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.

Data Streams

Data streams support binary I/O of primitive data type values (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`) as well as String values. All data streams implement either the `DataInput` interface or the `DataOutput` interface. This section focuses on the most widely-used implementations of these interfaces, `DataInputStream` and `DataOutputStream`.

The `DataStreams` example demonstrates data streams by writing out a set of data records, and then reading them in again. Each record consists of three values related to an item on an invoice, as shown in the following table:

Order in record	Data type	Data description	Output Method	Input Method	Sample Value
1	`double`	Item price	`DataOutputStream.writeDouble`	`DataInputStream.readDouble`	`19.99`
2	`int`	Unit count	`DataOutputStream.writeInt`	`DataInputStream.readInt`	`12`
3	`String`	Item description	`DataOutputStream.writeUTF`	`DataInputStream.readUTF`	`"Java T-Shirt"`

Let's examine crucial code in `DataStreams`. First, the program defines some constants containing the name of the data file and the data that will be written to it:

```
static final String dataFile = "invoicedata";

static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
static final int[] units = { 12, 8, 13, 29, 50 };
static final String[] descs = {
    "Java T-shirt",
    "Java Mug",
    "Duke Juggling Dolls",
    "Java Pin",
    "Java Key Chain"
};
```

Then `DataStreams` opens an output stream. Since a `DataOutputStream` can only be created as a wrapper for an existing byte stream object, `DataStreams` provides a buffered file output byte stream.

```
out = new DataOutputStream(new BufferedOutputStream(
              new FileOutputStream(dataFile)));
```

`DataStreams` writes out the records and closes the output stream.

```
for (int i = 0; i < prices.length; i ++) {
    out.writeDouble(prices[i]);
    out.writeInt(units[i]);
    out.writeUTF(descs[i]);
}
```

The `writeUTF` method writes out `String` values in a modified form of UTF-8. This is a variable-width character encoding that only needs a single byte for common Western characters.

Now `DataStreams` reads the data back in again. First it must provide an input stream, and variables to hold the input data. Like `DataOutputStream`, `DataInputStream` must be constructed as a wrapper for a byte stream.

```
in = new DataInputStream(new
            BufferedInputStream(new FileInputStream(dataFile)));

double price;
int unit;
```

```
String desc;
double total = 0.0;
```

Now `DataStreams` can read each record in the stream, reporting on the data it encounters.

```
try {
    while (true) {
        price = in.readDouble();
        unit = in.readInt();
        desc = in.readUTF();
        System.out.format("You ordered %d" + " units of %s at $%.2f%n",
            unit, desc, price);
        total += unit * price;
    }
} catch (EOFException e) {
}
```

Notice that `DataStreams` detects an end-of-file condition by catching `EOFException`, instead of testing for an invalid return value. All implementations of `DataInput` methods use `EOFException` instead of return values.

Also notice that each specialized `write` in `DataStreams` is exactly matched by the corresponding specialized `read`. It is up to the programmer to make sure that output types and input types are matched in this way: The input stream consists of simple binary data, with nothing to indicate the type of individual values, or where they begin in the stream.

`DataStreams` uses one very bad programming technique: it uses floating point numbers to represent monetary values. In general, floating point is bad for precise values. It's particularly bad for decimal fractions, because common values (such as `0.1`) do not have a binary representation.

The correct type to use for currency values is `java.math.BigDecimal`. Unfortunately, `BigDecimal` is an object type, so it won't work with data streams. However, `BigDecimal` *will* work with object streams, which are covered in the next section.

Previous page: I/O from the Command Line
Next page: Object Streams