BIRKBECK

(University of London)

MSc EXAMINATION FOR INTERNAL STUDENTS

MSc Computer Science

Department of Computer Science and Information Systems

Programming in Java

BUCI033S7

DATE OF EXAMINATION: Friday, 7th June 2013
TIME OF EXAMINATION: 10:00–13:00
DURATION OF PAPER: 3 hours

WITH OUTLINE SOLUTIONS

RUBRIC:

1. Candidates should attempt ALL 11 questions on this paper.
2. You are advised to look through the entire examination paper before getting started, in order to plan your strategy.
3. Simplicity and clarity of expression in your answers is important.
4. All programming questions should be answered using the Java programming language unless stated to the contrary.
5. Electronic calculators are NOT allowed.
6. **Start each question on a new page**.

Question:	1	2	3	4	5	6	7	8	9	10	11	Total
Marks:	4	4	8	5	8	8	8	8	15	12	20	100

Question 1 .... Total: 4 marks

(a) What is the difference between the `==` operator and an `equals` method? Why do we generally want to use an `equals` method when comparing object type variables as opposed to the `==` operator? [2 marks]

Outline Solution:
The `==` operator just checks to see if the variable values are the same, which is fine for primitives, since their values are simply numerical quantities. For reference variables, it will be *true* if the variables have the same address, meaning they refer to the exact same object in memory. However, we can write an `equals` method that will look at field values of objects instead of their memory addresses to determine whether or not they are "equal" where we define what "equal" means as appropriate to the type of object we are defining.

(b) Why do we want to create various classes as opposed to putting everything into different methods in the one class that contains the `main` method? [2 marks]

Outline Solution:
By making separate classes we encapsulate methods and data that apply to particular object types, and can then easily reuse these types in other applications. We can control how our data and functionality are accessed and in general, partition our software into various pieces which are separate both conceptually and functionally.

Question 2 .... Total: 4 marks

Consider the following code from the definition of the class `Building`:

```
public class Building {
    public int area;
    private String name;
    protected double cost;

    public Building() {
    }

    protected boolean isBuilt() {
        // ...
        return true;
    }

    public void moveIn() {
        // ...
    }

    public double rentSpace() {
        // ...
        return 0;
    }

    private int renovate(int days) {
        // ...
        return 1;
    }

    // ...
}
```

(a) Which members are visible to (accessible from) `Building` and its subclasses? 1 mark

Outline Solution:
public and protected members:

```
fields; area, cost

constructor: Building

methods: isBuilt, moveIn, rentSpace
```

(b) Which members are visible to (accessible from) only the `Building` class? 1 mark

Outline Solution:
private members:

```
field: name
method: renovate
```

(c) Which members are visible to (accessible from) any class? 1 mark

Outline Solution:
public members:

```
field: area
constructor: Building
methods: moveIn, rentSpace
```

(d) Which field(s) should we write getter(s) and setter(s) for if we want classes apart from `Building` and its subclasses to access them? 1 mark

Outline Solution:
the private and protected ones:

```
name, cost
```

Question 3 . . . . Total: 8 marks

(a) Briefly explain the meaning of the `finally` keyword and provide one example of its use. 4 marks

Outline Solution:
With reference to *exception handling* — The finally block always executes when the try block exits. This ensures that the finally block is executed even if an uncaught exception occurs.
Any appropriate example.

(b) Briefly explain the meaning of, and provide one example of, the use of the annotation `@Before` from JUnit 4. 4 marks

Outline Solution:
When writing tests, it is common to find that several tests need similar objects created before they can run. Annotating a (public void) method with `@Before` causes that method to be run before the *Test* method.
Any appropriate example.

Question 4 . . . . Total: 5 marks

Write a program in Java that asks the user for the bottom and top values in a range of integers. The program then asks the user for a number of integers. The user signals the end of input with a 0.

The program computes two sums:

- the sum of integers that are in the range (inclusive), and
- the sum of integers that are outside of the range.

For example:

```
Bottom end of range: 20
Top end of range:50
Enter data: 21
Enter data: 60
Enter data: 49
Enter data: 30
Enter data: 91
Enter data: 0

Sum of in range values: 100
Sum of out of range values: 151
```

Note: Your program does not need to validate the values input.

Outline Solution:

```
import java.util.Scanner;

public class Addr {
    public static void main(String[] args) {
        int lower;
        int upper;
        int sumIn = 0;
        int sumOut = 0;
        int value;

        Scanner sc = new Scanner(System.in);
        System.out.print("Bottom end of range: ");
        lower = sc.nextInt();
        System.out.println("Top end of range: ");
        upper = sc.nextInt();
        do {
            System.out.print("Enter data: ");
            value = sc.nextInt();
            if (value < lower || value > upper) sumOut += value;
            else sumIn += value;
        } while (value != 0);

        System.out.println("Sum of in range values: " + sumIn);
        System.out.println("Sum of out of range values: " + sumOut);
    }
}
```

5 marks

Question 5 .... Total: 8 marks

Write a recursive method that takes a string and produces the following output (this is for the string "abcde").

```
abcde
abcd
abc
ab
a
a
ab
abc
abcd
abcde
```

Outline Solution:

Just expecting the method...

```
public class RecursiveString {
    public static void main(String[] args) {
        stringRecurse("abcde");
    }

    public static void stringRecurse(String s){
        System.out.println(s);
        if (s.length() > 1) stringRecurse(s.substring(0, s.length() - 1));
        System.out.println(s);
    }
}
```

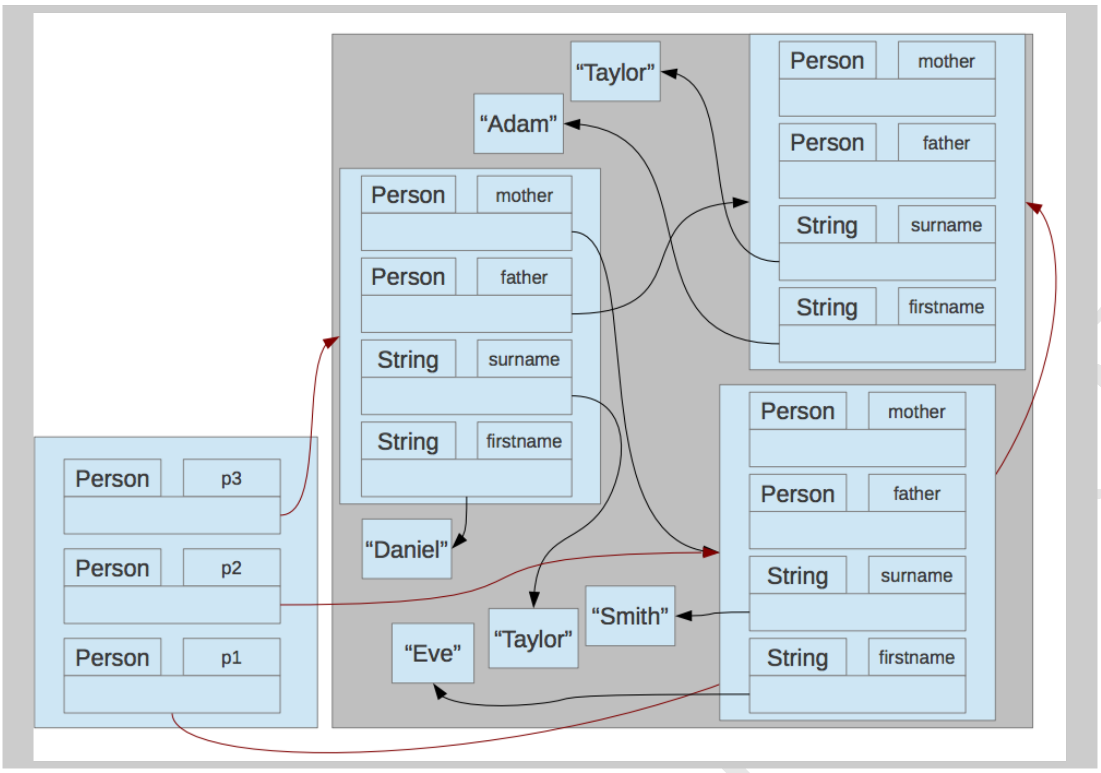
First print — 1
substring — 3
Second print — 1
recurse — 3

Question 6 .... Total: 8 marks

Using the same symbols as used in the materials presented in class, draw a diagram that represents the state of the computer's memory after the following Groovy script is executed:

```
class Person {
    String firstname;
    String surname;
    Person father;
    Person mother;

    String personDetails() {
        String result = "";
        result += "firstname: " + firstname;
        result += "\n";
        result += "surname: " + surname;
        return result;
    }

    String familyDetails() {
        String result = "";
        result += "father\n";
        result += "------\n";
        result += father.personDetails();
        result += "\n";
        result += "mother\n";
        result += "------\n";
        result += mother.personDetails();
        return result;
    }

    void showDetails() {
        println personDetails();
        println familyDetails();
    }
}

Person p1 = new Person()
p1.firstname = "Adam"
p1.surname = "Taylor"

Person p2 = new Person()
p2.firstname = "Eve"
p2.surname = "Smith"

Person p3 = new Person()
p3.firstname = "Daniel"
p3.surname = "Taylor"
p3.father = p1
p3.mother = p2
p3.showDetails()
```

Outline Solution:



Question 7 . . . . Total: 8 marks

Consider the following two classes:

```
package sample;

public class Base {
    public void methodOne() {
        System.out.println("A");
        methodTwo();
    }

    public void methodOne(int a) {
        System.out.println("W");
        methodTwo();
    }

    public void methodTwo() {
        System.out.print("B");
    }
}
```

```
package sample;

public class Derived extends Base {
    public void methodOne(int a) {
        super.methodOne();
        System.out.print("X");
    }

    public void methodOne(Integer a) {
        super.methodOne();
        System.out.print("C");
    }

    public void methodTwo() {
        super.methodTwo();
        System.out.print("D");
    }
}
```

The following declaration appears in a client program.

```
    Base b = new Derived();
```

What is printed as a result of the call `b.methodOne(0)`?

Outline Solution:

2 marks for getting A
2 marks for getting B
1 mark for X
3 marks for D

```
A
BDX
```

Question 8 .... Total: 8 marks

The following code does not compile:

```
110 public void printFile(Path path) {
111   BufferedReader reader = Files.newBufferedReader(path, Charset.forName("UTF-8"));
112   String line = null;
113   while ((line = reader.readLine()) != null) {
114     System.out.println(line);
115   }
116 }
```

The compilation error is

```
Line 113: Unreported exception java.io.FileNotFoundException; must be
caught or declared to be thrown.
```

Modify the method so that the code compiles noting that line 110 cannot be modified because it implements an interface that cannot be changed.

Outline Solution:

Only the `printFile` method need be reproduced...

```
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;

public class PrintFile {
    public void printFile(Path path) {
        try
                (BufferedReader reader =
                         Files.newBufferedReader(path, Charset.forName("UTF-8"));) {
            String line = null;
            try {
                while ((line = reader.readLine()) != null) {
                    System.out.println(line);
                }
            } catch (IOException ex) {
                System.err.println("Error reading line from file");
            }
        } catch (IOException ex) {
            System.err.println("Error with opening the file");
        }

    }
}
```

and can contain a single `try...catch`, rather than a nested one.

Question 9 . . . . Total: 15 marks

In statistics when we need to measure something we use a *variable*. This *variable* is like a variable in computer programming. It can be thought of as a box that hosts variable values. Every statistical measurement results in a series of values for a given variable. This is called a *sample*. The *frequency distribution* of a variable tells us what values a variable takes in the sample and how often (e.g., there are thirteen boys of age 10, fifteen of age 11, and one of age 12 in this class).

(a) Let's suppose that we have a numeric variable that takes values in the range 0-99. The size of the sample is 30. Compute the frequency distribution of these values in the range 0-99. Implement both the sample as an array and the frequency distribution as a `map<Integer, Integer>`. 4 marks

Outline Solution:
see solution to part (c)

(b) One way to summarise statistical data is to compute frequency distributions using class intervals. In this case we divide the range of values into classes and we count the number of occurrences for each class. Compute the frequency distribution of the above sample using 10 classes (0-9). The first class corresponds to values 0-9, the second to values 10-19 and so on. Implement both the sample and the frequency distribution as in (a) 6 marks

Outline Solution:
see solution to part (c)

(c) The frequency distribution values can then be used to identify certain patterns in the data. One of the things that statisticians may be interested in is the symmetry (or asymmetry) of the distribution. A distribution is symmetrical when the left 5 marks half of the values look as a mirror image of the right half. In this case to keep things simple we will consider a distribution as symmetrical if the the sum of the values in one half is not greater than twice the sum of the values in the other half. Write a method that takes the frequency distribution of part (b) as input and returns a boolean value indicating whether the distribution is symmetrical or not.

Outline Solution:

```
import java.util.HashMap;
import java.util.Map;

public class FreqDist {
    public static void main(String[] args) {
        int[] sample = //new int[30]; // sample size
                {0, 23, 4, 5, 5, 6, 34, 56, 99, 45, 87};
        // values are stored in sample
        // values have the range 0..99

        Map<Integer, Integer> freqDist = new HashMap<>();

        // compute distribution for (a)

        for (int i : sample) {
            int temp = 0;
            if (freqDist.containsKey(i))
                temp = freqDist.get(i);
            freqDist.put(i, temp + 1);
        }

        for (int i : freqDist.keySet())
            System.out.println("Number " + i + " has frequency " +
                    freqDist.get(i));
        System.out.println("=======");

        // compute distribution for (b)

        freqDist = new HashMap<>();
        for (int i : sample) {
            int temp = 0;
            int slot = i / 10;
            if (freqDist.containsKey(slot))
                temp = freqDist.get(slot);
            freqDist.put(slot, temp + 1);
        }

        for (int i : freqDist.keySet())
            System.out.println("Number range " + i + " has frequency " +
                    freqDist.get(i));
        System.out.println("=======");

        // compute distribution for (c)

        int sumLower = sum(0, 5, freqDist);
        int sumUpper = sum(5, 10, freqDist);

        System.out.println(sumLower <= sumUpper * 2 || sumUpper <= sumLower * 2);
    }

    private static int sum(int lower, int upper, Map<Integer, Integer> freqDist) {
        int sum = 0;
        for (int i = lower; i < upper; i++) {
            if (freqDist.containsKey(i))
                sum += freqDist.get(i);
        }
        return sum;
    }
}
```

Question 10 .... Total: 12 marks

You are responsible of the data structures for a simple drawing application in 2D. Create classes for circles, ellipses, rectangles, squares, and points. Take into account that:

- All shapes must have a central point that defines where they are on the plane.
- Both rectangles and squares must have a method `contains(Point p)` that returns `true` if the point is inside them and `false` otherwise.
- All classes have getters for observing their main attributes (circle: radius; ellipse: major axis, minor axis; square: side; rectangle: width, breadth; all: centre), but they have no setters.
- You must try to minimise the total number of constructors.
- You must not repeat code.

Outline Solution:

All classes should probably be `public` but the subclasses were incorporated into the same source file for ease of presentation.

```
package shapes;

public abstract class Shape {
    private Point centre;

    public Shape(Point centre) {
        this.centre = centre;
    }

    public Point getCentre() {
        return centre;
    }
}
```

```
package shapes;

public class Point {
    private double x;
    private double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }
}
```

```
package shapes;

public class Ellipse extends Shape {
    private double majorAxis;
    private double minorAxis;

    public Ellipse(Point p, double majorAxis, double minorAxis) {
        super(p);
        this.majorAxis = majorAxis;
        this.minorAxis = minorAxis;
    }

    public double getMajorAxis() {
        return majorAxis;
    }

    public double getMinorAxis() {
        return minorAxis;
    }
}
```

```
package shapes;

public class Circle extends Ellipse {

    public Circle(Point p, double radius) {
        super(p, radius, radius);
    }

    public double getRadius() {
        return getMajorAxis();
    }
}
```

```
package shapes;

public class Rectangle extends Shape {
    private double width;
    private double breadth;

    public Rectangle(Point p, double width, double breadth) {
        super(p);
        this.width = width;
        this.breadth = breadth;
    }

    public boolean contains(Point p) {
        // compute bounds
        Point centre = getCentre();
        double lowerx = centre.getX() - (getWidth() / 2);
        double lowery = centre.getY() - (getBreadth() / 2);
        double upperx = centre.getX() + (getWidth() / 2);
        double uppery = centre.getY() + (getBreadth() / 2);
        return lowerx < p.getX() && lowery < p.getY()
                && p.getX() < upperx && p.getY() < uppery;
    }

    public double getWidth() {
        return width;
    }

    public double getBreadth() {
        return breadth;
    }
}

package shapes;

public class Square extends Rectangle {
    public Square(Point p, double side) {
        super(p, side, side);
    }

    public double getSide() {
        return getWidth();
    }
}
```

use of `Point` — 2
structure (inheritance) — 2
general coding — 3
use of `super`, etc. — 2
`contains` method — 3

Question 11 .... Total: 20 marks

Consider the class `ListNodeImpl`:

```
public class ListNodeImpl<T> implements ListNode<T> { ... }
```

which implements a linked list of objects using the following interface:

```
public interface ListNode<T> {
    public T getItem();

    public void setItem(T item);

    public ListNode<T> getNext();

    public void setNext(ListNode<T> l);
}
```

(a) Write the method bodies for `ListNode` and add an appropriate constructor. 3 marks

Outline Solution:

```
public class ListNodeImpl<T> implements ListNode<T> {
    private T item; // An item in the list.

    private ListNode<T> next; // Pointer to next item in the list.

    public ListNodeImpl(T item, ListNode next) {
        this.setItem(item);
        this.setNext(next);
    }

    @Override
    public T getItem() {
        return item;
    }

    @Override
    public void setItem(T item) {
        this.item = item;
    }

    @Override
    public ListNode<T> getNext() {
        return next;
    }

    @Override
    public void setNext(ListNode<T> l) {
        next = l;
    }


}
```

0.5 for get/set plus 1 for the constructor

(b) Write a method `listSum` with the following signature 6 marks

```
static int listSum(ListNode<Integer> head)
```

which returns the sum of all the integers in a linked list, the first node of which is referenced by `head`.

Outline Solution:

```
public class ListNodeDriver {
    static int listSum(ListNode<Integer> head) {
        int total = 0; // The sum of all the items.

        while (head != null) {
            total += head.getItem(); // Add in the item in this node.
            head = head.getNext(); // Advance to the next node.
        }
        return total;
    }
}
```

(c) One important property of the stack data structure is that the first element pushed onto the stack is the last element to be popped off the stack. In other words, if you push a series of objects onto a stack in some order, the objects will be popped off in the reverse order. 11 marks

Using this property of a stack, and the `ListNode` and `ListNodeImpl` classes earlier, write a method that reverses the order of the elements in a linked-list. Your method should have the following signature:

```
public static <T> ListNode<T> reverse(ListNode<T> head)
```

where `head` references the first node of a linked list and `T` is the type of the value stored in the node. This method returns a reference to a linked-list whose elements are in the exact reverse order of the original list.

[**Note**: It is okay if the original list is destroyed during the execution of the method.]

You can assume that the class `StackImpl` which implements the following (generic) stack interface:

```
interface Stack<T> {
    public T top();

    public T pop();

    public T push(T c);

    public boolean isEmpty();
}
```

is provided for you.

Outline Solution:

This can be solved in a variety of ways, by using new nodes, by resetting the value fields, resetting the links, etc. All solutions are equally acceptable.

```
public class StackExample {

    public static <T> ListNode<T> reverse(ListNode<T> head) {
        Stack<ListNode<T>> stack = new StackImpl<>();
        ListNode<T> returnCell;

        while (head != null) {
            stack.push(head);
            head = head.getNext();
        }

        if (!stack.isEmpty()) {
            head = stack.pop();
        }

        returnCell = head;

        while (!stack.isEmpty()) {
            head.setNext(stack.pop());
            head = head.getNext();
        }

        head.setNext(null);

        return returnCell;
    }

    static void print(String s, ListNode l) {
        System.out.println("The " + s + " list");
        while (l != null) {
            System.out.println(l.getItem());
            l = (ListNode) l.getNext();
        }
    }

    public static void main(String[] args) {
        ListNode<Integer> one = new ListNodeImpl<>(1, null);
        ListNode<Integer> two = new ListNodeImpl<>(2, null);
        ListNode<Integer> three = new ListNodeImpl<>(3, null);

        ListNode head;

        one.setNext(two);
        two.setNext(three);

        head = one;
        // show the content of the original list
        print("original", head);
        head = StackExample.reverse(head);
        print("reversed", head);
    }
}
```