Letterkenny Institute of Technology

Course code: OOPR CP603

YEAR 2 COMPUTING

(Common paper for all streams)

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

Date: January 2014 **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 1: You may find it useful to refer to appendix A for this question.
Note 2: 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;
   }

}
```

a) Override the toString() method.

(2 marks)

b) Amend the Person class so the code snippet below will work properly

```
ArrayList<Person> people = new ArrayList<Person>();

//Assume Person objects have been added to the list

if (people.contains(new Person("Adam Ant", 48)))
{
   //Do something
}
else
{
   //Do something else
}
```

(8 marks)

c) Assuming that I want to use the sort method of the Collections class to sort the `people` list alphabetically by name - `Collections.sort(people);` - provide the required changes to the Person class.

(6 marks)

d) If I also wish to provide the additional capability of sorting by age, provide a mechanism that will allow this additional sorting capability. Show how you would invoke the Collections.sort() method for this.

(9 marks)

Question 2

a) What is wrong with the following interface?

```
public interface SomethingIsWrong {
   void aMethod(int aValue) {
        System.out.println("Hi Mom");
    }
}
```

(3 marks)

b) Explain what is required to fix the interface in part a).

(2 marks)

c) Distinguish between an abstract class and an interface.

(5 marks)

d) Explain the consequence of a subclass not overriding a superclass's abstract method.

(3 marks)

e) Using the information in Appendix B, create a Student class which is capable of doing part-time jobs.

The Student class should have the following instance fields:
name (String), id (int), course(String), spendingMoney (double).

In addition it should have the following methods:
an overloaded constructor; a method with the signature
public `int buyNoodles (double pricePerPacket).`
This method should return the how many packets of noodles the student can buy.

Provide a snippet of test-code showing how a student object could be used.

Note: You should assume that the Job class is fully coded, i.e. you don't have to write this class.

(12 marks)

Question 3

Appendix C 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)

(5 marks)

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

(2 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.

(5 marks)

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

[2, 5, 7, 13, 15, 16, 18, 20, 22, 24, 30, 34]

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

(7 marks)

(f) How many passes are required in a binarySearch for an array/list containing 40 elements? You should explain your answer.

(3 marks)

Question 4

Given the BankAccount class in Appendix D, 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 5

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 D so that the bankaccount objects can store an automatically generated account number. Your answer should include the use of a static variable and an accessor method to return an object's account number.

(7 marks)

Question 6

Appendix E contains a Book class, a BookStoreTester class and sample output for partial title matching.

(a) Provide a BookStore class. The class should initially contain:

- An ArrayList<Book> reference called **books**.

(2 marks)

- An **addBook** method which will accept a book object as a reference. It will add the book to the ArrayList.

(3 marks)

- A method called **listAll** which uses an enhanced for loop to iterate over the ArrayList printing each book's details to System.out

(4 marks)

(b) Add a **searchByTitle** method to the BookStore class which will provide the following:

- It should have the signature :
  `public ArrayList<Book> searchByTitle(String searchStr)`

- The method should allow partial matches – hint: the String class has a `contains()` method which takes a string argument and returns true/false depending on whether the current string contains the argument string.

- The method should return all books which match the *searchStr*. If there are no matches it should return an empty list.

(10 marks)

(c) Explain briefly (you do not have to provide the actual code) the modifications required to have a "search by genre" facility.

(6 marks)

Appendix A – Information on the Comparable and Comparator interfaces (Question 1)

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);
}
```

Interface Comparator<T>

Type Parameters:

T - the type of objects that may be compared by this comparator

Method Summary

Methods	
Modifier and Type	**Method and Description**
`int`	`compare(T o1, T o2)` Compares its two arguments for order.
`boolean`	`equals(Object obj)` Indicates whether some other object is "equal to" this comparator.

Appendix B – PartTimeAble interface and Job class (Question 2)

Interface PartTimeAble

```
public interface PartTimeAble
```

The interface is designed for short-term jobs. Employees, Managers, and Students can all earn additional money via this interface. The specifics of this are left to each implementing class

Method Summary

Modifier and Type	Method and Description
`void`	`doJob(Job j)` what this method will entail differs from class to class

Method Detail

doJob

```
void doJob(Job j)
```

what this method will entail differs from class to class

Class Job

java.lang.Object
    Job

```
public class Job
extends java.lang.Object
```

Encapsulates information about a job. Note: can be used as standalone or in conjunction with the PartTimeAble interface

Field Summary

Fields

Modifier and Type	Field and Description
`private java.lang.String`	`jobDescription` A short description of the job, eg "temp office work"
`private double`	`rate` rate is hourly rate
`private double`	`time` time spent doing the job

Constructor Summary

Constructors

Constructor and Description
`Job(java.lang.String jobDescription, double rate, double time)`

Method Summary

Methods

Modifier and Type	Method and Description
`java.lang.String`	`getJobDescription()`
`double`	`getPrice()` Price is calculated as a product of rate and time
`double`	`getRate()`
`double`	`getTime()`
`java.lang.String`	`toString()`

Appendix C: Part 1 - Sorter class (Question 3)

```
/* 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 C: 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);
    }
  }
}
```

Appendix D – BankAccount class (Question 4 and Question 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;
   }

//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 E : Part 1 – Book class (Question 6)

```
public class Book
{
   private String title;
   private String author;
   private double price;

   public Book(String title, String author, double price)
   {
      this.title = title;
      this.author = author;
      this.price = price;
   }

   public String getAuthor()
   {
      return author;
   }

   public String getTitle()
   {
      return title;
   }

   public double getPrice()
   {
      return price;
   }

   public String toString()
   {

      return ("title: " + title + "\t"
            + "author: " + author + "\t"
            + "price: " + price + "\n");
   }
}
```

Appendix E : Part 2 – BookStoreTester class

```
public class BookStoreTester
{
  public static void main(String args[])
  {
    BookStore theBookShop = new BookStore();

    theBookShop.addBook(new Book("Big Java", "Horstmann", 38.99));
    theBookShop.addBook(new Book("The Corrections", "Franzen", 12.99));
    theBookShop.addBook(new Book("Five Go Camping", "Enid Blyton", 7.90));
    theBookShop.addBook(new Book("Head First Java", "Kathy Sierra", 42.50));

    ArrayList<Book> searchResult;

    searchResult = theBookShop.searchByTitle("Java");

    System.out.println(searchResult);

  }
}
```

Appendix E : Part 3 – Sample output from BookStoreTester