Supervision work

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

01.12.2015

1 Copying Objects

1.1 Exercise 36

(a)

```
package uk.ac.cam.tma33.supervision8;

public class MyClass {

private String mName;
private int[] mData;

// Copy Constructor

public MyClass(MyClass toCopy) {
        this.mName = toCopy.mName;
        this.mData = toCopy.mData.clone();
}

}
```

(b)

```
// Clone method
@Override
public Object clone() throws CloneNotSupportedException{
        MyClass cloned = (MyClass) super.clone();
        cloned.mName = new String(this.mName);
        cloned.mData = new int[mData.length];
        for (int i = 0; i < mData.length; i++) {
                cloned.mData[i] = new Integer(mData[i]);
        }
        return cloned;
}
```

(c) When we try to copy an object that is referenced using its parent type, our program will not compile. In order to get the right answer, we would have to manually cast the result to the desired type, and this thing is costly and can be easily forgotten.

(d) A *copy* constructor can be useful when we have a complex Object, with many references and would be hard to perform deep cloning on it.

1.2 Exercise 37

I used the following classes to test what actually happens :

```
public class SomeOtherClass {

public int a;

public Object clone() throws CloneNotSupportedException{
  return super.clone();
}

}

public class SomeClass extends SomeOtherClass
                        implements Cloneable {

private int[] mData = new int[100];

public Object clone() {
  SomeClass sc = new SomeClass();
  sc.mData = mData.clone();
  return sc;
}

}

public class CloneTester {

public static void main(String[] args) {
  SomeClass object = new SomeClass();
  object.a = 10;
  SomeClass clone = (SomeClass) object.clone();
  System.out.println(object.a);
  System.out.println(clone.a);
}

}
```

It prints, in this order :
10
0
Therefore, we can deduce that, if we don't use the super.clone() in our clone() method, we risk not to clone the parameters declared in the "mother" class. In this case, clone.a is assigned just 0, so a new integer, in stead of cloning the one from object.a (i.e. taking the value 10).

1.3 Exercise 38

The given class has an array as a parameter, which basically is a reference to a bunch of memory on the heap. The difficulty of deep-cloning this class is deep copying the array itself. We would have to create a new array and to assign to its every element a reference to an integer equal to its correspondent from *mData*

2 Object Comparison

2.1 Exercise 45

```
package uk.ac.cam.tma33.supervision8;

public class Point3D implements Comparable<Point3D>{

private int mX;
private int mY;
private int mZ;

public Point3D(int x, int y, int z) {
  mX = x;
  mY = y;
  mZ = z;
}

@Override
public int compareTo(Point3D p) {
  if (mZ < p.mZ) return -1; //this < p
  else if (mZ > p.mZ) return 1; //this > p
  else { //this.mZ == p.mZ
    if (mY < p.mY) return -1; // this < p
    else if (mY > p.mY) return 1; // this > p
    else  { //this.mY == p.mY
      if (mX < p.mX) return -1; //this < p
      else if (mX > p.mX) return 1; // this > p
      else return 0; // this == p
```

```
    }
  }
}

}
```

2.2 Exercise 46

In the first case, the programme will print $false$. This happens because s1 and s2 are two different strings and despite the fact $s1.equals(s2)$ is $true$, they are two different instances, so they are not equal, as Objects.
In the second case, the programme will print $true$. Here, both strings are assigned the same string, "Hi".
This happens because the "==" operator compares two Objects and not their contents.

2.3 Exercise 47

(a) One way of keeping the collection sorted alphabetically *without* using a Comparator is to make the car implement $Comparable < Car >$ and to override the *compareTo* method :

```
package uk.ac.cam.tma33.supervision8;

public class Car implements Comparable<Car>{

private String mManufacturer;
private int mAge;

public Car(String name, int age) {
  mManufacturer = name;
  mAge = age;
}

public void print() {
  String msg = mManufacturer;
  msg = msg + " ";
  msg = msg + Integer.toString(mAge);
  System.out.println(msg);
}

@Override
public int compareTo(Car otherCar) {
  return mManufacturer.compareTo(otherCar.mManufacturer);
}
```

```
}
```

In this case, the following code will print, in this order :
Dacia 3
Mercedes 1
Skoda 10

```
public static void main(String[] args) {
  ArrayList<Car> cars = new ArrayList<Car>();
  Car polo = new Car("Mercedes",1);
  Car fabia = new Car("Skoda", 10);
  Car logan = new Car("Dacia", 3);
  cars.add(polo);
  cars.add(fabia);
  cars.add(logan);
  Collections.sort(cars);
  for (Car car : cars) {
    car.print();
  }
}
```

(b) In this case, the *Car* class had to be added 2 more methods (*getManufacturer*() and *getAge*()) :

```
public String getManufacturer(){
  return mManufacturer;
}

public int getAge(){
  return mAge;
}
```

I also declared the Comparator *CarComparator*, which implements *Comparator* < *Car* > :

```
package uk.ac.cam.tma33.supervision8;

import java.util.Comparator;

public class CarComparator implements Comparator<Car>{

public int compare(Car c1, Car c2) {
  String m1 = c1.getManufacturer();
  String m2 = c2.getManufacturer();
  int age1 = c1.getAge();
  int age2 = c2.getAge();
  if (m1.compareTo(m2) < 0) return -1;
```

```
  else if (m1.compareTo(m2) > 0) return 1;
  else return age1 - age2;
}

}
```

I tested my class using the following code :

```
public static void main(String[] args) {
  ArrayList<Car> cars = new ArrayList<Car>();
  Car polo = new Car("Mercedes",1);
  Car fabia = new Car("Skoda", 10);
  Car logan = new Car("Dacia", 3);
  cars.add(polo);
  cars.add(fabia);
  cars.add(logan);
  Collections.sort(cars,new CarComparator());
  for (Car car : cars) {
    car.print();
  }
}
```

and it printed, in this order :
Dacia 3
Mercedes 1
Skoda 10

2.4 Exercise 48

In order to store the pairs, I used the following *Pair* class :

```
package uk.ac.cam.tma33.supervision8;

public class Pair implements Comparable<Pair>{

private int mX;
private int mY;

public Pair(int x, int y) {
  mX = x;
  mY = y;
}

public void print(){
  System.out.println(mX + "," + mY);
}

@Override
public int compareTo(Pair p) {
  if (mX < p.mX) return -1;
  else if (mX > p.mX) return 1;
  else {
    if (mY < p.mY) return -1;
```

```
    else if (mY > p.mY) return 1;
    else return 0;
  }
}

}
```

To test this and to read from the file I used the following code :

```
package uk.ac.cam.tma33.supervision8;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;

public class TestPairs {

public static void main(String[] args) throws IOException{
  FileReader reader = new FileReader("in.txt");
  BufferedReader buff = new BufferedReader(reader);
  String input = buff.readLine();
  ArrayList<Pair> pairs = new ArrayList<Pair>();
  while (input!=null) {
    String[] numbers = input.split(",");
    int x = Integer.parseInt(numbers[0]);
    int y = Integer.parseInt(numbers[1]);
    pairs.add(new Pair(x,y));
    input = buff.readLine();
  }
  buff.close();
  Collections.sort(pairs);
  for (Pair p : pairs) {
    p.print();
  }
}

}
```

For the file containing the following data :

$1, 2$
$1, 3$
$3, 4$
$5, 6$
$2, 1$
$2, 4$
$2, 3$

the programme printed, in this order :

$1, 2$
$1, 3$
$2, 1$
$2, 3$
$2, 4$
$3, 4$
$5, 6$