Universidad Carlos III de Madrid

Systems Programming

Leganés, June 17th, 2014 — Extraordinary Call (Problems)
Time: 120 min — Grade: 5 points out of 10 from the exam

Problem 1 (1.5 points)

City councils apply two types of municipal taxes to buildings. The first one is the IBI (Building Property Tax, in Spanish “Impuesto de Bienes Inmuebles”) and the second one is the TRU (Urban Garbage Tax, in Spanish “Tasa de Residuos Urbanos”). The city council of our city defines the following interface for calculating these taxes. The IBI tax amount will be calculated by multiplying the area of the building (in square meters) by the `BASIC_IBI` constant.

```
public interface CityCouncilTaxes {
  static final double BASIC_IBI = 10.0;
  static final double FACADE_SUPPLEMENT = 20.0;
  static final double ROOM_SUPPLEMENT = 15.0;
  double calculateIBI ();
  double calculateTRU ();
}
```

Section 1 (0.3 points)

Write the code for the `Estate` class, which represents every element to which the taxes must be applied (basically, housing and establishments). This class must implement the `CityCouncilTaxes` interface. Take into account the following requirements:

- `Estate` class must have the following attributes. Notice that all attributes are private, except for `age`, which is `protected` so that it can be directly accessed from the subclasses without a `get` method.

  ```
  private String name;   // Estate's name
  protected int age;     // Number of years since the estate was built
  private double area;   // Area in square meters
  ```

- The only constructor of the `Estate` class receives these three data as arguments.
- The class must implement the `calculateIBI()` method. This method calculates the basic IBI, multiplying the estate area by the `BASIC_IBI` constant defined by the city council.
- The `calculateTRU()` method must not be implemented in the `Estate` class because there is not enough information to do it. Therefore, the children classes must be the ones implementing it. Remember that you have to indicate so in the class definition.

Section 2 (0.7 points)

Write the code for the `Housing` and `Establishment` classes, both of them extending the `Estate` class. Take into account that:

- **Duplicate code or attributes will be penalized.** The attributes must be private, except indicated otherwise.
- The `Housing` class must inherit the `Estate` class attributes, and have an additional attribute, called `rooms`. This attribute is an integer representing the number of rooms in the house.

- The `Establishment` class must inherit the attributes of the `Estate` class, and have an additional attribute, called `facadeLength`. This attribute is a real number representing the number of meters the establishment occupies in the street (i.e., the length of its facade).
- The constructor of the `Housing` class must receive the name, age, area and number of rooms. The constructor of the `Establishment` class must receive the name, age, area and length of the facade in meters.
- Both classes must override the `calculateIBI()` method defined in the previous section. In order to implement it, you must use the method in the parent class (do not duplicate its code). The `calculateIBI()` method in the `Housing` class must add to the IBI returned by the parent class a 20 % increment when the building age is less than 10 years. The `calculateIBI()` method in the `Establishment` class must add an additional 10 % to the IBI returned by the method in the parent class when the establishment age is less than 20 years.
- Both classes must implement the `calculateTRU()` method, but in a different manner. In the `Housing` class, the method must return the result of multiplying the `ROOM_SUPPLEMENT` constant by the number of rooms in the house. The method in the `Establishment` class must return the result of multiplying the `FACADE_SUPPLEMENT` by the length of the establishment facade (in meters).

Section 3 (0.5 points)

Write the code for the `static void printAverageIBIMaxTRU(Estate data[])` method. This method must calculate the average value of the IBI, and the maximum value of the TRU, of the estates stored in the array received as argument, and then print these values to the standard output.

Problem 2 (3.5 points)

An ∞-ary tree is a tree that might have among 0 and infinite subtrees. It can be defined through mutual recursion in the following way:

- An **∞-ary tree** contains a datum and a forest.
- A **forest** is a queue of ∞-ary trees.

The following classes define a ∞-ary tree that uses *Strings* as data.

```
public class Tree {
   private String name;
   private Forest forest;

   public Tree(String name, Forest forest) {
      setName(name);
      setForest(forest);
   }

   private void setName(String name) {
      if (name == null) {
         throw new IllegalArgumentException();
      }
      this.name = name;
   }

   private void setForest(Forest forest) {
      if (forest == null) {
         throw new IllegalArgumentException();
      }
      this.forest = forest;
   }
}
```

```
public class Forest implements Queue<Tree> {
   private Node<Tree> top;
   private Node<Tree> tail;
   private int size;

   public Forest() {
      top = null;
      tail = null;
      size = 0;
   }

   public boolean isEmpty() {
      return (top == null);
   }

   public int size() {
      return size;
   }

   public Tree dequeue() {
      // assume this method is already coded
   }
}
```

```
public interface Queue<E> {
   boolean isEmpty();
   int     size();
   void    enqueue(E info);
   E       dequeue();
}
```

```
class Node<E> {
   private E info;
   private Node<E> next;

   public Node(E info, Node<E> next) {
      setInfo(info);
      setNext(next);
   }

   public Node<E> getNext() { return next; }
   public E getInfo() { return info; }
   public void setNext(Node<E> next) {
      // assume this method is already coded
   }
   public void setInfo(E info) {
      // assume this method is already coded
      // thrwos an exception if info is null
   }
}
```

Note: *we recommend to solve the following sections in the same order they appear.*

Note: *when solving the following sections, you CANNOT implement nor use any other additional classes or methods than the ones you are asked to implement in each section. Although, you can use the classes and methods mentioned in the initial description of the problem and the ones in the previous sections to the section your are solving.*

Section 1 (0.75 points)

Write the code for the `public void enqueue(Tree tree)` method of the `Forest` class.

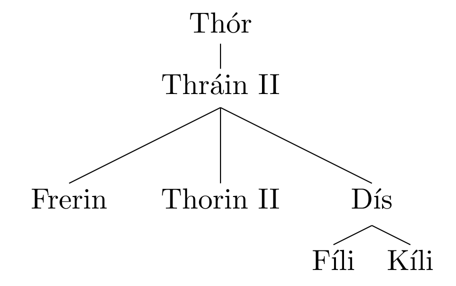
If the implementation of this method is not efficient in time, the solution will be penalized with 0.3 points.

Section 2 (0.5 points)

Write the code for the `public boolean isLeaf()` method in the `Tree` class. This method must return `true` if the tree with which the method is invoked is a leaf, or `false` otherwise.

Section 3 (1 point)

Write the code for the `public static Tree Thor()` method in a new `Test` class. This method must create, build and return the tree describing *Durin*'s descendants, from *Thór* to *Fíli* and *Kíli*, as shown in the following figure:



Section 4 (1.25 points)

Implement the `public String toString()` method in the `Tree` class. This method must return the textual representation of the tree data in pre-order, separated by end-of-line characters.

In order to solve this section you also must implement the `public String toString()` method of the `Forest` class.

For instance, the result of printing the tree shown in the previous figure would be:

```
Thór
Thráin II
Frerin
Thorin II
Dís
Fíli
Kíli
```

Hint: in Java, you can obtain a `String` with the end-of-line characters by calling the `System.lineSeparator()` method.

Solution to section 1.1 (0.3 points)

```
public abstract class Estate implements CityCouncilTaxes {
  private String name;
  protected int age;
  private double area;

  public Estate (String name, int age, double area){
    this.name = name;
    this.age = age;
    this.area = area;
  }

  public double calculateIBI(){
    return (area * BASIC_IBI);
  }

  public abstract double calculateTRU();
}
```

Solution to section 1.2 (0.7 points)

```
public class Housing extends Estate {

  private int rooms;

  public Housing (String name, int age, double area, int rooms){
    super(name, age, area);
    this.rooms = rooms;
  }

  public double calculateIBI() {
    if(super.age < 10) {
      return super.calculateIBI() * 1.20;
    } else {
      return super.calculateIBI();
    }
  }

  public double calculateTRU() {
    return (ROOM_SUPPLEMENT * rooms);
  }
}
```

```
public class Establishment extends Estate {
  private double facadeLength;

  public Establishment (String name, int age, double area, double facadeLength) {
    super(name, age, area);
    this.facadeLength = facadeLength;
  }

  public double calculateIBI() {
    if (super.age < 20) {
      return super.calculateIBI() * 1.10;
    } else {
      return super.calculateIBI();
    }
  }

  public double calculateTRU() {
    return (FACADE_SUPPLEMENT * facadeLength);
  }
}
```

Solution to section 1.3 (0.5 points)

```
public static void printAverageIBIMaxTRU(Estate data[]) {

  double sumibi = 0.0;
  double maxtru = 0.0;

  for (int i=0; i<data.length; i++) {
    sumibi += data[i].calculateIBI();
    if (data[i].calculateTRU() > maxtru) {
      maxtru = data[i].calculateTRU();
    }
  }
  System.out.println("Average IBI" + (sumibi / data.length));
  System.out.println("Maximum TRU" + maxtru);
}
```

Solution to section 2.1 (0.75 points)

```
public void enqueue(Tree tree) {
    Node<Tree> n = new Node<Tree>(tree, null);
    if (isEmpty()) {
        top = n;
    } else {
        tail.setNext(n);
    }
    tail = n;
    size++;
}
```

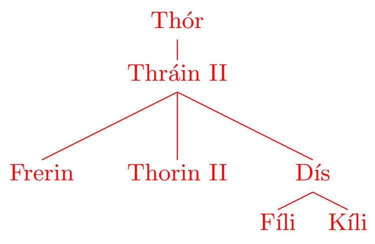
```
* 0,15 si tienen bien los generics

* 0,15 si tienen bien el caso en que la cola está vacía

* 0,30 si tienen bien el caso en la cola no está vacía (ignorando si es
  eficiente o no).

* 0,15 si incrementan el tamaño

* (-0.15) si tienen mal la signatura del método (por ejemplo, si usan E en vez
  de Tree)

* (-0.30) si tienen mal la construcción del nodo

* (-0,30) si recorren la cola para insertar
```

Solution to section 2.2 (0.5 points)

```
public boolean isLeaf() {
    return forest.isEmpty();
}
```

```
* 0,5 si está bien, 0 en caso contrario.
```

Solution to section 2.3 (1 point)



```
public class Test {

    public static Tree Thor() {

        // Fíli and Kíli
        Tree fili = new Tree("Fíli", new Forest());
        Tree kili = new Tree("Kíli", new Forest());

        // Dís
        Forest forest_dis = new Forest();
        forest_dis.enqueue(fili);
        forest_dis.enqueue(kili);
        Tree dis = new Tree("Dís", forest_dis);

        // Frerin and Thorin II
        Tree frerin = new Tree("Frerin", new Forest());
        Tree thorin2 = new Tree("Thorin II", new Forest());

        // Thráin II
        Forest forest_thrain2 = new Forest();
        forest_thrain2.enqueue(frerin);
        forest_thrain2.enqueue(thorin2);
        forest_thrain2.enqueue(dis);
        Tree thrain2 = new Tree("Thráin II", forest_thrain2);

        // Thór
        Forest forest_thor = new Forest();
        forest_thor.enqueue(thrain2);
        Tree thor = new Tree("Thór", forest_thor);

        return thor;
    }
}
```

```
No importa si no declaran la clase Test.

* 0,25 si usan correctamente el constructor de Tree

* 0,25 si usan correctamente el constructor de Forest

* 0,25 si encolan correctamente los árboles en los bosques

  # Estos 0,25 se reducen a 0,10 si el órden es el inverso

* 0,25 si el árbol que construyen es correcto (las relaciones entre padres e
  hijos son correctas)

* (-0,25) si no retornan el árbol una vez construido

* Si usan nulls, como mucho tendrán un 0,25 si todo lo demás está bien.
```

Solution to section 2.4 (1.25 points)

```
// Tree.java
public String toString() {
    if (isLeaf()) {
        return name +
               System.lineSeparator();
    } else {
        return name +
               System.lineSeparator() +
               forest.toString();
    }
}

// Forest.java
public String toString() {
    String retval = new String();
    for (Node<Tree> current = top;
         current != null;
         current = current.getNext()) {
        retval += current.getInfo().toString();
    }
    return retval;
}
```

```
* 0,50 si se nota que entienden el concepto de recursión mútua, aunque tengan
  otras partes del apartado mal.

* 0,15 si Tree.toString() es correcta en el caso base (ignorando el uso de
  EOL)

* 0,15 si Tree.toString() es correcta en el caso recursivo (ignorando el uso
  de EOL)

* 0,30 si Forest.toString() es correcta, independientemente si lo resuelven
  mediante recursión o iteración.

  # Estos 0,30 se reducen a 0,10 si se dejan el último nodo por recorrer

* 0,15 si utilizan y emplazan correctamente el fin de línea.

No importa si Tree.toString() está resuleto de forma iterativa usando la cola
del enunciado. En ese caso se puntuará con 0,30 si la implementación es
correcta (que es justo la suma del caso base y el recursivo si se hace por
recursión).

No importa si usan un String o un StringBuffer.
```