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

Answers for Tutorial 5
Week 6

1. Discuss arrays and discuss them with the following example: Write a statement to declare and instantiate an array to hold marks obtained by different students in different subjects in a class. Assume that there are up to 300 students in a class and they take up to 6 subjects.

   Sample Answer:
   An array is an ordered collection of a fixed number of elements of the same type. These elements consist of either primitive values or references to objects. Here is a possible declaration and instantiation statement for the above example:

   ```
   float[][] marks = new float[300][6];
   ```

2. How does String class differ from the StringBuffer class? Discuss with suitable examples.

   Sample Answer:

   A StringBuffer is a string that can be modified.

   The StringBuffer class is thread-safe and its methods are synchronized. If methods are called on the same string by multiple threads, then one method call must finish before another is begun.

   StringBuffers are used to implement many of the methods in the string class.

   Example:

   ```
   public String reverseIt(String source) {
       int c, len = source.length();
       StringBuffer dest = new StringBuffer(len);
       for (c = (len - 1); c >= 0; c--) {
           dest.append(source.charAt(c));
       }
       return dest.toString();
   }
   ```

3. What is a vector and how does it differ from arrays?

   Sample Answer:

A vector is an ordered collection of elements just like an array, but more flexible. A vector can store variable number of elements of potentially different types. A vector can grow and shrink in size based on the number of elements that it contains.

4. What are applications of Wrapper classes?

   Sample Answer:

   One of the applications of wrapper classes is to convert the value of a primitive data type (e.g. int, float, etc.) into an object. An instance of a wrapper contains, or wraps, a primitive value of the corresponding type.

   Wrappers allow for situations where numerical values are needed but objects instead of primitives are required. For example, a very useful tool is as the Vector class. However, the elements of a Vector instance are object references. So if one wants to use a Vector to hold a list of numbers, the numbers must be wrapped in instances of their corresponding type

   The wrapper classes also provide various tools such as constants (the smallest and largest possible int value, for example) and static methods. You will often use wrapper methods to convert a number type value to a string or a string to a number type.

   Another application of wrapper classes is to wrap around objects in order to provide more functionality and/or flexibility. Typically an instance of a lower level class is created and then it is wrapped inside more specialized stream instances by passing it to the wrapper via a constructor argument. For instance, the Java I/O framework uses wrapper classes to build specialized I/O streams. Conceptually the aim is to provide great flexibility and modularity by *wrapping* streams with classes that provide particular capabilities as needed. Here is a complete example for reading from a file using object wrappers:

```
import java.io.*;
class FileReadTest {
     public static void main (String[] args) {
          FileReadTest f = new FileReadTest();
          f.readMyFile();
     }

     void readMyFile() {
        DataInputStream dis = null;
        String record = null;
        int recCount = 0;
        try {
           File f = new File("mydata.txt");
           FileInputStream fis = new FileInputStream(f);
           BufferedInputStream bis = new
                                BufferedInputStream(fis);
           dis = new DataInputStream(bis);
           while ( (record=dis.readLine()) != null ) {
              recCount++;
```

```
                System.out.println(recCount + ": " + record);
             }
          } catch (IOException e) {
             // catch io errors from FileInputStream or readLine()
             System.out.println("Uh oh, got an IOException error!"
                                                    + e.getMessage());
          } finally {
             // if the file opened okay, make sure we close it
             if (dis != null) {
          try {
            dis.close();
          } catch (IOException ioe) {
          }
             }
          }
       }
 }
```

5. What are major similarities and differences between interfaces and classes? Illustrate with suitable example.

   Sample Answer:

   An interface can be thought as a fully abstract class (a class with all its methods being abstract). However, when defining an interface there is no need to use keyword *abstract* to explicitly define its methods as abstract, because all in an interface, methods are abstract by default.

   Interface methods cannot be declared static. An interface is abstract by definition and therefore cannot be instantiated.

   Interfaces can inherit from each other just the same way as classes.
   An interface cannot implement any methods, whereas an abstract class can.

   A class may implement as many interfaces as it needs. But, obviously, an interface can not implement another interface, why?