File I/O In Java

Prof. Chris Jermaine
cmj4@cs.rice.edu

Our Simple Java Programs So Far

- Aside from screen I/O...
  - — ...when they are done, they are gone
- They have no lasting effect on the world
  - — When the program leaves the JVM, it is gone

Our Simple Java Programs So Far

- Aside from screen I/O...
  - — ...when they are done, they are gone
- They have no lasting effect on the world
  - — When the program leaves the JVM, it is gone
- Not ideal for several reasons... precludes
  - — Checkpointing long-running computations
  - — Saving results/state for later use
  - — Communication between programs

How To Communicate With the World?

- Via “I/O” (short for “In/Out”)
- I/O is always about communication
  - — You are writing (reading) data for someone else to use
- Most I/O is file-oriented
  - — This will be our focus
- But not all
  - — Ex: try to use same interface for inter-process communication via pipes

Two Kinds Of I/O In Java

- And in most PLs as well
  - — Raw binary I/O
  - — Character I/O

Raw Binary I/O

- Binary I/O is useful when you want to communicate data...
  - — To a computer program
  - — When you know no human will ever look at the data
- Might want to communicate with youself (save state)
  - — Or with another program
  - — The “other” program could be running at some time in the future
  - — Or at the same time (might be writing a parallel/distributed program)
- Though this latter case is outside the scope of the class!
- Advantage compared to character I/O
  - — Data is much more compressed

Raw Binary I/O

- Classes for raw binary I/O are descended from
  - — “InputStream”
  - — “OutputStream”
- If doing file I/O, use
  - — “FileInputStream”
  - — “FileOutputStream”

Example (stolen from Oracle website)

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

public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("origin.txt");
            out = new FileOutputStream("output.txt");
            int c; // note: c is just a container for the bits!

            while ((c = in.read()) != -1) {
                out.write(c);
            }

        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Serialization/Deserialization

- Often you're gonna have an object you want to save to a file
  - — Or that you want to send to another program
- Assuming you don't care about it being human readable
  - — You want to write it in binary to a file
- How to do this?
- Could write a method to encode it as an array of ints
  - — Then use a variant of the code from the last slide to write/read it
- Conveniently, all standard classes do this for you!
  - — Almost trivial to use

Consider the Following, Silly Class

```
import java.io.FileInputStream;public class TestClass implements java.io.Serializable {
   private String name;
   private ArrayList <TestClass> children;

   public TestClass (String nameToUse) {
     name = nameToUse;
   }

   public TestClass (int level) {
     children = new ArrayList <TestClass> ();
     if (level == 0) {
       children.add (new TestClass (new String ("joe")));
       children.add (new TestClass (new String ("sam")));
       children.add (new TestClass (new String ("sarah")));
       children.add (new TestClass (new String ("alex")));
     } else {
       children.add (new TestClass (level - 1));
       children.add (new TestClass (level - 1));
   }}

   public void print () {
     if (name != null) {
       System.out.format ("%s", name);
     } else {
       for (TestClass i : children) {
         System.out.format ("<");
         i.print ();
         System.out.format (">");
}}}}
```

Here's An Example of What It Does

```
public void testX() {

  TestClass temp = new TestClass (3);
  temp.print ();
}
```

- Output is:

```
<<<<joe><sam><sarah><alex>><<joe><sam><sarah><alex>>><<<joe><sam><sarah><alex>><<joe><sam>
<sarah><alex>>>><<<<joe><sam><sarah><alex>><<joe><sam><sarah><alex>>><<<joe><sam><sarah><a
lex>><<joe><sam><sarah><alex>>>>
```

Very Easy To Save An Instance For Posterity

```
public void testY () {
  TestClass temp = new TestClass (3);
  try {
    FileOutputStream fileOut = new FileOutputStream ("output");
    ObjectOutputStream out = new ObjectOutputStream (fileOut);
    out.writeObject (temp);
    out.close ();
    fileOut.close ();
  } catch(IOException i) {
    i.printStackTrace ();
  }
}
```

And Just As Easy To Recover It

```
public void testZ () {
      TestClass temp = null;
       try {
          FileInputStream fileIn = new FileInputStream ("output");
          ObjectInputStream in = new ObjectInputStream (fileIn);
          temp = (TestClass) in.readObject();
          in.close();
          fileIn.close();
      } catch (IOException i) {
          i.printStackTrace();
          return;
      } catch (ClassNotFoundException c){
          System.out.println ("didn't find the TestClass class as expected");
          c.printStackTrace ();
          return;
      }
      temp.print ();
}
```

- Output is the same as “testX”:

```
<<<<joe><sam><sarah><alex>><<joe><sam><sarah><alex>>><<<joe><sam><sarah><alex>><<joe><sam>
<sarah><alex>>>><<<<joe><sam><sarah><alex>><<joe><sam><sarah><alex>>><<<joe><sam><sarah><a
lex>><<joe><sam><sarah><alex>>>>
```

Ser./Deser.: the Finer Points

- To get this to work
  - — just need to implement “java.io.Serializable” interface
  - — Java does the rest
- In the (somewhat) rare case...
  - — ...where just writing the object contents won’t suffice
  - — Example: you want to store some other program state with an object
  - — Or you don’t want to store ***all*** of the object’s contents due to space
  - — You can over-ride “writeObject” and “readObject”
- Be aware: only somewhat robust accross different object versions
  - — Can’t serialize an object...
  - — Then add a data field and re-compile
  - — Then deserialize using the new version of the code

Character I/O (example stolen from Oracle)

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
    public static void main(String[] args) throws IOException {
        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream =
                new BufferedReader(new FileReader("input.txt"));
            outputStream =
                new PrintWriter(new FileWriter("output.txt"));
            String l;
            while ((l = inputStream.readLine()) != null)
                outputStream.println(l);
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

The Character I/O Wrappers

- Notice we wrap “FileReader” in “BufferedReader”
- And we wrap “FileWriter” in “PrintWriter”
- Why?
  - — “FileReader” and “FileWriter” provide low-level character I/O
  - — While the other two give much higher-level interfaces

In General

- Are a very complex web of I/O classes in Java
  - — This lecture gives enough to get started
  - — You can spend a long time understanding them all

Questions?