Letterkenny Institute of Technology

Course code: OOPR CP603

YEAR 2 COMPUTING

(Common paper for all streams)

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

Date: January 2013 **Examiners:** Mr. D. Hegarty
Ms. O. McMahon

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

Given the BankAccount class in Appendix A, 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 provides a framework for run-time polymorphism.

   (7 marks)

Question 3

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 A, provide code to implement the Comparable interface (see Appendix B 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 4

The following class is given:

```
public class Swapper
{
   static void swap(ArrayList<Integer> list, int index1, int index2)
   {
      int temp = list.get(index1);
      list.set(index1, list.get(index2));
      list.set(index2, temp);
   }

}
```

a) Provide valid test code which calls the `swap()` method and prints out the subsequent list.

(5 marks)

b) The method, as shown, mutates the ArrayList parameter. Provide a non-mutator version and, again, show how you would correctly invoke the method.

(8 marks)

c) Add a try-catch block to the non-mutator method which will handle the obvious potential for out-of-bounds errors.
Note: your catch block should simply report the error, using the exception object's built-in facilities.
Also, you can just use the general Exception class – see Appendix C - for your error handling, if you wish.

(5 marks)

d) Show the modifications required to the BankAccount class in Appendix A so that the bankaccount objects can store an automatically generated account number. Your answer should include the use of a static variable.

(7 marks)

Question 5

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

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 6

Appendix E consists of two parts: A Sorter class with selectionSort and bubbleSort methods; A Searcher class with a binarySearch method.

(a) The bubbleSort algorithm could potentially quit sorting if it realised that there were no swaps on the previous pass. Amend the method to provide this functionality (hint: use a boolean flag)

(6 marks)

(b) State what changes are required to the code to make the bubbleSort code sort in descending order.

(3 marks)

(c) For a list with 7 elements in it, determine how many comparisons would be required by the selectionSort algorithm.

(3 marks)

(d) For the Searcher class, provide a sequentialSearch method which will search for a given key in a **sorted** array (of type int). The method should return the index of the key or -1 if the key is not found.
Note: The method should quit searching when appropriate.

(6 marks)

(e) For the following array (13 elements), determine the indices that binarySearch will look at when searching for the search-key 70.

[6 9 14 21 30 45 46 51 57 67 72 88 92]

Your answer should identify what occurs on each pass in terms of the variables **`start, mid, end`**.

(7 marks)

Appendix A – BankAccount class

```
/**
   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;
   }

//Continued...
```

```
   /**
      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 B – Information on the Comparable interface

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 C – Information on the Exception class

Class Exception
java.lang.Object
  └ java.lang.Throwable
      └ **java.lang.Exception**

```
public class Exception
extends Throwable
```

The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch.

Method Summary

Methods inherited from class java.lang.Throwable

```
fillInStackTrace, getCause, getLocalizedMessage, getMessage,
getStackTrace, initCause, printStackTrace, printStackTrace,
printStackTrace, setStackTrace, toString
```

Appendix D – Example of Composition

```
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 E – Part 1 - Sorter class

```
/* 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);
     }
  }

  /**  Performs a Bubble Sort on an integer array,
  *     Note: this version does not stop once the array is sorted
  *    @param array to sort
  */
  public static void bubbleSort( int [] array )
  {
     for ( int i = 0; i < array.length – 1; i++ )
     {

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


//Continued...
```

```
  /**  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;
  }

}
```

Appendix E – part 2 - Searcher class

```
public class Searcher
{

  //This method will return the index of the searchItem in the array
  //If it doesn't find the searchItem, it will return -1 to indicate
  //this
  public static int binarySearch(int[] list, int searchItem)
  {
    int start=0;
    int end = list.length - 1;

    int mid = 0;

    boolean found = false;

    //Loop until found or end of list.
    while(start <= end && !found)
    {
        mid = (start + end) /2;

        if(list[mid] == searchItem)
        {
          found = true;
        }
        else
        {
            if(list[mid] > searchItem)
            {
               end = mid -1;
            }
            else
            {
               start = mid + 1;
            }
        }
    }

    if(found)
    {
       return mid;
    }
    else
    {
       return(-1);
    }
  }
}
```