



Chapter 4

Serialization

Java serialization allows you to take the state of a Java object and write it into a byte stream. From the stream, you can later create a second Java object that has a state that is equivalent to the state of the original. This facility allows you to persist an instance to a file, transfer an object to another machine on the network, move an object from one class loader to another, or re-create the state of an object against a newer version of the code. In short, serialization lets you make components that are mobile. Several high-level Java technologies build on the backbone of serialization, including Remote Method Invocation (RMI), JavaBeans, Enterprise JavaBeans (EJB), and JINI.

4.1 Serialization and Metadata

Metadata plays two critical roles in serialization. First, serialization relies on class metadata and reflection to extract the state of an instance. In the common case, reflection does all the heavy lifting, and developers do not have to write any per-class serialization code. Unfortunately, the simplicity of basic serialization is deceptive. Java is designed to support dynamic, changing systems, which makes serialization more complex than it appears at first brush. In a dynamic system, it is possible for the receiver of a serialized object to not have the appropriate class. If this is the case, the receiver will need to know what sort of class loader it needs to create to access the class.

What is even worse than not having the class at all is the possibility that the receiver may have a different version of the class. If this occurs, the serialization architecture allows the developer to salvage the information in the stream, even



if it does not correspond to the local definition of the class. Real applications may also need to customize an object's serialized format for performance, or to choose between pass-by-value and pass-by-reference semantics when they are passing objects from one virtual machine to another.

The second role that metadata plays is ensuring that the stream matches the receiver's expectations for a class. The serialization binary format defines its own metadata, which can be compared to the metadata in the binary class format. The serialization binary format makes it possible to recover an object's state, even if the code for a class is no longer available, or has changed. The serialization format also includes a number of hooks for customizing how objects are transmitted. This chapter will show you the basics of serialization, and then it will show you how to tweak serialization to improve performance, protect invariants, discover the location of necessary class files at runtime, and recover data if class files have changed since an object was serialized.

4.2 Serialization Basics

Not all Java classes are serializable. To indicate that you want a class to be serializable, you must implement the `java.io.Serializable` interface. This interface has no methods; rather, it simply acts as a marker interface indicating a class's willingness to have its instances serialized. In Listing 4-1, the `Person` class is serializable, but the `Humanoid` class is not. When an instance of `Person` is written to a stream, the serialization architecture uses reflection to extract the field values and write them to the stream.

Listing 4-1 The Humanoid and Person Classes

```
public abstract class Humanoid {
    protected int noOfHeads;
    private static int totalHeads;
    public Humanoid() {
        this(1);
    }
    public Humanoid(int noOfHeads) {
        if (noOfHeads > 10)
            throw new Error("Be serious. More than 10 heads?!");
        this.noOfHeads = noOfHeads;
        synchronized (Humanoid.class) {
```

```

        totalHeads += noOfHeads;
    }
}
public int getHeadCount() {
    return totalHeads;
}
}
import java.io.*;
public class Person extends Humanoid
    implements java.io.Serializable {
    private String lastName;
    private String firstName;
    private transient Thread workerThread;
    private static int population;
    public Person(String lastName, String firstName) {
        this.lastName = lastName;
        this.firstName = firstName;
        synchronized (Person.class) {
            population++;
        }
    }
    public String toString() {
        return "Person " + firstName + " " + lastName;
    }
    static synchronized public int getPopulation() {
        return population;
    }
}

```

Given that reflective access can work on *any* field of *any* object, there was no mechanical limitation that prevented the language designers from making all classes serializable. They did not do so for two reasons. First, some classes' instances contain resources that are local to a VM, process, or machine. Such classes wrap threads, files, sockets, database connections, and so on. There is no well-defined way to create a new instance from one of these objects that has state that is "equivalent" to the original object.

The second reason has to do with security. Some classes take the privacy of their private fields very seriously. Serialization provides a back door through which these fields can be accessed by writing them into a serialization stream and analyzing the stream's contents. Rather than force developers to

worry about these issues for every class, the Java language simply considers objects *not* serializable by default.

To write a serializable object to a stream, you need to create an instance of `java.io.ObjectOutputStream`. `ObjectOutputStream` is a wrapper stream; its constructor takes an `OutputStream` argument that will actually receive data. This wrapper architecture makes it easy to use the same `ObjectOutputStream` class, regardless of whether you are serializing an object to a file, socket, or other destination.

The `WriteInstance` class in Listing 4–2 writes a `Person` to a file specified on the command line. The meat of this example is the call to `os.writeObject(p)`. It is that simple. If the object referenced by `p` were not serializable, the call would throw a `java.io.NotSerializableException`. Since the object referenced by `p` is of a serializable type, the object's state is extracted via reflection and written to `oos`, which in turn writes the state to a file specified on the command line. To verify that this works, you can read the object back in later using `ReadInstance`. You should see output showing that the state "Julius Drabbih" was recovered correctly.

Listing 4–2 The `WriteInstance` and `ReadInstance` Classes

```
import java.io.*;
public class WriteInstance {
    public static void main(String [] args) throws Exception
    {
        if (args.length != 1) {
            System.out.println("usage: java WriteInstance file");
            System.exit(-1);
        }
        FileOutputStream fos = new FileOutputStream(args[0]);
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        Person p = new Person("Drabbih", "Julius");
        oos.writeObject(p);
    }
}

import java.io.*;
public class ReadInstance {
    public static void main(String [] args) throws Exception
    {
        if (args.length != 1) {
```

```

        System.out.println("usage: java ReadInstance filename");
        System.exit(-1);
    }
    FileInputStream fis = new FileInputStream(args[0]);
    ObjectInputStream ois = new ObjectInputStream(fis);
    Object o = ois.readObject();
    System.out.println("read object " + o);
}
}

```

4.2.1 Serialization Skips Some Fields

Serialization does not necessarily read and write *all* class fields to the stream. There are three exceptions, all of which can be seen in Listing 4–1.

1. Base class fields are only handled if the base class itself is serializable. This is in keeping with the idea that a class should only be serializable if the class's author specifically makes it so. In our example, this has the odd consequence of making our `Person` forget how many heads he has. The `noOfHeads` field is part of a nonserializable base class, so serialization ignores it.¹
2. Second, serialization ignores static fields because they are not part of any particular instance's state.² So, `Person`'s `population` field is not written to the stream.
3. You can use the `transient` keyword to disable serialization for specific fields. Normally, you will use `transient` if some of your class's instance fields do not have a logical serialized form. In the `Person` example, `workerThread` is marked `transient` because there is no standard notion of moving a thread from one VM or process to another. Note that even though `workerThread` is not written to the stream, it will still have a well-defined value of `null` when a `Person` is deserialized. When the Java virtual machine instantiates an object, object reference fields are initialized to `null`, numeric primitive fields to zero, and Boolean fields to `false`. This happens prior to any constructors or serialization code, guaranteeing that objects will have a well-defined initial state.

1. You can modify the default behavior and capture the fields from nonserializable base classes. (See §4.3).

2. Even if you wrote a static field during serialization, then what? When you read back in two different instances of a class, which one gets to set the one and only copy of the static field? Serialization doesn't happen for static fields because it does not have clear semantics.

4.2.2 Serialization and Class Constructors

An intriguing aspect of serialization is its relationship to constructors. It is an article of faith among Java programmers that Java objects are never created without a constructor invocation. This is essential for security because otherwise maleficent code could create invalid instances of core system classes in an attempt to destabilize the VM. Additionally, the promise that constructors will run is important to guarantee class invariants. Many constructors include checks to make sure that the object's state is valid, and later methods assume a valid state because they know that a constructor ran.

However sacred constructors may be, serialization does not always invoke them. You can verify this by adding a call to `System.out.println()` to the constructors for the `Humanoid` and `Person` classes used in Listing 4-1. If you do this and then rerun `WriteInstance` and `ReadInstance`, your session should look like Listing 4-3.

Listing 4-3 Deserialization Does Not Invoke Constructors

```
>java -cp classes WriteInstance Person.ser
Humanoid constructor ran
Person constructor called
>java -cp classes ReadInstance Person.ser
Humanoid constructor ran
read object Person Julius Drabbih
```

When `WriteInstance` uses `new` to create a `Person` object, the `Humanoid` and `Person` constructors both run, as expected. However, when `ReadInstance` reads the `Person` from the file, only the `Humanoid` constructor fires. What happened to `Person`'s constructor?

The answer is that serialization does not need to invoke `Person`'s constructor because it plans to assign `Person`'s fields from the stream anyway. Running a constructor for `Person` would be redundant at best. Moreover, notice that `Person` has no default constructor. How would deserialization synthesize arguments for a nondefault constructor?³ If there were more than one constructor, which one would deserialization choose? Serialization avoids these pitfalls by

3. In this case it would be easy, but in general it would not.

skipping the constructor step altogether. It is able to do this without using an invalid bytecode sequence because it creates the object from native code, where the bytecode rules do not apply.

Note that `Humanoid`'s constructor is still invoked because `Humanoid` is not serializable. Since serialization has no way to assign `Humanoid`'s fields, it relies on `Humanoid`'s default constructor. This implies one of the rules of serialization: If you mark a class `Serializable`, any nonserializable base classes of your class must have a default constructor.

In the simplest case, you would not worry about the various fields that serialization skips or the constructor behavior. This is because the goal of the architecture is to let developers simply mark their classes `Serializable` and then forget about them. However, the `Person` class demonstrates how quickly one can run afoul of the details. When a `Person` class is read from a serialized stream, there are two semantic problems that default serialization does not solve:

1. `Humanoid`'s default constructor sets the number of heads the `Person` has to one. This defect might not manifest as a bug for quite a long time since most people do in fact have only one head. However, if and when the bug occurs it would be very confusing.
2. `Person`'s `population` field is supposed to track the total number of people instantiated in a particular VM. Serialization bypasses `Person`'s constructor, so `population` is not incremented.

Both of these problems necessitate some ability to customize the serialization process. Most of the rest of this chapter will be spent looking at various customization hooks, starting with one that can be used to solve the `population` problem.

4.3 Using `readObject` and `writeObject`

If a class wishes to add some custom code that runs when an object is read from serialization, it can implement the `readObject` method:

```
private void readObject(ObjectInputStream stream)
    throws IOException, ClassNotFoundException
```

Before it reads an object's state from the stream, `ObjectInputStream` uses reflection to check to see if the object's class implements `readObject`. If it does, `ObjectInputStream` simply calls `readObject` instead of executing normal deserialization.

When you implement `readObject`, you normally do two things:

1. Call back to `ObjectInputStream` and ask it to read the fields as in normal serialization.
2. Execute any custom steps that you wish to add to the deserialization process.

Listing 4-4 shows how to use `readObject` to deal with the population problem.

Listing 4-4 Using `readObject` as a Deserialization Constructor

```
import java.io.*;
public class Person extends Humanoid
    implements java.io.Serializable {
    //repeat all code from original version of Person, plus:
    private void readObject(ObjectInputStream ois)
        throws IOException, ClassNotFoundException
    {
        ois.defaultReadObject();
        synchronized (Person.class) {
            population++;
        }
        System.out.println("Adjusting population in readObject");
    }
}
```

The call to `ois.defaultReadObject()` causes `ObjectInputStream` to read the field values from the stream and then to use reflection to assign the field values to the object. The call to increment `population` correctly tracks that a new instance of `Person` is alive in this VM.

You should add any per-instance validation code after the call to `defaultReadObject`. You should also use `readObject` to initialize any transient fields if Java's default values of `null/zero/false` are inappropriate for your object. Any invariant that you would check in a constructor should also be checked by `readObject`. In short, *treat `readObject` like a public constructor*.

4.4 Matching Streams with Classes

Serialization may involve loading classes if the instance being deserialized is of a type not already present in the virtual machine. The design for how classes are loaded is simple. The common case works without any special effort on the part of the programmer; the current class loader is simply asked to load the class whose name matches the class name in the serialization stream.

When deserialization triggers class loading, there are two problem cases to worry about. When the runtime attempts to load a class to deserialize an object, it might find a different version of the class than the one that serialized the object, or it might not find the class at all. This section will cover the former problem; the problem of finding classes is covered in §4.9.

Fortunately, the common case is handled trivially by the basic class loader architecture. When deserialization needs to load a class, it leverages implicit class loading; so, when `ReadInstance` needs to load `Person`, the virtual machine finds the class loader that loaded `ReadInstance` and tries to load `Person` with the same loader.

You can verify this using the custom version of `URLClassLoader` from §2.8.3. Listing 4–5 shows a run of `ReadInstance` with class loader logging turned on; note that the same loader loads `ReadInstance`, `Person`, and `Humanoid`. You might also try to read in the `Person` after deleting the `Person.class` file. If you do, deserialization will fail with a `ClassNotFoundException`. This proves that the default serialization mechanism does not embed the actual class file in the stream. If the receiver does not already have the binary class, it will not be able to deserialize the object.

Listing 4–5 Deserialization Leverages Implicit Class Loading.

```
{output clipped to show only relevant details}
>java -Xbootclasspath/p:boot/ -cp classes ReadInstance
>Person.ser
ReadInstance loaded by sun.misc.Launcher$AppClassLoader@ac738
Humanoid loaded by sun.misc.Launcher$AppClassLoader@ac738
Person loaded by sun.misc.Launcher$AppClassLoader@ac738
```

The serialized stream does not contain the entire binary format of the object's class. Nevertheless, there needs to be some way to detect if the class file

that the sender used is the same as the one the receiver is using. The classes might have different fields, different methods, or different semantics, and in any of these situations, the receiver may be unable to deserialize a valid instance. This could be a fatal flaw in the architecture since the problems caused by such an invalid instance might percolate to distant parts of the system and be difficult to track down.

4.4.1 The `serialVersionUID`

To avoid this problem, serialization sends a fingerprint as part of the class metadata in the serialized stream. This fingerprint takes the form of a 64-bit value called the stream unique identifier, or `serialVersionUID` (SUID). The run-time calculates a SUID for a class using several pieces of class metadata, including the class name, class modifiers, superinterfaces, and the signatures of most fields, constructors, and methods. All of this metadata is written into a `java.io.DataOutputStream`, which is then hashed using the Secure Hash Algorithm (SHA-1). (This is an abridged version of what happens; the exact details are in the serialization spec. See [Ser] for details.) The important point is this: Almost any change to a class, other than editing a method's implementation, will cause the class's SUID to change. If the SUID for a class does not match the SUID from the stream, then deserialization will fail.

You can retrieve the SUID of a class with the command-line tool `serialver` as follows:

```
>serialver Person
Person: static final long serialVersionUID =
3880321542343815834L;
```

Now, try adding a new field to the `Person` class as demonstrated here:

```
private int age;
```

If you attempt to read in the old version of `Person.ser` using this changed `Person` class, you will get an exception like the one shown here:

```
java.io.InvalidClassException: Person;
Local class incompatible:
stream classdesc serialVersionUID=3880321542343815834
local class serialVersionUID=8695226918703748744
```

The old version of the object does not have a value for `age`, and the `ObjectInputStream` would have no way to decide a reasonable value for `age`, so it rejects the attempt to deserialize a `Person`.

The last argument attributes too much intelligence to `ObjectInputStream`. The serialization architecture does not actually know that one version of the class had an `age` field and one did not; all it knows is that the SUIDs are different. This implies that even an innocuous change will break serialization. To see the problem, remove the `age` field from `Person.java`, and add the following method instead:

```
public void innocuousMethod() {}
```

This method does nothing at all; nevertheless, the SUID changes, and you can no longer read old versions of the class. The SUID is inexpensive, costing only 64 bits in the stream, but it is also a brute-force approach. From the perspective of the SUID, all changes are significant changes, and they all break serialization.⁴

It is interesting to compare serialization versioning to the class compatibility rules for class loading. When it is loading classes, the virtual machine uses the name and type information in the class file format to verify the linkage between classes. If there is a version mismatch, the error information can be quite precise, pinpointing the field or method that is missing. The SUID is another variant of this same idea, but it is compressed for efficient transmission. Because the metadata is hashed down to a single 64-bit value, serialization can only tell you that two classes are different—not what the difference is.

4.4.2 Overriding the Default SUID

If you make a change to a class that you know to be innocuous, you can assert its compatibility with older versions of the class by assigning an explicit `serialVersionUID` field. If you add a field with this signature

```
private static final long serialVersionUID = {somevalue}L;5
```

4. To be fair, there is a roughly 1 in 2^{64} chance that changing a class will *not* change the `serialVersionUID`, but don't hold your breath waiting for this.

5. The serialization specification states that the `serialVersionUID` field should be `private`, `static`, and `final`. However, the `serialver` tool omits the `private` keyword, and the implementation only verifies that the field is `static` and `final`.

to a class, the runtime will use this value instead of calculating the hash code. So, all you have to do to read a different version of a class is discover that class's SUID and set a `serialVersionUID` field to match. Discovering the original SUID is a snap because it is in the serialization stream, and it is also contained in the text of the `InvalidClassException` that is thrown when deserialization fails.

Armed with this information, you can create a new version of `Person` that is capable of reading the original version, as is shown in Listing 4–6. This version of `Person` has seen several changes from the original. The static and transient fields are gone, and the instance field `age` has been added. Nevertheless, this version of `Person` can be used to read the original `Person` from a stream because the matching `serialVersionUID` has been added.

Listing 4–6 Using an Explicit `serialVersionUID`

```
public class Person extends Humanoid
    implements java.io.Serializable {
    static final long serialVersionUID=3880321542343815834L;
    private String lastName;
    private String firstName;
    private int age; //this field is new to this version!
    public Person(String lastName, String firstName, int age) {
        this.lastName = lastName;
        this.firstName = firstName;
    }
    public String toString() {
        return "Person " + firstName + " " + lastName +
            " aged " + age;
    }
}
```

As soon as you deploy a second version of any serializable class, you will need to set the `serialVersionUID`. In fact, it is a good idea to manually set the `serialVersionUID` in the *first* version of a serializable class; you do this by running the `serialver` tool and pasting the result back into your source code. Calculating the SUID is expensive, and by setting it yourself, you can pay this cost once, at development time, instead of paying it the first time the class is serialized in each runtime.

4.4.3 Compatible and Incompatible Changes

Once you set the `serialVersionUID`, you are on your own to make sure that the old and new versions of the class are truly compatible. You have traded one problem for its opposite. Instead of all changes being considered bad, all changes are now considered OK. To add some order to this chaos, the serialization spec groups possible changes to a class into two categories: compatible and incompatible changes. *Compatible changes* include adding new serializable fields or adding or removing classes from the inheritance hierarchy. *Incompatible changes* include deleting fields, juggling the order of classes in the inheritance hierarchy, or changing the type of a primitive field. The two types of changes are summarized in Table 4–1.

Table 4–1 Compatible and Incompatible Changes

Type of Change	Examples
Compatible change	Adding fields, adding/removing classes, adding/removing <code>writeObject/readObject</code> , adding <code>Serializable</code> , changing access modifier, removing <code>static/transient</code> from a field
Incompatible change	Deleting fields, moving classes in a hierarchy, adding <code>static/transient</code> to a field, changing type of a primitive, switching between <code>Serializable</code> or <code>Externalizable</code> , removing <code>Serializable/Externalizable</code> , changing whether <code>readObject/writeObject</code> handles default field data, adding <code>writeReplace</code> or <code>readResolve</code> that produces objects incompatible with older versions

When you make a compatible change to a class, the runtime does the best job it can with the data it finds in the stream. For example, if you add a class to the inheritance hierarchy after serializing an instance, there will be no data in the stream for that class's data members. So, when you deserialize the object, the new class's members will be initialized to the default value appropriate to their type: `false` for Booleans, zero for numeric types, and `null` for references.

Similarly, if you add a serializable field, old versions of the object will not have a value for that field.

The new version of `Person` demonstrates this because it has added an `age` field. When you read an old `Person` stream into a new `Person` class, the `age` value will be zero as shown here:

```
read object Person Julius Drabbih aged 0
```

In the next section, you will see a more advanced use of `readObject` that can help deal with this situation.

Unfortunately, the serialization spec is unclear about what should happen when you make an incompatible change to a class. Based on the term “incompatible,” you might expect that an incompatible change would cause deserialization to fail with an exception. In the Java 2 SDK, this is true for some, *but not all*, types of incompatible changes. For example, if you delete a field from the `Person` class, then the stream will have a value for that field, and nowhere to put it. Rather than throw an `InvalidClassException`, `ObjectInputStream` silently drops the field.

Most other incompatible changes will throw exceptions. The exact behavior of the Java 2 SDK version 1.3 for each type of incompatible change is summarized in Table 4–2. Since the spec is not clear in mandating these behaviors, other implementations might be different.

Table 4–2 How Java 2 SDK 1.3 Handles Incompatible Changes

Incompatible Change	Runtime Response
Deleting a field	Silently ignored
Moving classes in inheritance hierarchy	Exception
Adding <code>static</code> / <code>transient</code>	Silently ignored
Changing primitive type	Exception
Changing use of default field data	Exception
Switching <code>Serializable</code> and <code>Externalizable</code>	Exception
Removing <code>Serializable</code> or <code>Externalizable</code>	Exception
Returning incompatible class	Depends on incompatibility

4.5 Explicitly Managing Serializable Fields

With default serialization, the mapping between class fields and stream fields is automatic and transparent. At serialization time, a field's name and type in the class become the field's name and type in the stream. The fields written by default serialization are called the *default field data*. At deserialization, a field's name and type in the stream are used to find the correct field to assign in the new instance.

The serialization API exposes hooks so that you can take control of any of these steps. Two nested classes do most of the work. `ObjectInputStream.GetField` allows you to explicitly manage pulling fields out of the stream, and `ObjectOutputStream.PutField` allows you to explicitly manage inserting fields into the stream.

`ObjectOutputStream.GetField` presents all the stream fields as name/value pairs. In order to access the stream in this fashion, you have to implement `readObject`, but instead of calling `defaultReadObject`, you call `readFields`. Then, it is up to you to extract each field by name and assign it to the appropriate field in the object. Consider the new version of `Person` in Listing 4–7. This version of `Person` stores both names in the single field `fullName`.

Listing 4–7 Person Using `readFields` to Handle Different Versions

```
import java.io.*;

public class Person implements java.io.Serializable {
    private String fullName;
    static final long serialVersionUID=388032154234815834L;
    public Person(String fullName) {
        this.fullName = fullName;
    }
    public String toString() {
        return "Person " + fullName;
    }
    private void readObject(ObjectInputStream ois)
        throws IOException, ClassNotFoundException
    {
        ObjectInputStream.GetField gf = ois.readFields();
        fullName = (String) gf.get("fullName", null);
        if (fullName == null) {
            String lastName = (String) gf.get("lastName", null);
```

```

        String firstName = (String) gf.get("firstName", null);
        if ((lastName == null) || (firstName == null))
            throw new InvalidClassException("invalid Person");
        fullName = firstName + " " + lastName;
    }
}

```

The `readObject` method has been implemented to correctly read either new- or old-format streams. Instead of calling `defaultReadObject`, the `readObject` implementation begins with a call to `readFields`, which exposes the fields as a collection of name/value pairs. You can then extract the field values using a family of

```
type get(String name, type default)
```

methods, one for each primitive type and one for `Object`. The first call to `get` optimistically assumes that the stream version matches the class, and that `fullName` is available. If it is not, then `readObject` continues and tries to interpret the stream as the original `Person` version, extracting `firstName` and `lastName` fields. You can make your `readObject` implementations as complex as necessary, possibly handling multiple old versions of a class.

4.5.1 `ObjectInputStream.GetField` Caveats

There are two caveats to remember when you are using `ObjectInputStream.GetField` to manage fields. First, it is an all-or-nothing deal. If your class has 70 fields, there is no way to tell `ObjectInputStream` to “use `defaultReadObject` for these 65 fields, and let me handle the rest myself.” Once you decide to call `readFields`, you have to assign all the fields yourself.⁶

The second caveat is that the `GetField.get` methods do not like field names that do not appear in any version of the class being deserialized. If you attempt to `get` a field that cannot be found in the stream and that field also cannot be found in the local version of the class, the runtime will throw an

6. The spec does not appear to mandate this behavior. In fact, the source for `readFields` has this comment: “TBD: Interlock w/ `defaultReadObject`.” Perhaps a future version will allow you to call `defaultReadObject` and `readFields` for the same `Object`.

“`IllegalArgumentException: no such field.`”⁷ This situation is likely if you are dealing with three or more versions of a class over time. To handle this situation, wrap calls to `get` inside a `try` block.

4.5.2 Writer Makes Right

When you use `readObject` and `GetField` to control deserialization, the writer of an object does not worry about the stream format, instead, it leaves the reader to make things right. This can be more efficient than having the writer try to guess the format; if the writer guesses incorrectly, the result is that both writer *and* reader do extra work. However, the reader-makes-right approach has a disadvantage as well. While new versions of a class can read either old or new versions from the stream, an old version of a class cannot handle a newer version of the stream format.

If your design does not permit you to update all versions of a class everywhere, then you may need to code newer versions of a class to respect the original format. Serialization provides a hook for this with `GetField`’s mirror image, the `PutField` class. You customize serialization output by implementing `readObject`’s counterpart, `writeObject`:

```
private void writeObject(ObjectOutputStream oos)
    throws IOException;
```

The `PutField` class has a set of `put` methods that write field values to the stream. Listing 4–8 shows a version of `Person` that uses `writeObject` to control which fields are serialized. The first line of `writeObject` retrieves the `PutField` instance that is used to write objects to the stream. Then, the `put` method is used to assign name/value pairs, and the `writeFields` method adds all the fields to the stream. By implementing both `readObject` and `writeObject`, this new version of `Person` continues to both read and write the format established by the original version of `Person`.

7. This is an unnecessary complication. There are other ways to find out if the field exists in the stream or class; `GetField.get` would be easier to use if it always handled not finding the field by returning the default value passed as its second parameter.

Listing 4–8 Using writeObject for Backward Compatibility

```

import java.io.*;
public class Person implements java.io.Serializable {
    private String fullName;
    static final long serialVersionUID=388032154234815834L;
    public Person(String lastName, String firstName) {
        this.fullName = firstName + " " + lastName;
    }
    public String toString() {
        return "Person " + fullName;
    }
    private static final ObjectOutputStream[]
        serialPersistentFields
        = {new ObjectOutputStream("firstName", String.class),
          new ObjectOutputStream("lastName", String.class)};
    private void writeObject(ObjectOutputStream oos)
        throws IOException
    {
        ObjectOutputStream.PutField pf = oos.putFields();
        int delim = fullName.indexOf(" ");
        String firstName = fullName.substring(0, delim);
        String lastName = fullName.substring(delim+1);
        pf.put("firstName", firstName);
        pf.put("lastName", lastName);
        oos.writeFields ();
    }
    private void readObject(ObjectInputStream ois)
        throws IOException, ClassNotFoundException
    {
        ObjectInputStream.GetField gf = ois.readFields();
        String lastName = (String) gf.get("lastName", null);
        String firstName = (String) gf.get("firstName", null);
        if ((lastName == null) || (firstName == null))
            throw new InvalidClassException("invalid Person");
        fullName = firstName + " " + lastName;
    }
}

```

4.5.3 Overriding Class Metadata

Using the `writeObject` method introduces one additional complexity not present when using `readObject`. You cannot just write any field name and type

that you choose; you can only write fields whose names and types are part of the class metadata. This information cannot be modified at runtime because class metadata is only written to a stream once; later references to the same class simply reference the original metadata. Remember that by default serialization will use reflection to discover the names and types of a class's nonstatic, nontransient fields.

If you want to bypass reflection and specify the class serialization metadata directly, you must specify the class field

```
private static final ObjectOutputStreamField[] serialPersistentFields
```

The runtime will discover the `serialPersistentFields` array by reflection, and it will use them to write the class metadata to the stream.

`ObjectStreamField` is a simple collection class that contains a `String` holding a field name and a `Class` holding a field type. In the `Person` example in Listing 4-8, `writeObject` needs to write `firstName` and `lastName` to the stream, so `serialPersistentFields` is set to contain appropriate `ObjectStreamField` instances. If you change the class metadata by setting `serialPersistentFields`, you must also implement `writeObject` to write instance fields that match your custom metadata, and you must implement `readObject` to read those fields. If you don't, `ObjectOutputStream` will try to reflect against your class, find fields that do not match the metadata, and fail with an `InvalidClassException`.

4.5.4 Performance Problems

The current SDK implementations of `GetField` and `PutField` perform poorly. The class metadata, whether generated by reflection or specified explicitly via `serialPersistentFields`, is stored as an instance of `ObjectStreamClass`. Instead of using an efficient hash table, `ObjectStreamClass` stores the field information in a sorted array and uses a binary search to find fields at runtime. If you make heavy use of `GetField` and `PutField`, these binary searches become the primary bottleneck when serializing an object.

The default serialization mechanism does not pay this penalty because it makes a linear traversal of the sorted array. Unfortunately, this linear traversal is

not accessible to user code. This implementation defect may be repaired in a future version of the SDK.

4.5.5 Custom Class Descriptors

In addition to instance-specific metadata hooks, serialization also provides a mechanism for customizing the reading and writing of class metadata. This mechanism is rarely used because it requires matching modifications to both input and output streams, and it makes your streams usable only by stream subclasses that use the modified version. Consult the API documentation for details under `ObjectOutputStream`'s `writeClassDescriptor` and `ObjectInputStream`'s `readClassDescriptor`.

4.6 Abandoning Metadata

In all the scenarios discussed so far, class metadata is part of the serialization format. The first time an instance of a particular class is written, the class metadata is also written, including the class name, SUID, field names, and field types. When default serialization is used, the field names are discovered by reflection, and the SUID is calculated by taking an SHA-1 hash of the class metadata. When you override these behaviors by specifying `serialVersionUID` or `serialPersistentFields`, you are not eliminating metadata. Instead, you are just taking explicit control of what the metadata looks like.

The serialization mechanism also provides several hooks that allow you to skip sending metadata at all. This section will show you various techniques for reducing metadata, and then it will explain why you should avoid these techniques in most cases. There are three techniques for bypassing metadata: adding data after `defaultWriteObject`, making your object `Externalizable`, and replacing `defaultWriteObject` entirely.

4.6.1 Writing Custom Data after `defaultWriteObject`

The first technique, adding data after `defaultWriteObject`, allows you to make ad hoc extensions to an instance's serialization packet. `ObjectOutputStream` and `ObjectInputStream` implement `DataOutput` and `DataInput`, respectively. These interfaces provide helper methods for reading and writing

primitive types. Return to the original `Person` example from Listing 4–1. One of `Person`'s problems was that the `Humanoid` base class data was not written to the stream. You could solve this problem by adding extra lines to `readObject` and `writeObject` like this:

```
//add to Person.java. This is _not_ a great design!
private void writeObject(ObjectOutputStream oos)
    throws IOException
{
    oos.defaultWriteObject();
    oos.writeInt(noOfHeads);
}
private void readObject(ObjectInputStream ois)
    throws IOException, ClassNotFoundException
{
    ois.defaultReadObject();
    noOfHeads = ois.readInt();
    System.out.println("had " + noOfHeads + " heads");
}
```

After calling `defaultWriteObject` to write the standard fields and metadata, the call to `writeInt` simply tacks on an extra piece of data. Similarly, the call to `readInt` extracts that extra data item. Unlike `Person`'s `lastName` and `firstName` fields, this extra data travels naked—without any metadata describing what it is. This means that somebody with a different version of the class (or no version of the class at all) will be unable to determine what the `int` value in the stream means. In fact, a reader without this exact version of the class probably cannot even tell if the value is an `int`, as opposed to two shorts or four bytes. A more flexible solution to this problem is to use `serialPersistentFields` and `PutField` as discussed in the previous section. That way, you can explicitly guarantee that the correct metadata is present, which will vastly increase the chances that a reader of your stream will be able to interpret the data.

4.6.2 Externalizable

A more heavy-handed approach to bypassing metadata is to declare that your class implements `java.io.Externalizable`, which extends `Serializable`. When you implement `Externalizable`, you make your class `Serializable`,

but you also take full responsibility for transmitting data and metadata for that class and for transmitting data and metadata for any base class data and metadata. You must explicitly manipulate the stream using `Externalizable`'s two methods, shown here:

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

For our old friend `Person`, these methods might be implemented as shown in Listing 4–9. There are several important points to notice in this example.

1. The `Person` class deals explicitly with all fields for itself and its base classes.
2. Because the `readExternal` method must be declared public, malicious or ignorant code might invoke the `readExternal` method at any time, blasting some arbitrary state into your object.
3. An `Externalizable` class must have a public constructor.⁸
4. Finally, *no metadata is written*. Only the actual value of the fields is written to the stream.

Listing 4–9 Externalizable Version of `Person`

```
public class Person extends Humanoid
    implements java.io.Externalizable
{
    //other fields, methods, as before
    public Person() { /*required! */}
    public void readExternal(ObjectInput oi)
        throws IOException
    {
        lastName = oi.readUTF();
        firstName = oi.readUTF();
        noOfHeads = oi.readInt();
    }
}
```

8. The requirements that methods be public and that `Externalizable` objects have a public constructor are completely out-of-step with the rest of serialization. Most other serialization behaviors and customizations are implemented using reflection and naming conventions. Because serialization can use reflection to bypass language protections, it can hide its details in private methods. The designer of externalization must have momentarily forgotten about these advantages.

```
public void writeExternal(ObjectOutput oo)
    throws IOException, ClassNotFoundException
{
    oo.writeUTF(lastName);
    oo.writeUTF(firstName);
    oo.writeInt(noOfHeads);
}
}
```

Using `Externalizable` introduces three dangers:

1. Other versions of the class may not be able to figure out what is in the stream.
2. Generic tools that analyze serialization streams will have to skip over `Externalizable` data, treating it as an opaque array of bytes.
3. It is easy to introduce bugs when writing `Externalizable` code. If you write the fields out in one order, and then read the fields back in a different order, the best you can hope for is that the stream will break. If the wrong ordering is type-compatible with the correct ordering, you will silently assign data to the wrong fields.

Given all of the dangers of `Externalizable` objects, what purpose do they serve? In some situations, `Externalizable` objects offer better serialization performance. Skipping metadata has three potential performance benefits:

1. The stream is smaller because the metadata is not present.
2. There is no need to reflect over metadata for the class.
3. There is no need to use reflection on a per-instance basis to extract and assign values from fields.

How much actual performance benefit you get from externalizing a class will depend heavily on how that class is used, when it is serialized, and what type of stream `ObjectOutputStream` is wrapping. In many cases, the performance benefit will be negligible. Externalization should never be your first option; only consider it when your application is functioning correctly and you have profiling data to prove that externalization provides an essential speedup. The most likely place for externalization is for simple classes that never change, so for them, metadata is not important.

4.6.3 Using `writeObject` to Write Raw Data Only: Bad Idea

The third option for bypassing metadata is to implement `writeObject` to write data directly, without first calling `defaultWriteObject` or `putFields` to invoke the normal metadata mechanism. You should *never* use this option. The ability to write data in this way was an unintended loophole in the serialization specification. Unfortunately, this technique is used in a few places in the core API, so the spec is not likely to preclude this tactic anytime soon.

Listing 4–10 shows this technique. This version of `Person` uses the `readObject` and `writeObject` hooks for serialization-with-metadata, but the code looks like it belongs in an `Externalizable` object instead. You should never write code like this. If you truly want to bypass all metadata, you should (1) reconsider one last time; then (2) implement `Externalizable`.

Listing 4–10 Bad Style in `writeObject`

```
public class Person extends Humanoid
    implements java.io.Serializable
{
    //other fields, methods, as before
    private void readObject(ObjectInputStream ois)
        throws IOException
    {
        lastName = ois.readUTF();
        firstName = ois.readUTF();
        noOfHeads = ois.readInt();
    }
    private void writeObject(ObjectOutputStream oos)
        throws IOException, ClassNotFoundException
    {
        oos.writeUTF(lastName);
        oos.writeUTF(firstName);
        oos.writeInt(noOfHeads);
    }
}
```

To understand the problem with this use of `writeObject`, you need to look at the details of serialization's binary format. The binary format relies on the following assumptions about objects that implement `writeObject`.

1. The default field data will occur exactly once, and it will occur first. There is no need for a special marker in the binary format because this data will always be present. The implication for developers is that you should always begin your `writeObject` implementation with a call to `defaultWriteObject`, or with calls to the `PutField` nested class that do essentially the same thing.
2. If custom data is present, it will follow after the normal serialization data. Because custom serialization data is optional, the byte flag `TC_BLOCKDATA` indicates the beginning of custom data.

If you do violate these assumptions in your own code, either by writing to the `ObjectOutputStream` before writing the normal data, or by never writing the normal data at all, your code will still execute correctly. However, if anyone ever tries to read your stream without having the original class, they stand a 1-in-256 chance of being stymied.

If the first byte of your instance's serialized data happens to be the constant `TC_BLOCKDATA`, readers cannot depend on metadata to tell them what they are looking at, as demonstrated by Figure 4-1. Maybe that first byte is the beginning of some custom data, or maybe it is the beginning of normal data that just happens to start with that value. The benefit of metadata is lost because now readers must have a class file that knows how the original stream was produced. In your own code, you should obey the intent of the specification, and always write normal serialization data first.

I would summarize the options for avoiding metadata as ranging from bad, to worse, to worst.

- **Bad:** If you implement `writeObject` to first write the normal serialization data, and then use the stream's `DataOutput` capabilities to tack on extra data, your serialization stream will be an odd hybrid. The normal data will include metadata, and the extra data will not.
- **Worse:** If you implement `Externalizable`, you lose all metadata benefits, you have to handle base classes yourself, and you must write (and debug!) a lot of per-class code. However, both of these options have their uses. Appending extra data to `writeObject` is slightly easier than using `serialPersistentFields` to provide full metadata, and it may be suitable if you value development speed over flexibility. `Externalizable`

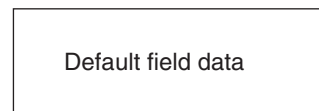


objects may be enough faster and smaller that you are willing to deal with inflexible, error-prone code.

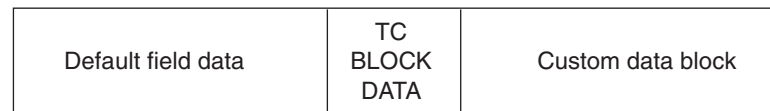
- Worst: Never violate the spirit of the specification by skipping the normal serialization data in your `writeObject` implementation. If you do, it will be impossible for a generic tool to reliably extract the data from the stream.

Do not throw metadata away. Instead of using the techniques explained here, use the metadata-friendly techniques described in §4.5.

Case 1. Simple serialization stream



Case 2. `writeObject` calls `defaultWriteObject` and then writes custom data.



Case 3. `writeObject` writes custom data. This cannot be distinguished from Case 1 when default field data begins with TC_BLOCKDATA flag.

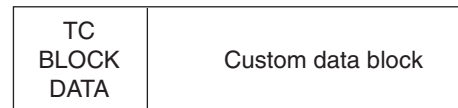


Figure 4–1 Streams generated by `WriteObject`

4.7 Object Graphs

Serialization is recursive. When you serialize an object, all of its serializable fields are also serialized. Primitives are serialized by the `DataOutput` methods of the stream, and objects are serialized by `writeObject`. This fact is implicit in the `Person` example of this chapter, since `lastName` and `firstName` are themselves object types, not primitives. Recursion to referenced objects is highly desirable because it simplifies serializing complex graphs of objects all in one step. However, the fact that `writeObject` actually serializes an entire *graph* of objects introduces a few wrinkles that you need to be aware of.



The first issue is the danger of serializing far more data than you wanted. Consider a data object that lives inside a hierarchical container:

```
public class PhotonTorpedo implements Serializable {  
    private int range;  
    private int power;  
    private Starship ship;  
    //etc.  
}
```

In this design, all `PhotonTorpedos` are contained by a `Starship`. Perhaps each `Starship` in turn belongs to a `Fleet`. This is a perfectly reasonable model, but when you serialize a `PhotonTorpedo`, you wind up attempting to serialize all the other weapons on the `Starship`, and all the other `Starships` in the `Fleet`. If a single one of these connected instances is not serializable, serialization will fail with a `NotSerializableException`. More amusingly, if the entire graph is serializable, you will wind up scratching your head wondering why a `PhotonTorpedo` takes up 48.7MB on disk, or why it takes four hours to send one over the network!

4.7.1 Pruning Graphs with Transient

If the reader does not care about containers like the `Starship`, you can use the `transient` keyword to block serialization of container references like `ship`. Once `ship` is marked `transient`, you probably will also need to add code to `readObject` or `readExternal` to correctly reinitialize the value of `ship` at deserialization time.

4.7.2 Preserving Identity

Another situation that arises when you are serializing an entire object graph is the possibility that the same instance might be serialized more than once.⁹ Consider the following additions to the `Person` class:

```
public class Person {  
    private Person spouse;  
    private Person boss;  
}
```

9. Of course, this can also happen if you simply call `writeObject` twice on the same object.

In this situation, it is very likely that `spouse` and `boss` are the same Java identity. However, serialization only writes the state once, and it uses that state to reinitialize both references when the object is deserialized. To track identity, each object is assigned a numeric token the first time it is written. When the same object needs to be written again, only the token needs to be written to the stream.

There are three reasons why the architecture works this way:

1. Sending the state only once is more efficient.
2. Sending the state only once provides more intuitive semantics. If I deserialize a `Person` and call `p.getSpouse().appease()`, I expect both the `spouse` and the `boss` to be appeased. This will only work if a single instance is deserialized and assigned to both references.
3. Failing to track object identity will lead to an infinite loop if there are circular references. Consider the `Person` example again. When you serialize a `Person`, you recursively serialize the `spouse` instance. But `spouse` is also a `Person`, so you end up serializing `Person.spouse.spouse`, which (hopefully!) is the original `Person` again. Serialization has to recognize that this is a `Person` it has seen before, or else it will run in circles until it blows the stack.

4.7.3 Encouraging the Garbage Collector with reset

Because Java tracks object identities and does not write an instance to the stream more than once, all these problems are solved without any effort on the programmer's part. However, the tracking mechanism must keep a reference to every object ever written to a stream.¹⁰ As a result, no object that is written to an object stream can be garbage collected! In a long running application that uses serialization, this can cause poor performance and may eventually lead to an `OutOfMemoryError`. In order to avoid this problem, `ObjectOutputStream` provides a `reset` method.

```
public void reset() throws IOException
```

10. This is not strictly true. The design could have used a `java.lang.ref.WeakReference` to track streamed objects while still allowing them to be garbage collectable. This option was rejected as unnecessarily expensive in the general case.

When you call `reset`, the stream nulls its internal table of already-written objects. If an object is written to the stream before and after the reset point, the object's state will be written twice, and the receiver will see two different object identities. If this identity-destroying behavior is desirable or acceptable in your application, then you can call `reset` regularly to allow streamed objects to be garbage collected. If this behavior is *not* desirable, you will have to carefully coordinate calls to `reset` to preserve the semantics required by the receiver.

The relationship between `reset` and garbage collection is needlessly complex in older versions of Java. For example, in SDK version 1.3, the following rules apply:

1. If an `ObjectOutputStream` is no longer reachable, then its references to streamed objects will no longer prevent them from being garbage collected. This is just vanilla GC behavior.
2. Calling `close` on an object stream does *not* `reset` the stream! If you want to free resources in a timely fashion, you must explicitly call `reset`, in addition to explicitly calling `close`.
3. Calling `reset` does *not* immediately clear the references to objects that have been streamed. This is an unfortunate accident of the `ObjectOutputStream`'s nested `HandleTable` implementation, which marks the table as empty without explicitly setting references to null. As a result, objects that pass through an object stream will only be collectable if a call to `reset` is followed by streaming some additional objects through to overwrite the old references.

These rules are demonstrated by the `YouMustReset` class in the accompanying source code. Note that rules 2 and 3 are inferred from the source code, not mandated by the serialization spec. The memory management problems of `close` and `reset` are fixed in the 1.4 version of Java. In 1.4, calling `close` does clear the object table and null out all old references. So, rules 2 and 3 apply only to versions of Java prior to 1.4.

4.8 Object Replacement

Both object streams and objects themselves have the ability to nominate replacement objects at serialization time. This object replacement feature has

many uses. If your object graph contains an object that is not serializable, you can replace it with an object that is serializable. If you are doing distributed programming, you can replace an object's state with an object's stub, causing the receiver to get a reference to the remote object, instead of a local by-value copy of the object. RMI uses object replacement for this purpose. Replacement may also be useful if you want to add additional semantics to serialization that go beyond the capabilities of the `readObject` and `writeObject` methods.

4.8.1 Stream-Controlled Replacement

You can implement stream-controlled object replacement by subclassing `ObjectOutputStream` and/or `ObjectInputStream`. The relevant methods are shown in Listing 4-11. You must call the `enableReplaceObject` and `enableResolveObject` methods to turn replacement on or off; the default is off. These methods also act as a chokepoint for a security check. Because a stream might use object replacement to corrupt an object graph, the `enable` methods require that the caller have the `SerializablePermission` "enableSubstitution".¹¹

Listing 4-11 Stream-Level Object Replacement APIs

```
package java.io;

public class ObjectOutputStream {
    //ObjectOutputStream replacement methods
    protected boolean enableReplaceObject(boolean enable);
    protected Object replaceObject(Object obj);
    //rest of class omitted for clarity
}

public classs ObjectInputStream {
    //ObjectInputStream replacement methods
    protected boolean enableResolveObject(boolean enable);
    protected Object resolveObject(Object obj);
    //rest of class omitted for clarity
}
```

11. See [Gon99] for a detailed explanation of Java 2 permissions.

To actually perform replacement and resolution, you override `replaceObject` and `resolveObject`, respectively. If replacement is enabled, these methods will be called once for every object serialized to the stream, allowing you to substitute a different object. You may substitute any object you want. However, if you replace an object with an object that is not type-compatible, you will need to resolve it back to a type-compatible reference or the stream will throw an exception.

One use of stream-controlled replacement is to serialize an object that would not otherwise be serializable. Consider the `SimplePerson` class in Listing 4-12. If you tried to serialize a `SimplePerson`, a normal object stream would throw a `NotSerializableException`. If you control the source for `SimplePerson`, you can fix this by declaring that `SimplePerson` implements `Serializable`. If you do not have control of the source code, the next best thing to do is to replace the `SimplePerson` instance with some other class instance that is serializable.

Listing 4-12 Serializing a Nonserializable Instance

```
public interface PersonItf {
    public String getFullName();
}

public class SimplePerson implements PersonItf {
    String fullName;
    public SimplePerson(String fullName) {
        this.fullName = fullName;
    }
    public String getFullName() {
        return fullName;
    }
}

import java.io.*;
public class WriteSimplePerson {
    private static class Replacer extends ObjectOutputStream {
        public Replacer(OutputStream os) throws IOException {
            super(os);
            enableReplaceObject(true);
        }
    }
}
```

```

        protected Object replaceObject(Object obj) {
            if (obj instanceof PersonItf) {
                PersonItf p = (PersonItf) obj;
                return new StreamPerson(p.getFullName());
            }
            return obj;
        }
    }

    public static void main(String[] args)
        throws IOException
    {
        FileOutputStream fos = new FileOutputStream
            (args[0]);
        ObjectOutputStream oos = new Replacer(fos);
        oos.writeObject(new SimplePerson("Fred Wesley"));
    }
}

public class StreamPerson implements java.io.Serializable,
PersonItf {
    String fullName;
    public StreamPerson(String fullName) {
        this.fullName = fullName;
    }
    public String getFullName() {
        return fullName;
    }
    public String toString() {
        return "StreamPerson " + fullName;
    }
}

```

The `WriteSimplePerson` class in Listing 4–12 demonstrates using replacement to cope with an instance that is not serializable. The `Replacer` class checks to see if an object implements `PersonItf`. If it does, then `Replacer` replaces it with the `StreamPerson` class, which is known to be `Serializable`. You could implement a corresponding `Resolver` subclass of `ObjectInputStream` to convert the object back to a `SimplePerson` at deserialization time. However, in this example there is probably no need to do so. If the receiver is expecting only a `PersonItf`, then `StreamPerson` is just as good as `SimplePerson`. The `ReadInstance` class from Listing 4–2 certainly doesn't care since it never casts the result to anything more specific than

Object. If you use `WriteSimplePerson` to create a file `SimplePerson.ser`, `ReadInstance` will happily deserialize a `StreamPerson` as shown here:

```
java -cp classes ReadInstance SimplePerson.ser
read object StreamPerson Fred Wesley
```

As long as type relationships are maintained, replacement is totally transparent for the reader. As far as `ReadInstance` knows, there never was any class other than `StreamPerson` involved. If the receiver did want to cast an object to the original implementation class, or if the object was referenced via its specific subtype in an object graph, then a resolution step would also be necessary to convert the stream type back into the type expected by the reader.

The usage of replacement without resolution opens an interesting possibility. Imagine that there are hundreds of implementations of the `PersonItf` interface, and that some of them are very large and expensive to serialize. One mechanism to trim down the cost of serializing the various classes would be to go through the source code marking fields `transient`. However, `transient` is a property of the field itself, not of any particular instance. If different clients care about different fields, you cannot selectively set the `transient` bit at runtime. Because you cannot use `transient` selectively, you end up having to serialize all the fields, even though any particular client might only care about some subset.

Object replacement allows you to customize serialization on a per-stream basis, unlike the `transient` keyword, which operates on a per-class basis. Look again at the `Replacer` class. It replaces *all* `PersonItf` implementations with `StreamPersonS`. If you know in advance that the receiver of an object stream cares only about the `PersonItf`ness of objects, then the `Replacer` class saves you from worrying about whether a particular `PersonItf` is serializable, and it gives serialization a predictable cost.

4.8.2 Class-Controlled Replacement

Replacement at the class level is syntactically very similar to stream-level replacement. A class that desires replacement implements the methods in Listing 4–13. Like many serialization hooks, these methods are not part of any interface and are discovered by reflection.¹² The type compatibility rules for class-

12. This is also why the access modifier does not matter.

level object replacement are the same as for stream-level replacement. If you replace an object, you need to make sure that the replacement is, or later resolves to, a type expected by the receiver.

Listing 4-13 Class-Level Object Replacement Hooks

```
ANY-ACCESS Object writeReplace()
                    throws ObjectOutputStreamException;
ANY-ACCESS Object readResolve()
                    throws ObjectOutputStreamException;
```

Class-level replacement has several important differences from stream-based replacement. Because it operates at the level of the class being serialized, you can use class-level replacement only if you have access to the source code for the class being replaced. On the positive side, no security check is necessary; since all the relevant code is in the same class, it is assumed to be within a single trust boundary.

Classes use the class-level object replacement mechanism to separate their serialized form from their in-memory representation. Explicitly coding a separate serialized form may be useful if a class has complex serialization code, or serialization code that is shared with other classes.

Another place where class-level replacement is useful is in designs that rely on object identity, such as certain implementations of the singleton design pattern. A *singleton* is a unique object in a system, often represented by a single object identity. Deserialization creates new object identities, so if a singleton is deserialized more than once, its unique identity is lost.

Object resolution can be used to patch this situation, as shown in Listing 4-14. The author of the `Planet` class wants to guarantee that there are only two singleton `Planets` in the entire VM: `earth` and `mars`. To prevent accidental planet creation, `Planet`'s constructor is marked `private`. For a nonserializable class, this would be good enough, but remember that serialization acts like a public constructor. When a `Planet` is deserialized, the serialization architecture creates a new instance and populates its fields by reflection.

If `earth` were deserialized twice, there would be two different object identities holding the same `earth` state. This could cause program bugs if code relies on reference equality to compare `Planets`, and at the very least it wastes

memory with unnecessary copies of semantically identical `Planet`s. The `resolve-Object` method sidesteps this problem by looking up the singleton `Planet` and returning it instead. Because of this, the `Planet` reference created by deserialization is never visible to client code and is available for garbage collection.

Listing 4-14 Using Class-Controlled Resolution to Preserve Identity

```
import java.io.*;
public class Planet implements Serializable {
    String name;
    private static final earth = new Planet("Earth");
    private static final mars = new Planet("Mars");
    private Planet(String name) {
        this.name = name;
    }
    public static Planet getEarth() {
        return earth;
    }
    public static Planet getMars() {
        return mars;
    }
    private Object readResolve() throws IOException
    {
        if (name.equals("Earth"))
            return earth;
        if (name.equals("Mars"))
            return mars;
        throw new InvalidObjectException("No planet " + name);
    }
}
```

4.8.3 Ordering Rules for Replacement

Object replacement has a number of ordering rules that you must take into account if you are doing any nontrivial replacements.¹³

1. *Class-level* replacement is invoked prior to *stream-level* replacement. This is the only sensible ordering; letting the stream go first could violate a class's internal assumptions about how it will be serialized. Similarly, class-level resolution occurs prior to stream-level resolution.

13. These ordering rules live in a gray area between the specification and a specific implementation's detail. I would expect most vendors to follow the SDK's lead, but if you are relying on these behaviors it wouldn't hurt to test them on each Java implementation you plan to support.

2. Class-level *replacement* is recursive; that is, a replacement can nominate another replacement and so on ad infinitum. Class-level *resolution* is not recursive.¹⁴
3. Stream-level replacement/resolution is not recursive. Streams get exactly one chance to replace/resolve an object.
4. During serialization, objects are replaced as they are encountered. During deserialization, objects are replaced only after they are fully constructed.

These rules are shown graphically in Figure 4–2 and Figure 4–3.

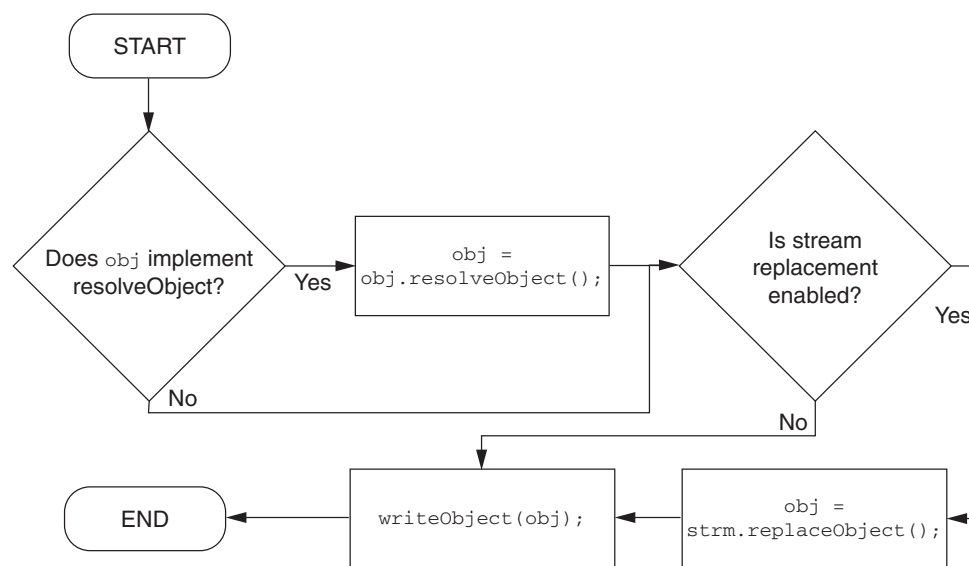


Figure 4–2 Choosing a replacement for obj during serialization

Rules 2 and 4 have asymmetries that are counterintuitive. While class replacement executes recursively, class resolution does not. So, if class *A* nominates replacement *B*, and *B* nominates replacement *C*, the only way you can force *C* to resolve to *A* is in a single step.

14. This behavior has an odd history. In SDK version 1.2, neither replacement nor resolution were recursive. This was reported as bug ID 4217737, and subsequently “fixed” in SDK 1.3 Beta. While the bug report asked that both behaviors be made recursive, only replacement was changed.

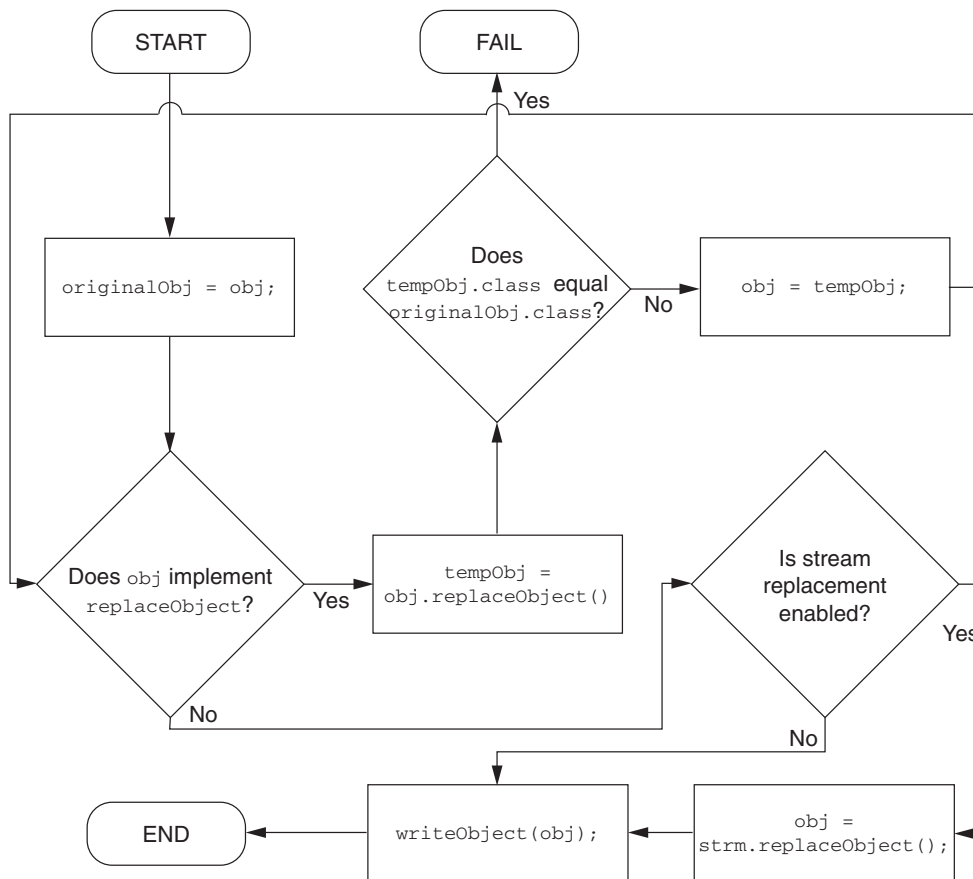


Figure 4-3 Resolving obj

Rule 4 implies that an object graph with both circular references and object replacement may be impossible to deserialize correctly. Consider the `Worker` and `Boss` classes in Listing 4-15.

Listing 4-15 Circular References May Cause Problems During Resolution.

```

import java.io.*;
public class Worker implements PersonItf, Serializable {
    private String name;
    private PersonItf boss;
    public Worker(String name, PersonItf boss) {
        this.name = name;
        this.boss = boss;
    }
}

```

```

    }
    public String toString() {
        return "Worker " + name + " (boss " + boss.getFullName()+")";
    }
    private Object writeReplace() {
        System.out.println("Replacing worker");
        return new WireWorker(name, boss);
    }
    public String getFullName() {
        return name;
    }
    public PersonItf getBoss() {
        return boss;
    }
}

import java.io.*;
import java.util.*;
public class Boss implements PersonItf, Serializable {
    private String name;
    private ArrayList workers = new ArrayList();
    public Boss(String name) {
        this.name = name;
    }
    public String toString() {
        StringBuffer sb = new StringBuffer("Boss " +
            name + ", workers:");
        for (int n=0; n<workers.size(); n++) {
            sb.append("\n\t").append(workers.get(n));
        }
        return sb.toString();
    }
    public void addWorker(PersonItf worker) {
        workers.add(worker);
    }
    public String getFullName() {
        return name;
    }
}

```

Serializing an instance of one of these classes is likely to involve a circular reference since `Boss` keeps an `ArrayList` of `Workers`, and each `Worker` keeps a reference to its `Boss`. Serialization will also involve replacement, as `Worker` nominates a replacement class `WireWorker`. To keep the example simple,

`WireWorker` does not actually add any functionality; it simply holds references to the data members of the original `Worker` instance. Consider what happens when a `Worker/Boss` tandem is serialized as shown in Listing 4–16.

Listing 4–16 Serializing a Worker

```
import java.io.*;
public class WriteWorker {
    public static void main(String[] args) throws IOException
    {
        FileOutputStream fos = new FileOutputStream(args[0]);
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        Boss b = new Boss("Queen");
        Worker w = new Worker("Drone", b);
        b.addWorker(w);
        System.out.println(w);
        System.out.println(b);
        oos.writeObject(w);
        System.out.println("wrote worker to " + args[0]);
    }
}

import java.io.*;
public class ReadWorker {
    public static void main(String [] args)
        throws Exception
    {
        String fileName = args[0];
        FileInputStream fis = new FileInputStream(fileName);
        ObjectInputStream ois = new ObjectInputStream(fis);
        Worker w = (Worker) ois.readObject();
        System.out.println("read worker from " + args[0]);
        System.out.println(w);
        System.out.println(w.getBoss());
    }
}
```

The stream replaces `w` with a `WireWorker`, call it `w'`. While the stream continues to chase references, it writes `b`, which has *another* reference to `w` in the `workers` array. Because replacement occurs as references are encountered, this reference to `w` will also be replaced by `w'`. The stream's view of the object graph is depicted in the middle portion of Figure 4–4.

At deserialization time, the object graph is read back into memory. When the top-level object *w'* is reconstructed, it gets a chance to `readResolve`. At this point, the top-level reference is turned back into a `Worker`—call it *w* for symmetry. However, the runtime does not keep a table of previously resolved objects. When the runtime encounters *w'* again, it sees a handle for a `WireWorker` instance and returns that instance, as shown in the bottom stripe of Figure 4–4.

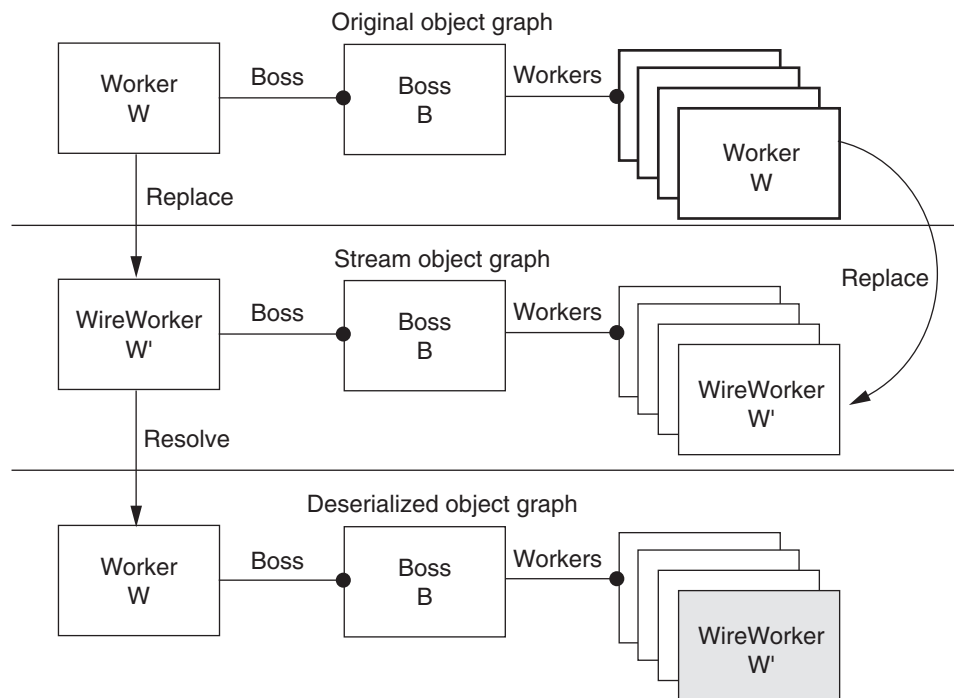


Figure 4–4 The worker object graph in transit

The resulting object hierarchy contains a `WireWorker` instance in the `Boss`'s `workers` array, almost certainly counter to the programmer's intent. This behavior can produce bizarre symptoms. Since the `workers` array is not strongly typed, the fact that the object hierarchy has changed will not be detected as a `ClassCastException` during deserialization. Later, some method of `Boss` will fail because it expects the entries in the `workers` array to be of type `PersonItf`, not `WireWorker`.

4.8.4 Taking Control of Graph Ordering

The problem with circular references and object replacement is only the most egregious example of a problem shared by all of the serialization hooks discussed so far. Serialization is designed to transparently handle object graphs, without forcing you to know or worry about the exact order in which objects in the graph are serialized. Custom serialization hooks are also designed to hide the ordering of the object graph. Unfortunately, some hooks are likely to have ordering dependencies, which leads to the need for potentially confusing ordering rules such as the object replacement rules discussed earlier. The situation calls for some way to take explicit control of the order of events during deserialization. The `registerValidation` hook, shown in Listing 4–17, addresses this need.

Listing 4–17 Object Validation APIs

```
package java.io;
public class ObjectInputStream {
    public void registerValidation(ObjectInputValidation obj,
                                int prio)
        throws NotActiveException,
               InvalidObjectException;

    //remainder omitted for brevity
}

package java.io;
public interface ObjectInputValidation {
    public void validateObject() throws InvalidObjectException
}

```

The `ObjectInputStream` class implements the method `registerValidation`. You can call this method at any time during deserialization, by passing in an `ObjectInputValidation` implementation that will be called after the entire object graph is reconstructed. The `validateObject` method has two advantages over `resolveObject`. First, because `validateObject` is called after the entire object graph has been reconstituted, the graph is in a reliable, reproducible state. Second, you can exert additional control over multiple `validateObject` calls by setting the `prio` value when registering the callback.

Callbacks with a higher `prio` value are made first, and there are no guarantees for callbacks with the same priority.

The modified version of `Boss` shown in Listing 4–18 registers a callback to fix the nested references that are not handled by object resolution. Rather than attempt to validate an individual object's state in `readObject`, `SafeBoss` calls `registerValidation` so that it can validate the object after the entire object graph has been re-created. Then, `validateObject` verifies that each field is correct and iterates over the `workers` array making sure each worker is, or is convertible to, a `PersonItf`.

Even though the `validateObject` method says nothing about replacement, it is fully capable of replacing nested objects such as those in the `workers` array. However, `validateObject` returns `void`, so it cannot replace the top-level object being validated. You will often use `resolveObject` and `validateObject` in tandem to deal with ordering dependencies caused by circular references.¹⁵

Listing 4–18 Using `ObjectInputValidation` to Control Ordering

```
public class SafeBoss implements PersonItf, Serializable,
                               ObjectInputValidation
{
    //only methods that differ from Boss included, for brevity
    private void readObject(ObjectInputStream ois)
        throws IOException, ClassNotFoundException
    {
        System.out.println("registering validation");
        ois.defaultReadObject();
        ois.registerValidation(this, 0);
    }
    public void validateObject()
        throws InvalidObjectException
    {
        System.out.println("running validation");
        if ((name == null) || (workers == null)) {
            throw new InvalidObjectException("unexpected null field");
        }
        for (int n=0; n<workers.size(); n++) {
```

15. If the object graph is very simple, it doesn't matter much which approach you use.

```

        Object o = workers.get(n);
        if (o instanceof PersonItf) continue;
        if (o instanceof WireWorker) {
            WireWorker w = (WireWorker) o;
            workers.set(n, new Worker(w.data1, w.data2));
        } else {
            throw new InvalidObjectException(
                "unexpected worker type " + o.getClass());
        }
    }
}

```

4.9 Finding Class Code

The serialization hooks discussed thus far either customize how instances are serialized or customize what data is written in the standard class metadata format. The serialization specification also includes hooks for extending or replacing the class metadata format. The `annotateClass` and `resolveClass` methods of the object streams allow arbitrary per-class payloads to be added to a stream.

While there are no restrictions on the data included in a class annotation, the primary use for this mechanism is to help the receiver locate the correct class file if it is not available locally. The sender writes a URL string for each class, and the receiver uses the URL to instantiate a `ClassLoader` if it cannot find a local definition for the class. Consider the object streams in Listing 4–19; these serialize and load objects from a location not on the classpath.

Listing 4–19 Annotating Classes with a URL

```

class AnnotatedOutputStream extends ObjectOutputStream {
    private final String url;
    public AnnotatedOutputStream(OutputStream os, String url)
        throws IOException {
        super(os);
        this.url = url;
    }
    protected void annotateClass(Class cl)
        throws IOException {
        writeObject(url);
    }
}

```

```

    }

    class ResolvingInputStream extends ObjectInputStream {
        public ResolvingInputStream(InputStream is)
            throws IOException
        {
            super(is);
        }
        protected Class resolveClass(ObjectStreamClass v)
            throws IOException, ClassNotFoundException
        {
            String url = (String) readObject();
            URL u = new URL(url);
            URLClassLoader ucl = new URLClassLoader(new URL[]{u});
            String cls = v.getName();
            System.out.println("resolving " + cls + " from " + url);
            return Class.forName(cls, true, ucl);
        }
    }

```

The `AnnotatedOutputStream` class marks every class it serializes with a URL string. The `ResolvingInputStream` class reads in the string and uses it to create a `URLClassLoader`, which then loads the class. This simple system allows the sender of an object to tell the receiver how to download the code necessary to deserialize and use the object. To flesh out this idea, you would want to add the ability to annotate different classes with different locations and cache the class loaders created during resolution.

4.9.1 Annotation in RMI

The primary customer of many advanced serialization features is Java RMI. RMI includes a full solution for dynamically downloaded code that is basically an elaborate version of the annotation classes shown in Listing 4–19. Dynamic code download makes it possible for RMI to ship serialized objects around the network without worrying about installing class files; they will be downloaded when and where needed.

As powerful as this mechanism is, it works harder than necessary due to a weakness of the serialization architecture. Reading a serialized graph is an all-or-nothing prospect. If a single class cannot be found, deserialization fails and

both the object graph and the stream become unusable. This can be particularly irritating when class files need to be downloaded. Consider the distributed workflow situation depicted in Figure 4–5. A `Message` object is passed from machine to machine, and each machine operates on a fragment of the `Message`'s contents.

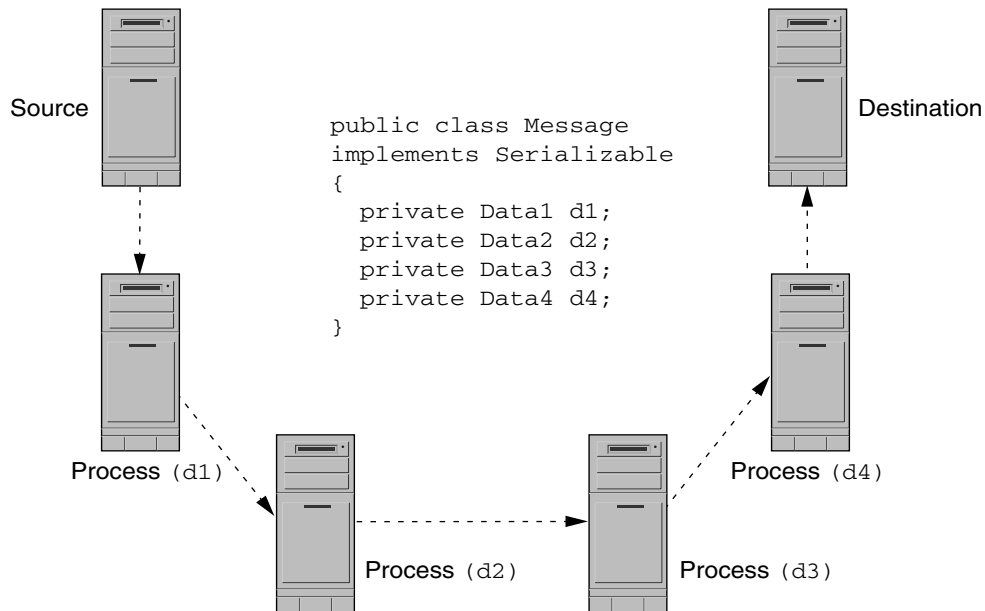


Figure 4–5 Message must be entirely deserialized at each node.

Because the `Message` is transmitted as a serialized Java object, every single class in the graph must be downloaded to every machine. This is overkill since each machine looks at only one of the `Data(N)` classes that make up the overall `Message`. It would be nice if there were some way to mark objects in a stream as “deserialize only when absolutely necessary” and then have each virtual machine deserialize and download code for only the instances actually used.

Serialization does not provide a marker for deserialize-on-demand, but you can achieve the desired effect in a straightforward manner. Simply write a wrapper class that, instead of holding a reference to an object, holds a reference to a byte array that contains the serialized form of the object. The wrapper class

can be serialized and deserialized as much as you like without ever having to touch the contents of the byte array. An object wrapped in this fashion can travel through many intermediate steps until it is needed, without there being any need to load the object's class. When you finally do need the object, you build an `ObjectInputStream` around the array and extract the object, downloading its class if necessary. RMI includes an implementation of this technique called a `MarshaledObject`.

4.9.2 RMI MarshalledObjects

The deserialize-on-demand idiom is so useful that RMI provides a complete implementation in the `java.rmi.MarshaledObject` class. The `MarshaledObject` API is simple and is shown in Listing 4-20. When you pass an object to the `MarshaledObject` constructor, that object is stored in serialized form until you later request it with the `get` method. RMI uses the `MarshaledObject` to store initialization parameters for `Activatable` objects, but you can use a `MarshaledObject` anywhere that you need to maintain the *ability* to instantiate an object at any time without actually holding a reference to the object itself.

Listing 4-20 Key MarshalledObject Methods

```
package java.rmi;
public class MarshalledObject {
    public MarshalledObject(Object o);
    public Object get();
    remainder omitted for clarity
}
```

4.10 Onward

Java metadata makes basic serialization trivial. If you mark a class `Serializable`, an `ObjectOutputStream` can discover your class's fields via reflection, and it can automate the process of writing an instance to a stream or reading it back again. This alone is quite a trick if you come from a language background that does not include metadata.

The interesting parts of Java serialization, though, are the hooks that Java provides to fine-tune how your classes transition to and from object streams. You can use the `readObject` and `writeObject` methods to add custom

per-instance data or validation to the default serialization behavior. With class annotation, you can add custom per-class data, typically to support dynamically downloading the code necessary to deserialize an object. The `registerValidation` hook copes with order dependencies by providing control at the level of an entire object graph, instead of on a per-instance basis. You can use the `serialPersistentFields` to change the metadata associated with a class, keeping the serialized form readable by all class versions as the code evolves.

With these powers, however, come some dangers. The `Externalizable` interface is tempting because it is often associated with better performance, but it accomplishes this by eliminating metadata from the stream format, and this can lead to a maintenance nightmare. The `serialVersionUID` (SUID) provides an efficient way to compare two classes for compatibility, but it gives an all-or-nothing answer. You may have to provide custom `readObject` and `writeObject` code to glue together serialized forms that are *almost* compatible and that the SUID test would have rejected.

You can replace objects during serialization, in order to control the semantics of transmission for higher efficiency or to clearly separate a class's serialization format from its in-memory format—but you have to be careful when you deal with cyclical graphs. Finally, you must be aware that serialized objects cannot be garbage collected until the stream itself is collectable, or until you call `reset`. As of SDK version 1.4, calling `close` has the same GC-friendly effects as calling `reset`.

Serialization is not a general persistence mechanism. It does not provide random access to objects embedded in the graph. For long-term storage, or for query and update operations against data, JDBC or Enterprise JavaBeans (EJB) may be more appropriate. For simple transmission of objects across space or time, serialization is a simple and flexible solution.

4.11 Resources

For more on Java serialization, you should probably begin with [Har99], which covers Java I/O in general. In particular, Chapter 11 is devoted entirely to object serialization, providing coverage that is more basic but also more systematic



than the material presented here. [Blo01] is a wide-ranging, excellent book on using Java effectively. It includes a short chapter on serialization that discusses performance and validation in a fashion complementary to this chapter. Finally, the serialization specification [Ser] is clear and concise, although it emphasizes “what” and “how” instead of “why.”

