CSCI 475 # Learning Java Exercise #2 Fall 2004

More Java

Objectives:

1. To explore some differences between Java and C++.
2. To write Java applications that use methods and classes.

WARNING: Our Java textbook is the sixth edition and covers the new release of Java (Java 1.5). We are currently running Java 1.4. I have asked for Java 1.5 to be installed. There are significant differences between Java 1.4 and Java 1.5. Be warned that some of the examples in the sixth edition will not run under Java 1.4.

Preparation: Before Exercise read the following chapters in *Java: How to Program* by Deitel and Deitel, fourth edition: Chapters 6-10 (6-11 in fifth; 6-10, 29 in sixth).

Assignment:

This is a collection of small exercises that point out some novel features in Java as well as some pitfalls that some C++ programmers have in programming in Java.

1. Primitive Types Vs. Objects:

Java has primitive types, i. e., **int**, **boolean**, **byte**, **char**, **float** and **double** which have some subtle differences from C++ (See page 183 in fourth edition (153 in fifth; Appendix D in Sixth) of Java Text). For examples, **int** is always 32 bits in Java whereas in C++ it depends on the computer platform. In Java, **byte** is 8 bits while **char** is 16 bits to store international characters. The use of the primitive types are cleaner than in C++. For example, the **if**, and **while** statements require a boolean expression whereas in C++ one could use an **int**. Therefore, in Java you would say

```
while(true) {
}
```

instead of **while(1)**.

Java makes major distinctions between primitive types and objects. Primitive types can be just declared while objects such as arrays must use **new**. **String** is an exception – it is an object where you do not need to use **new**. In Java you use **new** a lot more than in C++. For example, to declare an array you must use **new** as follows:

```
int i;
int a[];  //in Java can’t specify array dimension in a type expression
a = new int[10];

i = 2;
a[i] = 7;

System.out.println( "i is " + i + " a[i] is  " + a[i]);
```

2. Passing Parameters to a Method:

Passing *primitive types* to a Java method (function in C++) is *always* done by call by value. There is no call by reference mechanism such as the & symbol in C++. *Objects* always pass their reference to the method (You can think of this as a pointer to the object.) Remember from last week that Java does have pointers (references) but you can't do arithmetic on references and there is no indirect operator (*) in Java as in C++. See page 329 fourth (293 in fifth; 306 in sixth) edition of Java text.

Write a Java application that allows the user to enter up to 20 integer grades into an array from **System.in**. (See Marvin Solomon's web pages if you need help. There is a link to them on the CS479 web pages.) Stop the loop by typing in -1. Your **main** method should call an Average method that returns the average of the grades. Use the **DecimalFormat** class (see page 171, 381 fourth (142 in fifth. In sixth, see Chapter 29 for some new Java 1.5 features to use instead of DecimalFormat.) edition of Java Text and the **System.out.println** method to print the average to 2 decimal places.

Hint: To make the Average method act like a free function in C++, make it **static**.

A static method is called by using *Class-name.method-name*() and not an object name. And all the calls use the same state of instance variables. Be careful with the use of **static** keyword. Normally we avoid its use except with **main**.

3. A Common Design Pattern for Java Applications:

Java has no top-level or global variables or functions. A Java program is *always* one or more classes. A file may contain several classes but only one can be public and that class **must** have same name as the file with **.java** extension. A class without a qualifier, e.g., the keyword `public`, is known only to the current package. Only two things may appear before the first class construct **-** **package** and **import** statements.

Because of this requirement that a Java program must be a set of classes, many programmers use a common design pattern for a Java application as shown below:

```
class Ex2Part3 {

    // Instance objects (data members in C++) traditionally after class.
    // Used to communicate information across the class's methods.
    private int a;

    // Constructor
    Ex2Part3 ()
    {
        a = 7;
    }

    // Other methods

    void Print()
    {
        System.out.println("a is " + a);
    }

    // main method
    public static void main ( String args [])
    {
        // Create a Ex2Part3 object called p
        // which automatically calls the constructor.
        Ex2Part3 p = new Ex2Part3();

        // Call other methods as needed.
        p.Print();
    }
}
```

The idea here is that the **main** method creates an object of the class which automatically calls the proper constructor then uses the object to call other methods. Notice how private instance variables are used to communciate objects across methods. To C++ programmers this structures may seem a little strange but it is very common in Java programs. You need to become familiar with it as you will use it often.

Copy your Java application of Exercise 2 into a new file and rewrite it to use the above design pattern. No longer make the Average method **static**.

4. Exceptions:

In many places, Java requires you to use exceptions such as when reading input with the method **readLine**. Exceptions in Java are really handy and you should learn to be comfortable in using them.

Copy the program in Exercise 3 to a new file and add exception handling code to catch the exception thrown by the **Integer.parseInt(line)** method when the user types in a non-integer like “cat”. Modify the code such that your program tells the user that what they typed was not legal and to retype.

See Chapter 14 fourth (Chapter 15 in fifth; Chapter 13 in sixth) edition of the Java text for information on exceptions. See especially the tables on pages 821-824 fourth (tables removed in fifth, see page 750; page 649 in sixth) edition for the proper exception class to use.

5. Strings:

Strings are true objects in Java. **String**s are different from the string class in C++ libraries in that Java **String**s are immutable, i. e., you can’t modify the value of a **String**. (See page 429 fourth (508 in fifth; 1352 in sixth) edition and Chapter 10 (Chapter 11 in fifth; Chapter 29 in sixth) edition of Java text.) For example, that means you can’t alter the third character in a **String**. However, you can reassign a **String** object a new value such as shown below:

```
String s1, s2;

s1 = "WOW";
s2 = "BOW";
s1 = s2 + " " + s1;

System.out.println(s1);
```

If you want to modify a string, use the **StringBuffer** class (See page 559 fourth (523 in fifth; 1364 in sixth) edition in Java text).

The most common error with strings is when comparing them. The following is probably not what the programmer intended.

```
String s1;

if( s1 == "WOW" )  // WRONG!
```

```
        {
             System.out.println(s1 + s2);
        }
```

This compares the two references (pointers) for equality! With **String** use the **equals** method.

```
        String s1;

        if( s1.equals("WOW") )
        {
             System.out.println(s1 + s2);
        }
```

Any Java *object* that can be compared for equality will have an **equals** method. And if you write your own classes where you test for equality, you should name your method **equals**.

Copy your program of Exercise 4 to a new file and change the program to stop the loop when the user types the word "done" instead of -1.

6. Using the Java Math Class Methods and the Java API:

Spend some time and focus on the **Math** class methods on pages 249-250 fourth (219 in fifth; 235 in sixth) edition to see what is available. On page 261 fourth (230 in fifth; 249 in sixth) edition, you can learn how to use randum numbers in Java.

Bookmark in your browser the following URL for the Java 2, v 1.4 Application Programming Interface (API):

```
      http://java.sun.com/j2se/1.4/docs/api/index.html
```

You should learn how to extract useful information from the API.

7. Design Your Own Class:

In Java, all *objects* extend the class **Object** directly or indirectly. For example, if you define a new class **Exam** as follows:

```
        class Exam extends Object {

        }
```

this is the same as leaving off "extends Object". In Java's jargon, we would say **Object** is the *superclass* of **Exam** and **Exam** is the *subclass* of **Object**. By "extends" we mean that **Exam** *inherits* methods and data instances from **Object**. (See page 382 fourth (345 in fifth; 421 in sixth) edition and Chapter 9 in fourth (10 in fifth; 9 in sixth) edition of Java text for more on inheritance.) One method that is part of the **Object** class and, therefore, inherited by *all* objects is **toString**. See page 384 fourth (407 in fifth; 424 in sixth) edition and 456 fourth edition. If you use an object in a **System.out.print** method, Java automatically calls the **toString** method associated with that object. Therefore, you should override **toString** when you create your own classes.

You are to write a new Java application which allows the user to enter up to 20 student names and their exam scores. The information will be stored in an array of a user-defined class **Exam**. This second class will be placed in the file after the primary class. The primary class should have a method to read in the information and a second method to print the information. Keep the **Exam** class as small as possible, i. e., only methods that directly operate on the two data members name and score.

Your class **Exam** should override method **toString** to allow the printing of the name and score with the following:

```
        System.out.println(grades[i]);
```

You will need to make your **toString** method **public**.

In contrast to the **int** array of Exercise 1, this exercise involves an array of objects, i. e., **Exam** objects. As in C++, using the **new** method to create an array of a class of objects *only* creates an array of references. You still need to use the **new** method repeatedly to create each element of the array.

Hand in:

For Exercises 2, 3, 4, 5 and 7, combine all the Java listings and outputs from runs into one handin file with a **.java** extension. Print using the **print** alias set up in Learning Java Exercise 1.