The File Class, Binary I/O, and Serialization

You should already have familiarity with reading and writing text files in Java. If not, review text I/O from a textbook or website. Here we introduce the File class, reading and writing binary files, and using the Serializable interface.

File Class

The File class is part of the java.io package. The class contains methods that allow you to check various properties of a file, such as whether a file exists, can be written to, etc. The constructor takes the pathname of the file directory as an argument. Once the object has been created we can use the following methods:

public boolean exists()	True if the file exists with the given pathname
public boolean canRead()	True if the file is readable
public boolean setReadOnly()	Sets the file to be read only
public boolean canWrite()	True if we have permissions to write
public boolean delete()	Removes the file, false if unable to delete the file or directory
public boolean createNewFile()	Creates a new empty file
public String getAbsolutePath()	Gets the full path
public String getPath()	Returns the abstract path name as a string
public boolean isFile()	True if abstract path is a file
public boolean isDirectory()	True if abstract path is a folder
public File[] listFiles()	Array of all files/folders in the path

Sample code:

```
import java.io.File;

public class Foo
{
      public static void main(String[] args)
      {
            File f = new File("Foo.java");
            System.out.println(f.exists());
            System.out.println(f.getPath());
            System.out.println(f.getAbsolutePath());
            System.out.println(f.isDirectory());
```

```
                f = new File("c:\\");
                for (File subfile : f.listFiles())
                        System.out.println(subfile.getName());
        }
}
```

Several Java classes return or use a File object, so it is good to have some familiarity with what it can do.

Binary Files

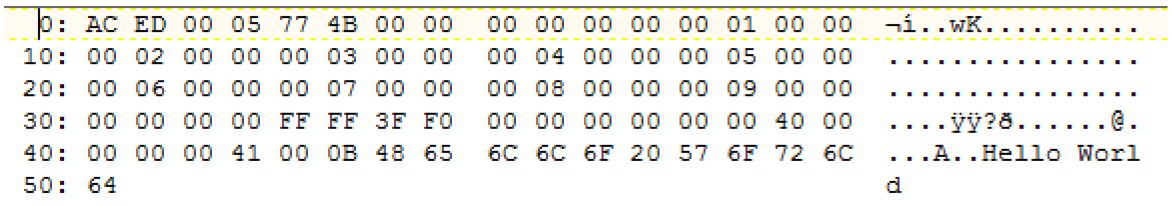
Binary files store data in the same format that is used in computer memory to store variables, so it can be more efficient than storing as text. As a binary file, it is generally not viewable with a text editor (it will interpret the binary as text which will usually be junk). The preferred stream classes for processing binary files are **ObjectInputStream** and **ObjectOutputStream**. Each has methods to read/write data one byte at a time and can automatically convert numbers and characters to bytes.

First, here is an example how to write some simple data to a binary file using ObjectOutputStream.

```
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class Foo
{
        public static void main(String[] args)
        {
                try
                {
                        ObjectOutputStream o = new ObjectOutputStream(
                                     new FileOutputStream("numbers.bin"));
                        for (int i = 0; i < 10; i++)
                        {
                                o.writeInt(i);
                        }
                        o.writeLong(65535);
                        o.writeDouble(1);
                        o.writeFloat(2);
                        o.writeChar('A');
                        o.writeUTF("Hello World");
                        o.close();
                }
                catch (IOException e)
                {
                        System.out.println(e);
                }
        }
}
```

This creates a binary file containing the data written as a float, char, String, etc.



```
 0: AC ED 00 05 77 4B 00 00  00 00 00 00 00 01 00 00  ¬í..wK..........
10: 00 02 00 00 00 03 00 00  00 04 00 00 00 05 00 00  ................
20: 00 06 00 00 00 07 00 00  00 08 00 00 00 09 00 00  ................
30: 00 00 00 00 FF FF 3F F0  00 00 00 00 00 00 40 00  ....ÿÿ?ð......@.
40: 00 00 00 41 00 0B 48 65  6C 6C 6F 20 57 6F 72 6C  ...A..Hello Worl
50: 64                                                d
```

Here is a program that could read the data from numbers.bin back into variables:

```
import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.EOFException;
import java.io.FileNotFoundException;

public class Foo
{
      public static void main(String[] args)
      {
            try
            {
                  ObjectInputStream o = new ObjectInputStream(
                        new FileInputStream("numbers.bin"));
                  for (int i = 0; i < 10; i++)
                  {
                        int num;
                        num = o.readInt();
                        System.out.println("Read: " + num);
                  }
                  long l = o.readLong();
                  System.out.println(l);
                  double d = o.readDouble();
                  System.out.println(d);
                  float f = o.readFloat();
                  System.out.println(f);
                  char c = o.readChar();
                  System.out.println(c);
                  String s = o.readUTF();
                  System.out.println(s);
                  int noInt = o.readInt();  // Throws EOF Exception
                  o.close();
            }
            catch (FileNotFoundException e)
            {
                  System.out.println(e);
```

```
				}
				catch (EOFException e)
				{
					System.out.println("Reached the end of file!");
				}
				catch (IOException e)
				{
					System.out.println(e);
				}
			}
		}
```

This program does the reverse and inputs data into variables from the binary file. Note the way to test for end of file – we check for an EOFException which is thrown if we try to read past the end of the file.

Binary I/O of Objects via Serializable

There is also a **readObject** and **writeObject** method in the ObjectInputStream and ObjectOutputStream classes. These do what you expect, but only if the object implements the **java.io.Serializable** interface. The Serializable interface is another marker interface, like the Cloneable interface. It doesn't have any methods, but serves as a marker to the JVM to behave in a special way.

To make a class implement the Serializable interface you just add "implements Serializable" as you would for any interface. You can now use the readObject and writeObject methods on that class. However, any instance variables must also implement the Serializable interface. With a serializable class, Java will assign a serial number to each object of the class that it writes to a stream of type ObjectOutputStream. This serial number is then used to re-create the class when it is used with ObjectInputStream. This means that if two variables contain references to the same object and you write the objects to a file and later read them from the file, then the two objects that are read will again be references to the same object. So it is like the structure of your objects are re-created when read back in.

Here is a short example with a Foo class:

```
	public class Foo
	{
		private int x;
		private String s;

		public Foo()
		{
			this(100,"hello");
		}
		public Foo(int x, String s)
		{
			this.x = x;
```

```
                this.s = s;
            }
            public String toString()
            {
                return s + " " + x;
            }
        }
```

Normally, or at least with what we have covered, if we want to save an instance of the Foo class we could save the integer x, the String s, into a binary or text file, then read the int and the String and use it to re-create the class.

With serialization we can do this all in one shot! First here is how we could save the object. We just add the Serializable interface:

```
        import java.io.Serializable;
        public class Foo implements Serializable
        {
            private int x;
            private String s;

            public Foo()
            {
                this(100,"hello");
            }
            public Foo(int x, String s)
            {
                this.x = x;
                this.s = s;
            }
            public String toString()
            {
                return s + " " + x;
            }
        }
```

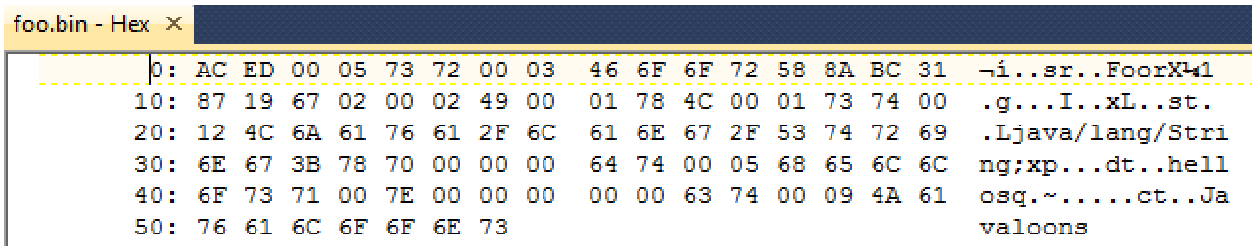
Here is some code that could go in main or elsewhere to save a couple of Foo objects:

```
            import java.io.ObjectOutputStream;
            import java.io.FileOutputStream;
            import java.io.IOException;
            import java.io.Serializable;

            …
```

```
                      try
                      {
                           ObjectOutputStream o = new
                            ObjectOutputStream(
                                     new FileOutputStream("foo.bin"));
                           Foo foo1 = new Foo();
                           Foo foo2 = new Foo(99,"Javaloons");
                           o.writeObject(foo1);
                           o.writeObject(foo2);
                           o.close();
                      }
                      catch (IOException e)
                      {
                           System.out.println(e);
                      }
```

This creates a file, foo.bin, on the disk, that contains serialized binary versions of foo1 and foo2.



```
 0: AC ED 00 05 73 72 00 03  46 6F 6F 72 58 8A BC 31  ¬í..sr..FoorX¼1
10: 87 19 67 02 00 02 49 00  01 78 4C 00 01 73 74 00  .g...I..xL..st.
20: 12 4C 6A 61 76 61 2F 6C  61 6E 67 2F 53 74 72 69  .Ljava/lang/Stri
30: 6E 67 3B 78 70 00 00 00  64 74 00 05 68 65 6C 6C  ng;xp...dt..hell
40: 6F 73 71 00 7E 00 00 00  00 00 63 74 00 09 4A 61  osq.~.....ct..Ja
50: 76 61 6C 6F 6F 6E 73                              valoons
```

To read the serialized binary files back in, we use readObject:

```
           import java.io.ObjectInputStream;
           import java.io.FileInputStream;
           import java.io.FileNotFoundException;
           import java.io.IOException;
           import java.io.Serializable;
           …
                      try
                      {
                           ObjectInputStream o = new
                             ObjectInputStream(
                               new FileInputStream("foo.bin"));
                           // Note the typecasts below
                           Foo foo1 = (Foo) o.readObject();
                           Foo foo2 = (Foo) o.readObject();
                           System.out.println(foo1);
                           System.out.println(foo2);
                           o.close();
                      }
                      catch (ClassNotFoundException e)
```

```
                    {
                         System.out.println(e);
                    }
                    catch (FileNotFoundException e)
                    {
                         System.out.println(e);
                    }
                    catch (IOException e)
                    {
                         System.out.println(e);
                    }
```

Note that we need to catch ClassNotFoundException and typecast the object returned by readObject.

The output is:

```
hello 100
Javaloons 99
```

If we have a complex class this makes saving and reading it much easier!   All your instance variables must also be serializable. If you use the standard Java classes then this is already done for you. For example we could have the following Foo class:

```
public class Foo implements Serializable
{
     private int x;
     private String s;
     ArrayList<String> list;

     public Foo()
     {
          this(100,"hello");
     }
     public Foo(int x, String s)
     {
          this.x = x;
          this.s = s;
          list = new ArrayList<>();
          list.add("foo");
          list.add("bar");
          list.add("zot");
     }
     public String toString()
     {
          String temp = "";
          for (String name : list)
               temp += name + " ";
          return temp + " " + s + " " + x;
          }
      }
```

We can save the whole Foo object with serialization, and read it back in, the ArrayList included, all in one fell swoop.

One final topic you may be interested in is random access binary files. In a random access binary file there is a file pointer and you can jump (seek) to different parts of the file. Typically you will need to have a fixed record size so you can calculate the proper offset of a data record of interest. See a Java reference for more details.