Supervision work

Tudor Avram, Homerton College, tma33@cam.ac.uk

24.11.2015

1 Lifecycle of an Object

1.1 Exercise 28

I declared the following 3 classes :

```
package uk.ac.cam.tma33.supervision7;

class B extends A {

}

class C extends B {

}

public class A {
        public int x;

        public A() {
                x=0;
        }
}
```

For them, the following code will print "0 0 0" on the console :

```
public static void main(String[] args) {
        A a = new A();
        B b = new B();
        C c = new C();
        System.out.println(a.x);
        System.out.println(b.x);
        System.out.println(c.x);
}
```

Modified such as the constructor of A takes 1 argument, the classes will look like this :

```java
package uk.ac.cam.tma33.supervision7;

class B extends A {

        public B(int a) {
                super(a);
        }
}

class C extends B {

        public C(int a){
                super(a);
        }
}

public class A {
        public int x;

        public A(int a) {
                x=a;
        }
}
```

In this case the following code will print "2 2 2" on the console :

```java
public static void main(String[] args) {
        A a = new A(2);
        B b = new B(2);
        C c = new C(2);
        System.out.println(a.x);
        System.out.println(b.x);
        System.out.println(c.x);
}
```

1.2 Exercise 29

The *Test* method is declared as follows :

```java
public void Test(){
        x=7;
}
```

Therefore, we can deduce that it is just a simple method and NOT a constructor. This is the reason why the given code prints 0 and not 7.

1.3 Exercise 30

In Java, garbage collection checks if there are objects on the heap that are not referenced by any element from the stack. Finalizers are methods that are run when an object is garbage collected. The issue with using them in Java is that we don't know if an object ever is garbage collected or not.

2 Error Handling

2.1 Exercise 31

```
package uk.ac.cam.tma33.supervision7;

public class RetValTest {
public static String sEmail = "";

public static void extractCamEmail(String sentence)
                                    throws Exception {
   if (sentence == null || sentence.length() == 0) {
      throw new Exception("Supplied string empty");
   }
   else {
      String[] tokens = sentence.split(" ");
      for (int i = 0; i<tokens.length; i++)
         if (tokens[i].endsWith("@cam.ac.uk")) {
            sEmail = tokens[i];
         }
   }
   if (sEmail == null || sEmail.length() == 0) {
      throw new
         Exception("No @cam address in supplied string");
   }
}

public static void main(String[] args){
   try {
      extractCamEmail("My email is rkh230@cam.ac.uk");
      System.out.println("Success: " + sEmail);
   }
   catch (Exception e) {
      String msg = e.getMessage();
      System.out.println(msg);
   }
}
}
```

2.2 Exercise 32

I created the following exception :

```
package uk.ac.cam.tma33.supervision7;

public class NegativeNumberException extends Exception{

public NegativeNumberException(String msg) {
        super(msg);
}

}
```

The square-root function is :

```
public static double sqrt (double x)
                              throws NegativeNumberException {
    if (x<0) {
        String msg = "There is no real square root of ";
        msg = msg + Double.toString(x);
        throw new NegativeNumberException(msg);
    }
    else {
        double epsilon = 1e-15; //error
        double result = x; //estimate of sqrt(x)
        while (Math.abs(result - x/result)>epsilon*result) {
            result = (x/result + result) / 2.0;
        }
        return result;
    }
}
```

===== TESTS =====

For *System.out.println(sqrt(3.0))* prints 1.7320508075688772
For *System.out.println(sqrt(-3.0))* throws *NegativeNumberException*

2.3 Exercise 33

The given implementation of *pow* computes x^n correctly and has a complexity of O(n). It uses the auxiliary method *powaux* to compute this. The *powaux* method is a rather strange one, because, in stead of returning an integer (i.e. the result), it throws the exception *Answer*, which is handled in *pow* using a *try*{...}*catch*(...){...} block. In my opinion, this method of computing x^n is rather inefficient. This way, the result is passed as an object of type *Answer* and it uses the *getAns*() method to access it, which, from a compiler's point of view, is more laborious. In other words, the compiler has to execute one more operation, compared with the case when *pow* returns directly an integer.

2.4 Exercise 34

We could include the part where we need the object in the "*try*" section and the destructor will be included in the "*finally*" section. In the end, our code will look something like this :

```
try {
//Here we use the Object

Object object = new Object();
oject.doSomething();
}
finally {
// Here we call the destructor of the object

object.destroy();
}
```

2.5 Exercise 35

When the given method is executed, it returns the value 6, but it also computes the instructions in the *finally* section. If the *finally* section also contains a *return* instruction, it will be prioritised. For example, the following code will return 1 :

```
public static int x() {
        try {
                return 6;
        }
        finally {
                return 1;
        }
}
```

3 Java Collections

3.1 Exercise 39

• *Vector* —>Acts like a classic array (i.e. can access any element from the structure using an integer index). Unlike the classic arrays, which are of a fixed size, a *Vector* object can grow or shrink its size.
• *LinkedList* —>Just like the *Vector* class, it allows access at every point of

the list. Even if any element can be accessed, it CANNOT be changed. Only the elements from one of the ends can be deleted and the insertion also can be done only at one end.
• *ArrayList* —>This class is roughly equivalent to $Vector$, except that it isn't synchronised.
• *TreeSet* —>keeps the elements sorted ans guarantees $O(log(n))$ complexity for every operation (delete, insert, get).

3.2 Exercise 40

OOPList interface :

```
package uk.ac.cam.tma33.supervision7;

public interface OOPList<T> {

public T getHead() throws Exception;
public void deleteHead() throws Exception;
public void Insert(T x);
public int length();
public T getNth (int n) throws Exception;

}
```

OOPLinkedList class :

```
package uk.ac.cam.tma33.supervision7;

public class OOPLinkedList<T> implements OOPList<T>{

private OOPLinkedListElement<T> head = null;

public T getHead() throws NullPointerException {
  if (head == null) {
    throw new
       NullPointerException("This linked list is empty!");
  }
  return head.value;
}

public void deleteHead() throws NullPointerException {
  if (head == null){
    throw new
       NullPointerException("This linked list is empty!");
  }
```

```
  else head = head.next;
}

public void Insert(T x)  {
  OOPLinkedListElement<T> a = new
              OOPLinkedListElement<T>();
  if (head!=null){
    a.value = x;
    a.next = head;
  }
  else {
    a.value = x;
    a.next = null;
  }
  head = a;
}

public int length() {
  int Sol = 0;
  while (head != null) {
    Sol++;
    head = head.next;
  }
  return Sol;
}

public T getNth(int n) throws Exception {
  int p = 1;
  while (p < n && head!=null){
    p++;
    head  = head.next;
  }
  if (head == null) {
    throw new
      Exception("There is no nth element!");
  }
  return head.value;
}

}
```

I also had to change the implementation of *OOPLinkedListElement*, such as it also uses Generics :

```
package uk.ac.cam.tma33.supervision7;

public class OOPLinkedListElement<T> {

public OOPLinkedListElement<T> next;
public T value;

public OOPLinkedListElement() {
  next = null;
}

}
```

3.3 Exercise 41

```
package uk.ac.cam.tma33.supervision7;

import java.util.*;

public class Students {

private LinkedList<String> mNames;
private LinkedList<Integer> mMarks;
private TreeMap<String, String> sortedByNames;
private TreeMap<Integer, String> sortedByMarks;

//CONSTRUCTOR
public Students(){
  mNames = new LinkedList<String>();
  mMarks = new LinkedList<Integer>();
  sortedByNames = new TreeMap<String, String>();
  sortedByMarks = new TreeMap<Integer, String>();
}
//adds a student
public void addStudent(String name, int mark) {
  mNames.add(name);
  mMarks.add(mark);
  sortedByNames.put(name, name);
  sortedByMarks.put(mark, name);
}
```

```
// gets the mean mark of the students
public double getMean() throws Exception{
  if (mNames.isEmpty()) {
    throw new Exception("No students!");
  }
  double mean = 0.0;
  int S = 0;
  for (int mark : mMarks) {
    S += mark;
  }
  int n = mNames.size();
  mean = ((double) S)/((double) n);
  return mean;
}

// returns all the names of the students
public LinkedList<String> getAllNames() {
  LinkedList<String> result;
  result = new LinkedList<String>(sortedByNames.values());
  return result;
}

// returns the names of the first p%
public LinkedList<String> getPCentNames(int p) {
  LinkedList<String> list;
  LinkedList<String> result = new LinkedList<String>();
  list = new LinkedList<String>(sortedByMarks.values());
  int n = list.size();
  n = (int)(((double)(n*p))/100.0);
  if (n>list.size()) {
    n = list.size();
  }
  for (int Index = 0; Index<=n; Index++) {
    result.add(list.get(Index));
  }
  return result;
}

}
```

3.4 Exercise 42

In Java Generics, *wildcards* work as supertypes for all kinds of collection. In other words, a *wildcard* is a type that matches anything. They can be useful because this way we can make our methods abstract and, therefore, our code will become shorter and "cleaner". For example, using wildcards, we only need to declare one method of printing a collection, whatever type it is :

```
public void printCollection(Collection<?> c) {
        for (Object e : c) {
        System.out.println(e);
        }
}
```

3.5 Exercise 43

Java Generics are declared such as they are entirely compile-time construct (i.e. the compiler turns all the uses into the casts to the right type). So, anything that is used as Generics has to be convertable to Object and primitives aren't. They work the way they do because they were not initially part of the language - they were added later.