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: Tuesday, 22nd May 2018
Duration of paper: 10:00 – 13:00 (3 HOURS)

1. Candidates should attempt ALL questions in the paper.
2. There are **10** questions on this paper.
3. The number of marks varies from question to question.

Question:	1	2	3	4	5	6	7	8	9	10	Total
Marks:	12	8	12	10	8	8	20	10	6	6	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 class. (**Total: 6 + 6 = 12** marks)

```
public class Person {
    private String name;

    // constructors, methods ...
}
```

(a) Override the method 6 marks

```
public boolean equals(Object obj)
```

from `Object` in `Person`. Two objects of class `Person` are equal if the instance variables `name` have equal values. Your implementation should also return with a result if `obj` or the instance variable `name` are the `null` reference.

(b) Now consider the following class. 6 marks

```
public class Employee extends Person {
    private int salary;

    // constructors, methods ...
}
```

Override the method

```
public boolean equals(Object obj)
```

from `Person` in `Employee`. Two objects of class `Employee` are equal if the instance variables `name` and `salary` have equal values. Your implementation should also return with a result if `obj` or the instance variable `name` are the `null` reference.

---

2. (**8** marks)

In addition to the default visibility (also called "package"), Java provides three further *access modifiers* for the instance variables of a class to express different visibility levels.

For each of the four visibilities, give an example for an instance variable declaration with that visibility and explain in which parts of a Java project that instance variable can be accessed.

---

3. (**Total: 4 + 8 = 12** marks)

The *run-length encoding* is a technique to compact a list of objects. The idea is to represent a list of objects as a list of pairs, where each pair stores both an object and the number of repeated occurrences of equal objects in the list. Recall that two objects `x` and `y` are equal if `x.equals(y)` evaluates to `true`. So if we have a list

```
["a", "a", "a", "b", "a", "a"]
```

that we want to represent, the run-length encoding would be:

```
[("a",3), ("b",1), ("a",2)]
```

To represent pairs in Java, we use the following generic class `Pair`.

```
public class Pair<A,B> {
    private A a;
    private B b;
    public Pair(A a, B b) {
        this.a = a;
        this.b = b;
    }
    public A getFirst() {   return this.a;   }
    public B getSecond() {   return this.b;   }

    public String toString() {
        return "(" + this.a + "," + this.b + ")";
    }
}
```

(a) Implement a generic method 4 marks

```
public static <E>
        ArrayList<E> decode(ArrayList<Pair<E,Integer>> runLength)
```

Here `runLength` is the run-length encoding of a list of objects of a generic type `E`. The method is supposed to return the original list represented by `runLength`.

For example, running the code fragment

```
ArrayList<Pair<String,Integer>> pairs = new ArrayList<>();
pairs.add(new Pair<>("a",3));
pairs.add(new Pair<>("b",1));
pairs.add(new Pair<>("a",2));
System.out.println(decode(pairs));
```

should lead to the following output:

```
[a, a, a, b, a, a]
```

In your method you may assume that `runLength` neither is nor contains `null` (directly or indirectly). Moreover, you may assume that the second component of each pair in `runLength` is a number n with $n \geq 1$.

(b) Implement a generic method [8 marks]

```
public static <E>
        ArrayList<Pair<E,Integer>> encode(ArrayList<E> data)
```

Here `data` is a list of objects of a generic type `E`. The method is supposed to return a new ArrayList with a run-length encoding of `data`. The produced list is supposed to be as short as possible.

For example, running the code fragment

```
ArrayList<String> strings = new ArrayList<>();
strings.add("a");
strings.add("a");
strings.add("a");
strings.add("b");
strings.add("a");
strings.add("a");
System.out.println(encode(strings));
```

should lead to the following output:

```
[(a,3), (b,1), (a,2)]
```

In your method you may assume that `data` neither is nor contains `null` (directly or indirectly).

---

4. Consider the following Java interface. (**Total: 5 + 5 = 10** marks)

```
public interface Converter<A, B> {
    B convert(A xs);
}
```

(a) Write a `default` method `convertAll` in the interface `Converter` that takes `ArrayList<A> xs` as input and returns a new `ArrayList<B>`. If the ArrayList `xs` is `[`x_1`, ..., `x_n`]`, the method returns `[convert(`x_1`), ..., convert(`x_n`)]`. In your method you may assume that `xs` is never `null`. If one of the calls to `convert` throws an exception, your method should not catch it. [5 marks]

(b) Write a class `StringConverter` that implements the above interface suitably. In particular, the class is supposed to have a method [5 marks]

```
public Integer convert(String word)
```

that behaves as follows:

- If `word` is the `null` reference, an `IllegalArgumentException` is thrown.
- Otherwise the method returns the *length* of `word`.

5. Write a `static` Java method that does the following: (**8** 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.

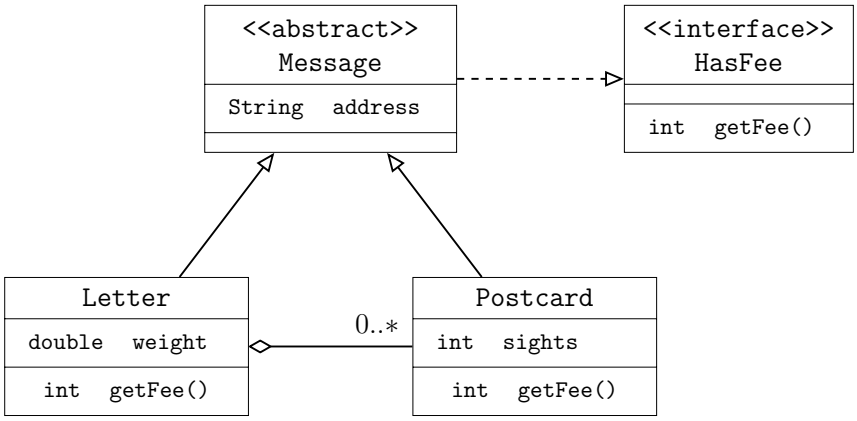
(3) If the input was not an `int` value, the method returns `-1`.

(4) If the input was `0`, the method returns how many `int` values that were strictly greater than 0 the user has entered.

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

---

6. Consider the following UML class diagram. (**8** marks)



Write Java classes that correspond to the above classes and interfaces as well as the relations between them. Instead of method bodies in the classes, just write `{ ... }`.

- For the access modifiers of the instance variables and the methods, **use the principles of information hiding**.
- You do not need to provide any constructors.
- You do not need to add code for instance variables or methods that are not in the diagram.

---

7. (**Total: 3 + 10 + 7 = 20** marks)

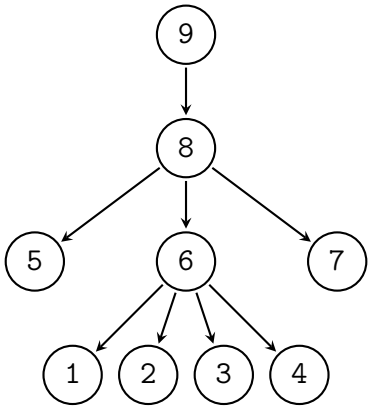
Consider the following Java class `Tree`, which implements a data structure for trees where each node can have an *arbitrary* number of children.

```
import java.util.ArrayList;

public class Tree {
    private int data;
    private ArrayList<Tree> children;

    public Tree(int data) {
        this.data = data;
        this.children = new ArrayList<>();
    }
}
```

So every object of type `Tree` has a `data` field with a number, and a reference to a *list* of its `children` from left to right. In particular, leaves of the tree have an empty list of `children` (the leaves are the nodes without any children). A graphical depiction of such a tree `t` is given below.



Throughout this question, you may assume that all method parameters and all instance variables of the given objects are not `null`.

(a) Implement a constructor 3 marks

```
public Tree(int data, Tree[] subtrees)
```

for the class `Tree` that constructs a new `Tree` object with the trees in `subtrees` as the entries of the instance variable `children`. The first statement in your implementation should be a call to the constructor `public Tree(int data)`.

(b) Implement a method 10 marks

```
public ArrayList<Integer> getLeafValues()
```

in the class `Tree`. The method is supposed to return a new list with the integers stored in the leaves of the tree in left-to-right order.

For our example `Tree t`, the method would return the following list.

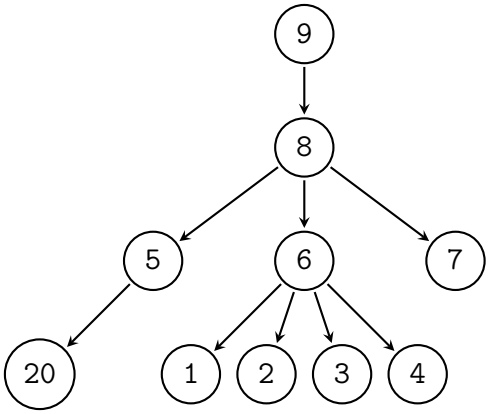
```
[5, 1, 2, 3, 4, 7]
```

(c) Implement a method 7 marks

```
public void addLeftmostLeaf(int value)
```

in the class `Tree`. The method is supposed to modify the present `Tree` by adding a new leaf to the current leftmost leaf of the tree.

For our example `Tree t`, the call `t.addLeftmostLeaf(20)` would modify the data structure to the following `Tree`.



Hint: Recall that you may introduce auxiliary methods.

---

8. Consider the following Java classes. (**10** marks)

```
public class Pet {
    protected int x = 15;
    public static int y = 2;

    public Pet() { this(20); }
    public Pet(int z) { x = 25; y++; }

    private void f() { x = 30; }
    public void g(int x) { x = 35; }
    public void g(Object o) { x = 40; }

    public static void main(String[] args) {
        Pet p1 = new Pet();
        System.out.println("1 : " + y);
        p1.f();
        System.out.println("2 : " + p1.x);
        Pet p2 = new Rabbit();
        System.out.println("3 : " + y);
        p2.f();
        System.out.println("4 : " + p2.x);
        p2.g("");
        System.out.println("5 : " + p2.x);
        try {
            Rabbit r = (Rabbit) p2;
            System.out.println("6 : " + y);
            r.f();
            System.out.println("7 : " + r.x);
            r.g("");
            System.out.println("8 : " + r.x);
            r.h();
            System.out.println("9 : " + r.x);
            r.g(new Object());
            System.out.println("10 : " + r.x);
        } catch (ClassCastException e) {
            System.out.println("CATCH : " + 50);
        } catch (RuntimeException e) {
            System.out.println("CATCH : " + 55);
        } catch (Exception e) {
            System.out.println("CATCH : " + 60);
        } finally {
            System.out.println("FINALLY : " + 65);
```



```
        }
    }
}

public class Rabbit extends Pet {
    public Rabbit() { y++; }

    public void f() { x = 85; }
    public void g(Object s) { x = 90; }
    public void g(String s) { x = 95; }

    public void h() {
        String s = null;
        x = s.length();
    }
}
```

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

---

9. Identify and explain three compile-time errors in the following Java code. (**6** marks) How would you correct the errors you have found (with as few changes as possible)?

```
public class Up {

  private String name;

  public Up(String name) {
    this.name = name;
  }

  public void setName(String name) {
    name = this.name;
  }

  public String getName() {
    return name;
  }
}
```

```
public class Down extends Up {

  @Override
  public Down(String name) {
    this.setName(name);
  }

  private String getName() {
    return super.getName()
      + " in the subclass";
  }

  public Object getObject() {
    return "Hello";
  }
}
```


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

```
/**
 * Computes the factorial of n for non-negative n.
 * Throws an IllegalArgumentException for negative n.
 *
 * @param n an int value for which we want to compute
 *  the factorial, should be greater than or equal to 0
 * @return factorial of n
 * @throws IllegalArgumentException if n is less than 0
 */
public static long factorial(int n) {
    if (n < 0) {
        String msg = "Illegal negative argument " + n;
        throw new IllegalArgumentException(msg);
    }
    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);
}
```

Write test cases for JUnit 4 that together achieve *test coverage* for this method (i.e., for each statement in the method `factorial` there should be at least one test case that uses the statement). Ensure that a test fails if running the test takes more than 2000 milliseconds.

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

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.

