The Java™ Tutorials

Trail: Essential Classes
Lesson: Basic I/O
Section: I/O Streams

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases.

Object Streams

Just as data streams support I/O of primitive data types, object streams support I/O of objects. Most, but not all, standard classes support serialization of their objects. Those that do implement the marker interface `Serializable`.

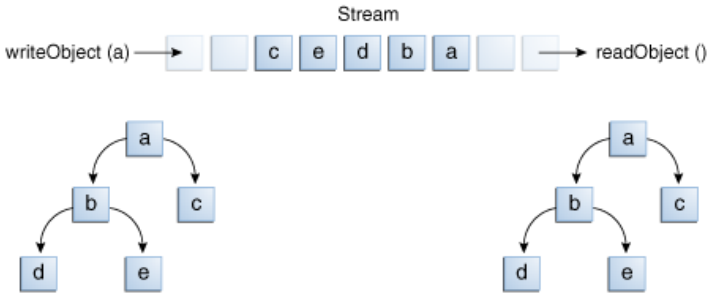
The object stream classes are `ObjectInputStream` and `ObjectOutputStream`. These classes implement `ObjectInput` and `ObjectOutput`, which are subinterfaces of `DataInput` and `DataOutput`. That means that all the primitive data I/O methods covered in Data Streams are also implemented in object streams. So an object stream can contain a mixture of primitive and object values. The `ObjectStreams` example illustrates this. `ObjectStreams` creates the same application as `DataStreams`, with a couple of changes. First, prices are now `BigDecimal`objects, to better represent fractional values. Second, a `Calendar` object is written to the data file, indicating an invoice date.

If `readObject()` doesn't return the object type expected, attempting to cast it to the correct type may throw a `ClassNotFoundException`. In this simple example, that can't happen, so we don't try to catch the exception. Instead, we notify the compiler that we're aware of the issue by adding `ClassNotFoundException` to the `main` method's `throws` clause.

Output and Input of Complex Objects

The `writeObject` and `readObject` methods are simple to use, but they contain some very sophisticated object management logic. This isn't important for a class like Calendar, which just encapsulates primitive values. But many objects contain references to other objects. If `readObject` is to reconstitute an object from a stream, it has to be able to reconstitute all of the objects the original object referred to. These additional objects might have their own references, and so on. In this situation, `writeObject` traverses the entire web of object references and writes all objects in that web onto the stream. Thus a single invocation of `writeObject` can cause a large number of objects to be written to the stream.

This is demonstrated in the following figure, where `writeObject` is invoked to write a single object named **a**. This object contains references to objects **b** and **c**, while **b** contains references to **d** and **e**. Invoking `writeobject(a)` writes not just **a**, but all the objects necessary to reconstitute **a**, so the other four objects in this web are written also. When **a** is read back by `readObject`, the other four objects are read back as well, and all the original object references are preserved.



I/O of multiple referred-to objects

You might wonder what happens if two objects on the same stream both contain references to a single object. Will they both refer to a single object when they're read back? The answer is "yes." A stream can only contain one copy of an object, though it can contain any number of references to it. Thus if you explicitly write an object to a stream twice, you're really writing only the reference twice. For example, if the following code writes an object `ob` twice to a stream:

```
Object ob = new Object();
out.writeObject(ob);
out.writeObject(ob);
```

Each `writeObject` has to be matched by a `readObject`, so the code that reads the stream back will look something like this:

```
Object ob1 = in.readObject();
Object ob2 = in.readObject();
```

This results in two variables, `ob1` and `ob2`, that are references to a single object.

However, if a single object is written to two different streams, it is effectively duplicated — a single program reading both streams back will see two distinct objects.

Previous page: Data Streams
Next page: File I/O (Featuring NIO.2)