The University of Melbourne
Department of Computer Science and Software Engineering
433-254 Software Design
Semester 2, 2003

Answers for Tutorial 6
Week 7

1. What are similarities and differences between interfaces and classes?

   Sample Answer:
   Please refer to the sample answer for exercise 5, tutorial 5, week 6.

2. How interfaces can be used to support multiple-inheritance (give an example)?

   Sample Answer:

   As discussed earlier, Java does not allow a class to inherit more than one class as in C++. That is,

   ```
   class Test extends aClass, bClass{ // Error !!
   ```

   There are situations where multiple inheritance can be useful but it can also lead to problems. For example, a reference to a method with the same name in both classes needs a system for making it explicit as to which class should be used.

   Interfaces provide most of the advantages of multiple inheritance with fewer problems. An interface is basically an abstract class but with all of the methods abstract. (The methods in an interface do not need the explicit abstract modifier since they are abstract by definition.)

   A class *implements* an interface rather than extends it. Any class that implements the interface must override all the interface methods with it's own methods.

   (In the case of identical method names in two interfaces, it's irrelevant since both methods are abstract and carry no code body. In a sense, both are overridden by the single method of that name in the implementing class.)

3. Compare File manipulation steps in C and Java with a suitable example.

   Sample Answer:
   In C, all file related information is captured in an standard data structure called FILE. In order to manipulate a file, a file handle of type FILE *, must be obtained (using standard fopen function). Once a file handle obtained, it can be used by verity of standard file I/O functions (e.g. fread, fscanf, fread, etc.) to manipulate the corresponding file. At the end, the file must be closed by calling fclose function.

Java I/O centers around the concept of the *stream*, which is an in or out flow of data. For example, an output stream carries data to a file and an input stream brings data from a file. The base file stream classes are

`FileInputStream` - binary file input base class
`FileOutputStream` - binary file output base class

`FileReader` - read text files
`FileWriter` - write to text files

An input stream from a file can be created with:

```
File fileIn = new File("data.dat");
FileInputStream in = new FileInputStream(fileIn);
```

If the file doesn't exist, this will throw an exception.

Usually, the `FileInputStream` object is wrapped with another stream to obtain greater functionality. For example, `BufferedInputStream` is used to smooth out the data flow.

Similarly, output streams to files are opened like:

```
File fileOut = new File("tmp.dat");
FileOutputStream out = new FileOutputStream(fileOut);
```

If the file doesn't exist, it will be created.

Also here, the output streams are wrapped with one or more other streams to obtain greater functionality such as `BufferedOutputStream`.

To create a directory use the *mkdir()* method in `File` class.

To append to an existing file use the overloaded constructor:

```
File fileOut = new File("old.dat");
FileOutputStream out = new FileOutputStream(fileOut,true);
```

where the second argument indicates that the file should be opened for appending. (`RandomAccessFile` can also be used for appending.)

The following example illustrates how to use the FileReader stream to read character data from text files.

```
import java.io.*;
import java.util.*;
//  Demonstrate reading text from a file.
public class TxtInFile
```

```
{
  public static void main(String arg[])
  {
    File file = new File("TextFileInput_App1.java");

    int numBytesRead=0;
    try {
      FileReader fileReader = new FileReader(file);
      // Read the bytes directly and count them.
      while (fileReader.read() != -1)
         numBytesRead++;
    }
    catch (IOException e){
        System.out.println( "IO error:" + e );
    }
    System.out.println("Number bytes read  = "+ numBytesRead);
    // Compare to the length reported by File class.
    System.out.println("File.length() = "+ file.length() );
  }
}
```

4. Write statements to create data streams for the following operations:
   a. Reading primitive data from a file.
   b. Writing primitive data to a file.

Sample Answer:

a) In the following example, we can read a binary file by opening the file with a `FileInputStream` object. Then we wrap this with a `DataInputStream` class to obtain the many *readXxx()* methods that are very useful for reading the various primitive data types.

```
import java.io.*;
import java.util.*;
//Demonstrate reading primitive type values to a binary file.
public class BinInFile
{
  public static void main(String arg[])
  {
    File file = new File("numerical.dat");

    try {
      // Wrap the FileInputStream with a DataInputStream
      //  to obtain its readDouble() method
      FileInputStream fileInput = new FileInputStream(file);
      DataInputStream dataIn = new DataInputStream( fileInput );
      while(true)
      {
         double d = dataIn.readDouble();
         System.out.println( “double = " + d );
         int i = dataIn.readInt();
         System.out.println( “int = " + i );
      }

    }
```

```
      catch (EOFException eof){
         System.out.println( "End of File");
      }
      catch (IOException e){
         System.out.println( "IO error: " + e );
      }
    }
  }
```

b) In this example, we open the file with the binary `FileOutputStream` class. Then we wrap it with the `DataOutputStream` class that contains many useful methods for writing various primitive types.

```
import java.io.*;
import java.util.*;
//Demonstrate writing primitive type values to a binary file.
public class BinOutFile
{
  public static void main(String arg[])
  {
    File file = new File("numerical.dat");
    try {
     // Wrap the FileOutputStream with a DataOutputStream
     // to obtain its writeDouble() method
     FileOutputStream fileOut = new FileOutputStream(file);
     DataOutputStream dataOut = new DataOutputStream(fileOut);
     dataOut.writeDouble(3.14);
     dataOut.writeInt(7);
    }
    catch (IOException e){
       System.out.println( "IO error: " + e );
    }
  }
}
```

5. Write and discuss a program to create a sequential file that could store details about five products. Details include product code, cost, and the number of items available. These details are provided through the keyboard.

Sample Answer:

```
import java.io.*;
import java.util.*;
//  Demonstrate reading text from a file.
public class Products
{
  public static void main(String arg[])
  {
    File file = new File("products");
    // Read the bytes directly and count them.
    try
    {
      FileOutputStream fos = new FileOutputStream(file);
      DataOutputStream dos = new DataOutputStream(fos);
```

```
        DataInputStream   dis = new DataInputStream(System.in);
        StringTokenizer st;

        for( int i=0; i<5; i++){
              System.out.print("Please enter the product code: ");
              st = new StringTokenizer(dis.readLine());
              String pnum = new String(st.nextToken());

              System.out.print("Please enter the cost: ");
              st = new StringTokenizer(dis.readLine());
              double cost = new Double(st.nextToken()).doubleValue();

              System.out.print("Please enter the quantity: ");
              st = new StringTokenizer(dis.readLine());
              int qty = new Integer (st.nextToken()).intValue();
              dos.writeBytes(pnum);
              dos.writeDouble(cost);
              dos.writeInt(qty);
        }
        dos.close();
      }
      catch (IOException e)
      {
          System.out.println( "IO error:" + e );
      }
    }
}
```