Birkbeck

(University of London)

BSc EXAMINATION

Department of Computer Science and Information Systems

Software and Programming II (COIY026H6)

Credit value: 15 credits

Date of examination: Wednesday, 7th June 2017
Duration of paper: 10:00 – 13:00 (3 HOURS)

1. Candidates should attempt ALL questions in the paper.
2. There are **8** questions on this paper.
3. The number of marks varies from question to question.

Question:	1	2	3	4	5	6	7	8	Total
Marks:	11	7	16	7	26	6	16	11	100

4. You are advised to look through the entire examination paper in order to plan your strategy before getting started.
5. Simplicity and clarity of expression in your answers are important.
6. Electronic calculators and supplementary material like notes and text-books are NOT permitted.
7. Answer questions using the Java programming language unless stated otherwise. You may add auxiliary methods unless stated otherwise.
8. Start each question on a new page.
9. An extract of the Java API is given at the end of this exam paper.

1. Consider the following Java classes. (**11** marks)

```
public class Fruit {
    public int k = 0;
    protected static int s = 5;

    public Fruit() {  this(10);  }
    public Fruit(int n) {  k = n;  s++;  }

    public void b(Object a, String b) {  k = 15;  }
    public void c(Object a) {  k = 20;  }
    public void c(String a) {  k = 25;  }
    public void d(String a) {  k = 30;  }
    public void d(int k) {  k = 35;  }
}

public class Orange extends Fruit {

    public Orange(int n) {  }

    public void b(Object a, String b) {  k = 45;  }
    public void b(String a, String b) {  k = 50;  }
    public void c(Object a) {  k = 55;  }
    public void c(String a) {  super.c(60);  }
    public void c(String a, Object b) {  k = 65;  }
    public void d(Object a) {  k = 70;  }

    public static void main(String[] args) {
        Fruit a = new Fruit();
        System.out.println("1 : " + a.k);
        a.d("Lime");
        System.out.println("2 : " + a.k);
        a.d(80);
        System.out.println("3 : " + a.k);
        System.out.println("4 : " + a.s);

        Orange b = new Orange(85);
        System.out.println("5 : " + b.k);
        b.c("Raspberry");
        System.out.println("6 : " + b.k);
        b.b("Banana", "Cherry");
        System.out.println("7 : " + b.k);
        System.out.println("8 : " + b.s);
```



```
        Fruit c = b;
        c.c("Apple");
        System.out.println("9 : " + c.k);
        c.b("Pear", "Lemon");
        System.out.println("10 : " + c.k);
        c.d("Pineapple");
        System.out.println("11 : " + b.k);
    }
}
```

Please write down the output that is produced when `java Orange` is invoked on the command line after the classes have been compiled.

---

2. (**Total: 4 + 3 = 7** marks)

(a) When is it better to use an array, and when is it better to use an ArrayList in a Java program? For both cases, give a suitable example and explain why the chosen approach is more appropriate for the example. [4 marks]

(b) What is meant by the term *generic type* in the context of Java? Provide an appropriate example to illustrate your answer. [3 marks]

---

3. Consider the following Java interface. (**Total: 8 + 8 = 16** marks)

```
public interface Combiner<A, B, C> {
    C combine(A x, B y);
}
```

(a) Write a class `StringMultiplier` that implements the above interface. In particular, the class is supposed to have a method [8 marks]

```
public String combine(String word, Integer n)
```

that behaves as follows:

- If `word` or `n` are the `null` reference, or if `n` is a negative number, an `IllegalArgumentException` is thrown.
- Otherwise the method returns a String obtained from appending `n` copies of `word` (where we get the empty String if `n` is 0).

For example, after we execute the line

```
String s = new StringMultiplier().combine("bc", 3);
```

the variable `s` will reference `"bcbcbc"`.

(b) Write a `default` method `crossCombine` in the interface `Combiner` that takes an `ArrayList<A> xs` and an `ArrayList<B> ys` as inputs and returns a new `ArrayList<C>`. 8 marks

If the first input list `xs` is $[x_1,\ldots,x_k]$ and the second input list `ys` is $[y_1,\ldots,y_n]$, your method is supposed to return a new `ArrayList<C>` with the following contents:

$[combine(x_1,y_1),\ldots,combine(x_1,y_n),\ \ldots,\ combine(x_k,y_1),\ldots,combine(x_k,y_n)]$

For example, the code fragment

```
ArrayList<String> words = new ArrayList<>();
words.add("a");
words.add("bc");
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(3);
numbers.add(1);
numbers.add(2);
StringMultiplier mul = new StringMultiplier();
ArrayList<String> result = mul.crossCombine(words, numbers);
System.out.println(result);
```

prints the following output on the screen:

```
[aaa, a, aa, bcbcbc, bc, bcbc]
```

In your solution you may assume that the parameters given to your method are never `null`. Your method is not supposed to modify the structure of the lists `xs` and `ys`. If one of the calls to `combine` throws an exception, your method should not catch it.

---

4. Write a `static` Java method that does the following: (**7** marks)

(1) The method first asks the user on the command line to enter an integer number.

(2) It then reads the input from the keyboard.

(3) If the input was not an `int` value, the method prints on the command line how many `int` values the user has entered and what the sum of these values is. Then the method returns.

(4) Otherwise the method continues from step (1).

---

5. (**Total: 7 + 12 + 7 = 26** marks)

Consider the following inheritance hierarchy for arithmetic expressions. An arithmetic expression is either a constant, a variable, or made up of two expressions combined by the plus operator (for simplicity we do not consider any other expressions here).

```
import java.util.ArrayList;

public interface Exp {
    public Exp replace(Var var, Exp val);
    public ArrayList<Var> getVars();

    // ... more code ...
}
```

```
public class Constant implements Exp {
    private final int value;

    public Constant(int v) { value = v; }
    // ... more code ...
}
```

```
public class Var implements Exp {
    private final String name;

    public Var(String n) { name = n; }
    // ... more code ...
}
```

```
public class PlusExp implements Exp {
    private final Exp left;
    private final Exp right;

    public PlusExp(Exp l, Exp r) {
        left = l;
        right = r;
    }
    // ... more code ...
}
```


Thus, for example the expression $(2 + (3 + x))$ would be represented as an object `e` of type `Exp` as follows:

```
Var x = new Var("x");
Exp e = new PlusExp(new Constant(2), new PlusExp(new Constant(3), x));
```

Throughout this question, you may assume that none of the method parameters and none of the instance variables of the given objects are `null`.

In this question, you will need to write code for three different classes. For each method that you write, please state clearly in which class or interface it is. In your solutions, you may add arbitrary methods to the given classes and interfaces.

(a) Implement the method `public Exp replace(Var var, Exp val)`, which is required by the interface `Exp`, for all three classes `Constant`, `Var`, and `PlusExp`. The method is supposed to return an expression obtained by replacing all occurrences of `var` in the current expression by the expression `val`. The current expression is not supposed to be modified. 7 marks

For example, after we run

```
Exp f = e.replace(x, new Constant(7));
```

the variable `f` would reference an object that represents the expression (2+(3+7)), and `e` would still reference an object that represents the expression $(2 + (3 + x))$.

(b) Implement the method `public ArrayList<Var> getVars()`, which is required by the interface `Exp`, for all three classes `Constant`, `Var`, and `PlusExp`. The method is supposed to return an ArrayList that contains exactly the variables from the expression on which the method is called. The order of the variables in the ArrayList does not matter, but each variable may occur at most once in the result list. 12 marks

For our example expression `e`, the call `e.getVars()` would return an ArrayList of length 1 that contains just `x`.

(c) Now consider the interface `Assignment`, which describes a mapping from Strings to integer values. 7 marks

```
public interface Assignment {
    int getValue(String var);
}
```

With the help of an `Assignment`, we can evaluate expressions. To this end, we add a new method

```
public int evaluate(Assignment a);
```

to the interface `Exp`. The method is supposed to evaluate the current expression using the assignment to get the values for the variables based on their name. Implement this method for all three classes `Constant`, `Var`, and `PlusExp`.

As an example, consider the following simple implementation of the `Assignment` interface, which just assigns 0 to all names.

```
public class ZeroAssignment implements Assignment {

    @Override
    public int getValue(String name) {
        return 0;
    }
}
```

Then writing

```
int n = e.evaluate(new ZeroAssignment());
```

will assign the value 5 to the variable `n` (because $(2 + (3 + 0)) = 5$).

---

6. Consider the following Java method. (**6** marks)

```
/**
 * Computes n*(n+1)/2 for non-negative n.
 * Throws an IllegalArgumentException for negative n.
 *
 * @param n an int value for which we want to compute n*(n+1)/2,
 *  should be greater than or equal to 0
 * @return n*(n+1)/2
 * @throws IllegalArgumentException if n is less than 0
 */
public static int gauss(int n) {
    if (n < 0) {
        throw new IllegalArgumentException();
    }
    int result = 0;
    for (int i = 0; i <= n; i++) {
        result = result + i;
    }
    return result;
}
```

Write JUnit test cases that together achieve *test coverage* for this method (i.e., for each statement in the method `gauss` there should be at least one test case that uses the statement). Ensure that a test fails if running the test takes more than 500 milliseconds.

Hint: Remember to use suitable Java annotations to label methods as JUnit test cases.

---

7. (**Total: 8 + 8 = 16** marks)

In this question we want to design a data structure for managing messages. An initial analysis has revealed the following properties.

- Every message is either important or not.
- A fax message has a certain number of pages.
- An e-mail has zero or more files as attachments and a timestamp.
- A file has a name and a size.
- A postal message is a message sent via postal services. Every postal message has a recipient address.
- A picture postcard is a postal message showing a certain number of sights.
- A letter is a postal message that has a certain weight in grammes.
- A registered mail message is a special letter for which it is known whether a return receipt is required.
- All postal messages and all fax messages are transported only for a certain fee. For this reason, they provide a method to calculate and return the fee for the message.

(a) Devise a suitable domain model for the messages listed above. Common features should be represented in (possibly abstract) superclasses. Introduce interfaces where appropriate. Use the UML class diagram notation from the lecture to draw your design. 8 marks

In particular, mark interfaces and abstract classes as such. For each *class*, you only need to mention the data type and the name of the attributes. You do not need to mention any methods in your classes. For each *interface*, only mention its name and for each of its methods the name, input types, and output type. For an aggregation between classes, also give the corresponding multiplicity.

(b) Implement a Java method `totalCost`. The method takes an array of messages as parameter. The method returns the sum of the fees of all messages that have a method for querying their fee. 8 marks

If the `null` array reference is passed to the method, your implementation is supposed to throw an `IllegalArgumentException` to inform the user of your method that they have passed an illegal parameter to your method. Your method should be able to handle the case that some of the entries of the array may be the `null` reference.

Use the keyword `static` for the method if and only if this is appropriate.

8. Consider the following Java class. (**11** marks)

```
public class CashRegister {

    private int itemCount;
    private double totalPrice;

    public CashRegister() {
        itemCount = 0;
        totalPrice = 0;
    }

    public void addItem(double price) {
        itemCount++;
        totalPrice += price;
    }

    public void clear() {
        itemCount = 0;
        totalPrice = 0;
    }

    public double getTotal() {
        return totalPrice;
    }

    public int getCount() {
        return itemCount;
    }
}
```

This class models a simple cash register. It has the functionality to add an item with a particular price to the current sale, to query the total amount of the current sale, to query the number of items in the current sale, and to clear the current sale.

For example, the code fragment

```
CashRegister reg1 = new CashRegister();
reg1.addItem(2.95);
reg1.addItem(1.99);
System.out.println(reg1.getCount() + " items, total " + reg1.getTotal());
reg1.clear();
System.out.println(reg1.getCount() + " items, total " + reg1.getTotal());
```

prints the following output on the screen:

```
2 items, total 4.94
0 items, total 0.0
```

We want to change the internal representation of the data in the class to track more information. To this end, we remove the instance variables `itemCount` and `totalPrice`, and we replace them by an instance variable `items` with the type `ArrayList<Double>` that keeps track of each entered item via its price.

So we now have the following modified class `CashRegisterMk2`.

```
import java.util.ArrayList;

public class CashRegisterMk2 {

    private ArrayList<Double> items;

    // ... more code ...
}
```

We now need to adapt the constructor and the existing four methods to work with the new instance variable. That is, instances of the class `CashRegisterMk2` are supposed to have the same behaviour as instances of the class `CashRegister`.

Reimplement the constructor and the four methods of `CashRegister` in the class `CashRegisterMk2` so that they work correctly using only the instance variable `items`. Do not introduce any other instance variables or class variables. You may however introduce additional methods.

Extracts from the Java API (incomplete)

All the following constructors and methods are `public`.

`java.util.ArrayList<E>`

`ArrayList()` Constructs an empty list with an initial capacity of ten.

`ArrayList(Collection<? extends E> c)` Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

`boolean add(E e)` Appends the specified element to the end of this list.

`public boolean contains(Object o)` Returns true if this list contains the specified element.

`E get(int index)` Returns the element at the specified position in this list.

`E remove(int index)` Removes the element at the specified position in this list. Returns the element that was removed from the list.

`int size()` Returns the number of elements in this list.

`java.util.Scanner`

`Scanner(InputStream source)` Constructs a new `Scanner` that produces values scanned from the specified input stream.

`boolean hasNext()` Returns true if this scanner has another token in its input.

`boolean hasNextInt()` Returns true if the next token in this scanner's input can be interpreted as an int value in the default radix using the `nextInt()` method.

`String next()` Finds and returns the next complete token from this scanner.

`int nextInt()` Scans the next token of the input as an `int`.

An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where radix is the default radix of this scanner.

`int nextInt(int radix)` Scans the next token of the input as an `int`. This method will throw `InputMismatchException` if the next token cannot be translated into a valid int value as described below. If the translation is successful, the scanner advances past the input that matched.

