out each time. Second, you could call the `ObjectOutputStream.reset()` method, which would tell the stream to release the cache of references it is holding so all new write calls will actually be written. Be careful, though -- the `reset` flushes the entire object cache, so all objects that have been written could be rewritten.

Version Control

With our second gotcha, imagine you create a class, instantiate it, and write it out to an object stream. That flattened object sits in the file system for some time. Meanwhile, you update the class file, perhaps adding a new field. What happens when you try to read in the flattened object?

Well, the bad news is that an exception will be thrown -- specifically, the `java.io.InvalidClassException` -- because all persistent-capable classes are automatically given a unique identifier. If the identifier of the class does not equal the identifier of the flattened object, the exception will be thrown. However, if you really think about it, why should it be thrown just because I added a field? Couldn't the field just be set to its default value and then written out next time?

Yes, but it takes a little code manipulation. The identifier that is part of all classes is maintained in a field called `serialVersionUID`. If you wish to control versioning, you simply have to provide the `serialVersionUID` field manually and ensure it is always the same, no matter what changes you make to the classfile. You can use a utility that comes with the JDK distribution called `serialver` to see what that code would be by default (it is just the hash code of the object by default).

Here is an example of using `serialver` with a class called `Baz`:

```
> serialver Baz
> Baz: static final long serialVersionUID = 10275539472837495L;
```

Simply copy the returned line with the version ID and paste it into your code. (On a Windows box, you can run that utility with the `– show` option to simplify the copy and paste procedure.) Now, if you make any changes to the `Baz` class file, just ensure that same version ID is specified and all will be well.

The version control works great as long as the changes are compatible. Compatible changes include adding or removing a method or a field. Incompatible changes include changing an object's hierarchy or removing the implementation of the `Serializable` interface. A complete list of compatible and incompatible changes is given in the Java Serialization Specification.

Performance Considerations

Our third gotcha: the default mechanism, although simple to use, is not the best performer. I wrote out a `Date` object to a file 1,000 times, repeating that procedure 100 times. The average time to write out the `Date` object was 115 milliseconds. I then manually wrote out the `Date` object, using standard I/O the same number of iterations; the average time was 52 milliseconds. Almost half the time! There is often a trade-off between convenience and performance, and serialization proves no different. If speed is the primary consideration for your application, you may want to consider building a custom protocol.

Another consideration concerns the aforementioned fact that object references are cached in the output stream. Due to that, the system may not garbage collect the objects written to a stream if the stream is not closed. The best move, as always with I/O, is to close the streams as soon as possible, following the write operations.

Conclusion

Serialization in Java is simple to instigate and almost as simple to implement. Understanding the three different ways of implementing serialization should aid in bending the API to your will. We have seen a lot of the serialization mechanism in that article, and I hope it made things clearer and not worse. The bottom line, as with all coding, is to maintain common sense within the bounds of API familiarity. That article has laid out a strong basis of understanding the Java Serialization API, but I recommend perusing the specification to discover more fine-grained details.

About the Author
Todd Greanier,*director of technology for ComTech Training, has been teaching and developing Java since it was introduced publicly. An expert in distributed Java technologies, he teaches classes in a wide range of topics, including JDBC, RMI, CORBA, UML, Swing, servlets/JSP, security, JavaBeans, Enterprise Java Beans, and multithreading. He also creates custom seminars for corporations, slanted to their specific needs. Todd lives in upstate New York with his wife, Stacey, and his cat, Bean.*

---

[1] As used on this web site, the terms "Java virtual machine" or "JVM" mean a virtual machine for the Java platform.

E-mail this page Printer View

Contact Us
US Sales: +1.800.633.0738
Global Contacts
Support Directory
Subscribe to Emails

About Oracle
Careers
Company Information
Social Responsibility
Communities

Downloads and Trials
Java Runtime Download
Java for Developers
Software Downloads
Try Oracle Cloud Free

News and Events
Acquisitions
Blogs
Events
Newsroom

© Oracle | Site Map | Terms of Use and Privacy | Cookie Preferences | Ad Choices