Letterkenny Institute of Technology

Course code: OOPR CP603

YEAR 2 COMPUTING

(Common paper for all streams)

Subject: Object Oriented Programming **Stage:** 2

Date: Autumn 2015 *(Repeat Semester 1)* **Examiners:** Mr. D. Hegarty, Ms. P. Keane

Time allowed: 3 hours

INSTRUCTIONS

Answer any **FOUR** questions. All questions carry equal marks.

NOTE: It may be useful to remove the appendices from the questions portion of the paper – i.e. you can have the questions side-by-side with the relevant code/diagrams, and don't need to go back and forth.

Question 1

A basic Person class is given below. Answer the subsequent questions based on this class. Note: You can assume that all code snippets are part of a valid tester class containing a `main()` method.

```
public class Person
{
   private String name;
   private int age;

   public Person(String name, int age)
   {
      this.name = name;
      this.age = age;
   }

   public boolean equals(Object o)
   {
      Person other = (Person)o;
      return(this.name.equals(other.name) &&
            this.age == other.age);
   }
}
```

a) Explain why the **`if`** statement in the following code snippet is legal, i.e. will compile, in java:

(4 marks)

```
Person p1 = new Person(“John Boyle”, 19);
String s1 = new String(“A String”);
if (p1.equals(s1))
{
   //Do something
}
```

b) What would happen if the code snippet in part a) was actually executed? Explain your reasoning.

(6 marks)

c) Describe the issue with the following piece of code:

(3 marks)

```
Person p1 = new Person(“John Boyle”, 19);
Person p2;
if (p1.equals(p2))
{
   //Do something
}
```

d) Provide a more robust `equals()` method which will handle any of the potential problems presented by the examples above.

(6 marks)

e) Explain, using an appropriate example, why we should not simply change the parameter type of the `equals()` method to type `Person`.

(6 marks)

Question 2

You are given a Person, Student and Date class in Appendix A.

a) Write a Subject class which maintains a subjectName (String) and a score (double). Provide a constructor to initialise both instance fields, getter methods, and a toString() method.

(5 marks)

b) Provide additional code to the Student class so that it can now maintain a list of subjects. Remember to provide an appropriate mutator method (for example, you could call it `addSubject`).

(6 marks)

c) Add a method with the signature public `Subject getBestSubject(),` which will loop through the student's list of subjects and return the subject with the best score.

(7 marks)

d) Provide test code to demonstrate usage of the Student class:
   - Create a student object
   - Add two subjects to the student
   - Print out the student's details followed by information about the student's best subject.

(7 marks)

Question 3

Distinguish between the Comparable interface and the Comparator interface.
Your answer should provide appropriate examples of class code which implements each interface and examples of their usage using the Collections class.

(25 marks)

Question 4

A Shape class is given below:

```
public abstract class Shape
{
   private String color;
   private boolean filled;

   public Shape(){
      color = "red";
      filled = true;
   }

   public Shape(String color, boolean filled){
      this.color = color;
      this.filled = filled;
   }

   public String getColor(){
      return color;
   }

   public boolean isFilled(){
      return filled;
   }

  public void setColor(String c){
    color = c;
   }

   public void setFilled(boolean f){
      filled = f;
   }

   public abstract double getArea();

   public String toString(){
      return "Shape: color=" + color + " filled=" + filled;
   }
}
```

a) Within the context of polymorphism, explain the purpose of making `getArea()` abstract. Provide an appropriate example to illustrate your explanation.

(10 marks)

b) If a subclass of class Shape does not override the `getArea()` method what consequence will this have?

(3 marks)

c) For the BankAccount Code in Appendix B, provide code to implement the Comparable interface (see Appendix C for details on Comparable). Provide a small snippet of test code which will demonstrate how you subsequently sort an ArrayList of bankaccounts using `Collections.sort()`.

(12 marks)

Question 5

Given the BankAccount class in Appendix B, answer the following:

a) Create a CurrentAccount class which "IS A" BankAccount. The following are the main requirements:
   - A CurrentAccount object keeps track of the number of transactions.
   - Each deposit/withdraw is a transaction (hint: override).
   - Each transaction will cost 0.30 and a `deductFees()` method will handle the automatic updating of the account.
   - Provide appropriate constructors.

   (12 marks)

b) Add a method called `directDebit()` to your CurrentAccount class which will permit a transfer of money to any other type of BankAccount.

   (6 marks)

c) Explain how overriding and the flexible use of superclass references provide a framework for run-time polymorphism.

   (7 marks)

Question 6

(a) Given the Sorter class in Appendix D write a small tester class that initially has an array of six unique values (either randomly generated or hard-coded), and invokes the bubbleSort method to sort the array.

(5 marks)

(b) Using a simple integer-based example, explain (using pseudo-code or otherwise) how the selection sort works.

(8 marks)

(c) Explain why a sorted group of elements is an important advantage to the efficiency of a search. Your answer should include a description of the binary search algorithm which uses this advantage.

(7 marks)

(d) Explain the use of the `arraySorted` flag in the bubbleSort method.

(5 marks)

Appendix A - Example of Composition (Question 2)

```
public class Date
{
   private int day, month, year;

   public Date(int d, int m, int y)
   {
      day = d;
      month = m;
      year = y;
   }

   public String toString()
   {
      String[] months=
            {"Jan","Feb","Mar","Apr","May","June","July","Aug","Sep",
            "Oct","Nov","Dec"};
      return day + "," + months[month-1] + "," + year;
   }
}

------------------------------------

public class Person
{
   private String name;
   private Date dob;

   public Person(String n, Date d)
   {
      name = n;
      dob = d;
   }

   public String getName()
   {
      return name;
   }

   public Date getDate()
   {
      return dob;
   }
}

------------------------------------

//Student class on next page
```

```
public class Student extends Person
{
   private String studentId;

   public Student(String n, Date d, String id)
   {
      super(n,d);
      studentId = id;
   }

   public String getId()
   {
      return studentId;
   }

   public String toString()
   {
      return "Name: " + getName() + " Date: " + getDate() + " ID: " +
                  getId();
   }

}
```

Appendix B – BankAccount class (questions 4 & 5)

```
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount
{

   // declare instance variables
   private double balance;

   /**
      Constructs a bank account with a zero balance
   */
   public BankAccount()
   {
      balance = 0;
   }

   /**
      Constructs a bank account with a given balance
      @param initialBalance the initial balance
  */
   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }

   /**
      Gets the current balance of the bank account.
      @return the current balance
   */
public double getBalance()
   {
     return balance;
   }

   /**
      Deposits money into the bank account.
      @param amount the amount to deposit
   */
   public void deposit(double amount)
   {
      balance = balance + amount;
   }


   /**
      Withdraws money from the bank account.
      @param amount the amount to withdraw
  */
   public void withdraw(double amount)
   {

        balance = balance - amount;
   }

   /**
      Transfers money from the bank account to another account
      @param amount the amount to transfer
      @param other the other account
   */
   public void transfer(double amount, BankAccount other)
   {
      withdraw(amount);
      other.deposit(amount);
   }

}
```

Appendix C- Information on the Comparable interface (Question 4)

Interface Comparable<T>

Type Parameters:

T - the type of objects that this object may be compared to

Method Summary

Methods

Modifier and Type	Method and Description
`int`	`compareTo(T o)` Compares this object with the specified object for order.

This is how the Comparable interface would be written:

```
public interface Comparable<T>
{
   public int compareTo(T o);
}
```

Appendix D – Sorter class (Question 6)

```
/* Sort Utility Class*/

public class Sorter
{
  /**  Uses Selection Sort to sort
  *      an integer array in ascending order
  *    @param the array to sort
  */
  public static void selectionSort( int [] array )
  {
   int max;  // index of maximum value in subarray

    for ( int i = 0; i < array.length; i++ )
    {
      // find index of largest value in subarray
      max = indexOfLargestElement( array, array.length - i );

      //Swap the elements at the index of the largest (max)
      // and the last index in our sub-array (see notes)
      swap(array, max, array.length - i - 1);
     }
  }

  /**  Finds index of largest element
  *    @param    size  the size of the subarray
  *    @ return  the index of the largest element in the subarray
  */
  private static int indexOfLargestElement( int [] array, int
size )
  {
    int index = 0;
    for( int i = 1; i < size; i++ )
    {
       if ( array[i] > array[index] )
          index = i;
    }
    return index;
  }

  /**  Swaps 2 elements in a given array
  *    @param    array  the array on which we are to perform the
swap
  *    @param    index1  the location of the 1st element
  *    @param    index2  the location of the 2nd element
  */
  private static void swap( int[] array, int index1, int index2)
  {
    int temp = array[index1];
    array[index1] = array[index2];
    array[index2] = temp;
  }


//Continued...
/**  Performs a Bubble Sort on an integer array,
  *      stopping when array is sorted
  *    @param array to sort
  */
  public static void bubbleSort( int [] array )
  {
     boolean arraySorted = false;

     for ( int i = 0; i < array.length - 1 && !arraySorted;
                               i++ )
     {
        arraySorted = true;   // start a new iteration;
                              //  maybe the array is sorted

        for ( int j = 0; j < array.length - i - 1; j++ )
        {
           if ( array[j] > array[j + 1] )
           {
              // swap the adjacent elements
              // and set arraySorted to false
              swap(array, j+1, j);

              arraySorted = false; // note that we swapped
           }
        }
     }
  }

}
```