IBM

developerWorks Technical topics Java technology Technical library

5 things you didn't know about ... Java Object Serialization

You thought serialized data was safe? Think again.

Java Object Serialization is so fundamental to Java programming that it's easy to take for granted. But, like many aspects of the Java platform, Serialization rewards those who go digging. In his first article of this new series, Ted Neward gives you five reasons to look twice at the Java Object Serialization API, including tricks (and code) for refactoring, encrypting, and validating serialized data.

Share:

Ted Neward is the principal of Neward & Associates, where he consults, mentors, teaches, and presents on Java, .NET, XML Services, and other platforms. He resides near Seattle, Washington.

26 April 2010 (First published 06 April 2010)
Also available in Chinese Russian Japanese Portuguese

A few years ago, while working with a software team writing an application in the Java language, I experienced the benefit of knowing a little more than your average programmer about Java Object Serialization.

A year or so prior, a developer responsible for managing the application's per user settings had decided to store them in a `Hashtable`, then serialize the `Hashtable` down to disk for persistence. When a user changed his or her settings, the `Hashtable` was simply rewritten back to disk.

This was an elegant and open-ended settings system, but it fell apart when the team decided to migrate from `Hashtable` to `HashMap` from the Java Collections library.

The disk forms of `Hashtable` and `HashMap` are different and incompatible. Short of running some kind of data conversion utility over each of the persisted user settings (a monumental task), it seemed that `Hashtable` would be the application's storage format for the remainder of its lifetime.

About this series

So you think you know about Java programming? The fact is, most developers scratch the surface of the Java platform, learning just enough to get the job done. In this series, Ted Neward digs beneath the core functionality of the Java platform to uncover little known facts that could help you solve even the stickiest programming challenges.

The team felt stuck, but only because they didn't know something crucial (and somewhat obscure) about Java Serialization: it was built to allow for evolution of types over time. Once I showed them how to do automatic serialization replacement, the transition to `HashMap` proceeded as planned.

This article is the first in a series dedicated to uncovering useful trivia about the Java platform — obscure stuff that comes in handy for solving Java programming challenges.

Java Object Serialization is a great API to start with because it's been around since the beginning: JDK 1.1. The five things you'll learn about Serialization in this article should convince you to look twice at even standard Java APIs.

Java Serialization 101

Java Object Serialization, introduced as part of the ground-breaking feature set that made up JDK 1.1, serves as a mechanism to transform a graph of Java objects into an array of bytes for storage or transmission, such that said array of bytes can be later transformed back into a graph of Java objects.

In essence, the idea of Serialization is to "freeze" the object graph, move the graph around (to disk, across a network, whatever), and then "thaw" the graph back out again into usable Java objects. All this happens more or less magically, thanks to the `ObjectInputStream`/`ObjectOutputStream` classes, full-fidelity metadata, and the willingness of programmers to "opt in" to this process by tagging their classes with the `Serializable` marker interface.

Listing 1 shows a `Person` class implementing `Serializable`.

Listing 1. Serializable Person

```
package com.tedneward;

public class Person
    implements java.io.Serializable
{
    public Person(String fn, String ln, int a)
    {
        this.firstName = fn; this.lastName = ln; this.age = a;
    }

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public int getAge() { return age; }
    public Person getSpouse() { return spouse; }

    public void setFirstName(String value) { firstName = value; }
    public void setLastName(String value) { lastName = value; }
    public void setAge(int value) { age = value; }
    public void setSpouse(Person value) { spouse = value; }

    public String toString()
    {
        return "[Person: firstName=" + firstName +
            " lastName=" + lastName +
            " age=" + age +
            " spouse=" + spouse.getFirstName() +
            "]";
    }

    private String firstName;
    private String lastName;
    private int age;
    private Person spouse;

}
```

Once `Person` has been serialized, it's pretty simple to write an object graph to disk and read it back again, as demonstrated by this JUnit 4 unit test.

Listing 2. Deserializing Person

```
public class SerTest
{
    @Test public void serializeToDisk()
    {
        try
        {
            com.tedneward.Person ted = new com.tedneward.Person("Ted", "Neward", 39);
            com.tedneward.Person charl = new com.tedneward.Person("Charlotte",
                "Neward", 38);

            ted.setSpouse(charl); charl.setSpouse(ted);

            FileOutputStream fos = new FileOutputStream("tempdata.ser");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(ted);
            oos.close();
        }
        catch (Exception ex)
        {
            fail("Exception thrown during test: " + ex.toString());
        }

        try
        {
            FileInputStream fis = new FileInputStream("tempdata.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            com.tedneward.Person ted = (com.tedneward.Person) ois.readObject();
            ois.close();
```

```
            assertEquals(ted.getFirstName(), "Ted");
            assertEquals(ted.getSpouse().getFirstName(), "Charlotte");

            // Clean up the file
            new File("tempdata.ser").delete();
        }
        catch (Exception ex)
        {
            fail("Exception thrown during test: " + ex.toString());
        }
    }
}
```

Nothing you've seen so far is new or exciting — it's Serialization 101 — but it's a good place to start. We'll use `Person` to discover five things you probably *didn't* already know about *Java Object Serialization*.

1. Serialization allows for refactoring

Serialization permits a certain amount of class variation, such that even after refactoring, `ObjectInputStream` will still read it just fine.

The critical things that the *Java Object Serialization* specification can manage automatically are:

- Adding new fields to a class
- Changing the fields from static to nonstatic
- Changing the fields from transient to nontransient

Going the other way (from nonstatic to static or nontransient to transient) or deleting fields requires additional massaging, depending on the degree of backward compatibility you require.

Refactoring a serialized class

Knowing that Serialization allows for refactoring, let's see what happens when we decide to add a new field to the `Person` class.

`PersonV2`, shown in Listing 3, introduces a field for gender to the original `Person` class.

Listing 3. Adding a new field to serialized Person

```
enum Gender
{
    MALE, FEMALE
}

public class Person
    implements java.io.Serializable
{
    public Person(String fn, String ln, int a, Gender g)
    {
        this.firstName = fn; this.lastName = ln; this.age = a; this.gender = g;
    }

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public Gender getGender() { return gender; }
    public int getAge() { return age; }
    public Person getSpouse() { return spouse; }

    public void setFirstName(String value) { firstName = value; }
    public void setLastName(String value) { lastName = value; }
    public void setGender(Gender value) { gender = value; }
    public void setAge(int value) { age = value; }
    public void setSpouse(Person value) { spouse = value; }

    public String toString()
    {
        return "[Person: firstName=" + firstName +
            " lastName=" + lastName +
            " gender=" + gender +
            " age=" + age +
            " spouse=" + spouse.getFirstName() +
            "]";
    }

    private String firstName;
    private String lastName;
    private int age;
    private Person spouse;
```

```
    private Gender gender;
}
```

Serialization uses a calculated hash based on just about everything in a given source file — method names, field names, field types, access modifiers, you name it — and compares that hash value against the hash value in the serialized stream.

To convince the Java runtime that the two types are in fact the same, the second and subsequent versions of `Person` must have the same serialization version hash (stored as the private static final `serialVersionUID` field) as the first one. What we need, therefore, is the `serialVersionUID` field, which is calculated by running the JDK `serialver` command against the original (or V1) version of the `Person` class.

Once we have `Person`'s `serialVersionUID`, not only can we create `PersonV2` objects out of the original object's serialized data (where the new fields appear, they will default to whatever the default value is for a field, most often "null"), but the opposite is also true: we can deserialize original `Person` objects out of `PersonV2` data, with no added fuss.

2. Serialization is not secure

It often comes as an unpleasant surprise to Java developers that the Serialization binary format is fully documented and entirely reversible. In fact, just dumping the contents of the binary serialized stream to the console is sufficient to figure out what the class looks like and contains.

This has some disturbing implications *vis-a-vis* security. When making remote method calls via RMI, for example, any private fields in the objects being sent across the wire appear in the socket stream as almost plain-text, which clearly violates even the simplest security concerns.

Fortunately, Serialization gives us the ability to "hook" the serialization process and secure (or obscure) the field data both before serialization and after deserialization. We can do this by providing a `writeObject` method on a `Serializable` object.

Obscuring serialized data

Suppose the sensitive data in the `Person` class were the age field; after all, a lady never reveals her age and a gentleman never tells. We can obscure this data by rotating the bits once to the left before serialization, and then rotate them back after deserialization. (I'll leave it to you to develop a more secure algorithm, this one's just for example's sake.)

To "hook" the serialization process, we'll implement a `writeObject` method on `Person`; and to "hook" the deserialization process, we'll implement a `readObject` method on the same class. It's important to get the details right on both of these — if the access modifier, parameters, or name are at all different from what's shown in Listing 4, the code will silently fail, and our `Person`'s age will be visible to anyone who looks.

Listing 4. Obscuring serialized data

```
public class Person
    implements java.io.Serializable
{
    public Person(String fn, String ln, int a)
    {
        this.firstName = fn; this.lastName = ln; this.age = a;
    }

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public int getAge() { return age; }
```

```
    public Person getSpouse() { return spouse; }

    public void setFirstName(String value) { firstName = value; }
    public void setLastName(String value) { lastName = value; }
    public void setAge(int value) { age = value; }
    public void setSpouse(Person value) { spouse = value; }

    private void writeObject(java.io.ObjectOutputStream stream)
        throws java.io.IOException
    {
        // "Encrypt"/obscure the sensitive data
        age = age >> 2;
        stream.defaultWriteObject();
    }

    private void readObject(java.io.ObjectInputStream stream)
        throws java.io.IOException, ClassNotFoundException
    {
        stream.defaultReadObject();

        // "Decrypt"/de-obscure the sensitive data
        age = age << 2;
    }

    public String toString()
    {
        return "[Person: firstName=" + firstName +
            " lastName=" + lastName +
            " age=" + age +
            " spouse=" + (spouse!=null ? spouse.getFirstName() : "[null]") +
            "]";
    }

    private String firstName;
    private String lastName;
    private int age;
    private Person spouse;
}
```

If we need to see the obscured data, we can always just look at the serialized data stream/file. And, because the format is fully documented, it's possible to read the contents of the serialized stream without the class being available.

3. Serialized data can be signed and sealed

The previous tip assumes that you want to obscure serialized data, not encrypt it or ensure it hasn't been modified. Although cryptographic encryption and signature management are certainly possible using `writeObject` and `readObject`, there's a better way.

If you need to encrypt and sign an entire object, the simplest thing is to put it in a `javax.crypto.SealedObject` and/or `java.security.SignedObject` wrapper. Both are serializable, so wrapping your object in `SealedObject` creates a sort of "gift box" around the original object. You need a symmetric key to do the encryption, and the key must be managed independently. Likewise, you can use `SignedObject` for data verification, and again the symmetric key must be managed independently.

Together, these two objects let you seal and sign serialized data without having to stress about the details of digital signature verification or encryption. Neat, huh?

4. Serialization can put a proxy in your stream

From time to time, a class contains a core element of data from which the rest of the class's fields can be derived or retrieved. In those cases, serializing the entirety of the object is unnecessary. You could mark the fields *transient*, but the class would still have to explicitly produce code to check whether a field was initialized every time a method accessed it.

Given the principal concern is serialization, it's better to nominate a flyweight or proxy to go into the stream instead. Providing a `writeReplace` method on the original `Person` allows a different kind of

object to be serialized in its place; similarly, if a `readResolve` method is found during deserialization, it is called to supply a replacement object back to the caller.

Packing and unpacking the proxy

Together, the `writeReplace` and `readResolve` methods enable a `Person` class to pack a `PersonProxy` with all of its data (or some core subset of it), put it into a stream and then unwind the packing later when it is deserialized.

Listing 5. You complete me, I replace you

```
class PersonProxy
    implements java.io.Serializable
{
    public PersonProxy(Person orig)
    {
        data = orig.getFirstName() + "," + orig.getLastName() + "," + orig.getAge();
        if (orig.getSpouse() != null)
        {
            Person spouse = orig.getSpouse();
            data = data + "," + spouse.getFirstName() + "," + spouse.getLastName() + ","
              + spouse.getAge();
        }
    }

    public String data;
    private Object readResolve()
        throws java.io.ObjectStreamException
    {
        String[] pieces = data.split(",");
        Person result = new Person(pieces[0], pieces[1], Integer.parseInt(pieces[2]));
        if (pieces.length > 3)
        {
            result.setSpouse(new Person(pieces[3], pieces[4], Integer.parseInt
              (pieces[5])));
            result.getSpouse().setSpouse(result);
        }
        return result;
    }
}

public class Person
    implements java.io.Serializable
{
    public Person(String fn, String ln, int a)
    {
        this.firstName = fn; this.lastName = ln; this.age = a;
    }

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public int getAge() { return age; }
    public Person getSpouse() { return spouse; }

    private Object writeReplace()
        throws java.io.ObjectStreamException
    {
        return new PersonProxy(this);
    }

    public void setFirstName(String value) { firstName = value; }
    public void setLastName(String value) { lastName = value; }
    public void setAge(int value) { age = value; }
    public void setSpouse(Person value) { spouse = value; }

    public String toString()
    {
        return "[Person: firstName=" + firstName +
            " lastName=" + lastName +
            " age=" + age +
            " spouse=" + spouse.getFirstName() +
            "]";
    }

    private String firstName;
    private String lastName;
    private int age;
    private Person spouse;
}
```

Note that the `PersonProxy` has to track all of `Person`'s data. Often this means the proxy will need to be an inner class of `Person` to have access to private fields. The Proxy will also sometimes need to track down other object references and serialize them manually, such as `Person`'s spouse.

This trick is one of the few that isn't required to be read/write balanced. For instance, a version of a class that's been refactored into a different type could provide a `readResolve` method to silently transition a

serialized object over to a new type. Similarly, it could employ the `writeReplace` method to take old classes and serialize them into new versions.

5. Trust, but validate

It would be nice to assume that the data in the serialized stream is always the same data that was written to the stream originally. But, as a former President of the United States once pointed out, it's a safer policy to "trust, but verify."

In the case of serialized objects, this means validating the fields to ensure that they hold legitimate values after deserialization, "just in case." We can do this by implementing the `ObjectInputValidation` interface and overriding the `validateObject()` method. If something looks amiss when it is called, we throw an `InvalidObjectException`.

In conclusion

Java Object Serialization is more flexible than most Java developers realize, giving us ample opportunity to hack out of sticky situations.

Fortunately, coding gems like these are scattered all across the JVM. It's just a matter of knowing about them, and keeping them handy for when a brain-stumper presents itself.

Next up in the series: Java Collections. Until then, have fun twisting Serialization to your evil will!

Download

Description	Name	Size
Sample code for this article	5things1-src.zip	10KB

Resources

Learn

Develop and deploy your next app on the IBM Bluemix cloud platform.

Chat with Ted Neward: Ted Neward chats with dW on his new series and the Collections API.

"Test object serialization" (Elliotte Rusty Harold, IBM developerWorks, June 2006): Learn why it's important to test the serialized forms of objects, then try out various ways to test object serialization.

"Discover the secrets of the Java Serialization API" (Todd M. Greanier, JavaWorld, July 2000): An overview of the Java Serialization API, followed by three approaches to serializing Java objects.

"The Java Serialization algorithm revealed" (Sathiskumar Palaniappan, JavaWorld, May 2009): A closer look at the mechanics of the Java Serialization algorithm.

"Java Object Serialization: Download the Java Serialization spec as a PDF.

The developerWorks Java technology zone: Hundreds of articles about every aspect of Java programming.

Dig deeper into Java technology on developerWorks

Overview

New to Java programming

Technical library (tutorials and more)

Forums

Blogs

Communities

Downloads and products

Open source projects

Standards

Events

Bluemix Developers Community
Get samples, articles, product docs, and community resources to help build, deploy, and manage your cloud apps.

developerWorks Labs
Experiment with new directions in

Discuss

Get involved in the My developerWorks community.