IIP Second Partial - ETSInf
January 9th, 2017. Time: 2 hours and 30 minutes.

1. [6 points] You have available the `Building` class, that represents a building at UPV by using two data sets: those associated to the physical building (GPS coordinates and identity code in a map), and those associated to its assigned use (type of use and name of the using entity). This class is known from previous uses and below there is a summary of its documentation:

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	`DEPARTMENT` Constant that represents a departmental type for the building.
`static int`	`SCHOOL` Constant that represents a teaching type for the building.
`static int`	`SERVICES` Constant that represents other activities for the building, such as offices, restaurants, ...

Constructor Summary

Constructors

Constructor and Description
`Building()` Create a Building with coe "1F", uses by entity "DSIC", of departmental type and in coordinates (39.4625, -0.3472).
`Building(java.lang.String c, java.lang.String e, int t, RealPoint p)` Creates a Building with code c, used by entity e, of type t and coordinates given by RealPoint p.

Method Summary

Methods

Modifier and Type	Method and Description
`int`	`closestToRectorate(Building e)` Given a Building e, returns: -1 if building this is closer to rectorate than building e; 1 if e is closer to rectorate than this; or 0 when both building are at the same distance to rectorate.
`boolean`	`equals(java.lang.Object o)` Returns true when o is a Building wit the same code and coordinates then this; otherwise, returns false.
`java.lang.String`	`getEntity()` Returns the entity that uses building this.
`int`	`getType()` Returns the type of building this.

You must: Implement the class `VeraMap` that represents the buildings at Vera campus of UPV by using the following attributes and methods (remember to employ the constants defined in `Building` and `VeraMap` wherever required):

a) (0.5 points) Attributes (only first one is public):

- `MAX_BUILD`, class (`static`) constant that represents the maximum number of buildings in the map, with value 50.
- `numBuild`, integer in the interval `[0..MAX_BUILD]` that represents the actual number of buildings in the map in each moment.

- `buildings`, array with base datatype `Building`, of size `MAX_BUILD`, that stores the buildings in the map in each moment, that are in consecutive positions in the array, from `0` to `numBuild - 1` both included, **in ascendent order according to their distance to rectorate**, being `buildings[0]` rectorate, `buildings[1]` the closest to rectorate, and `buldings[numBuilds - 1]` the farthest to rectorate. When two buildings are **at the same distance** to rectorate, they are in consecutive positions of the `buildings` array (`i` and `i+1`, with $1 \leq$ `i` $<$ `numBuild - 1`), being `Building` in `buildings[i+1]` an object added to the array **after** that in `buildings[i]`.
- `numSchools`, non-negative integer that represents the number of teaching buildings in the map in each moment.

b) (1 point) A default constructor (without parameters) that creates a `VeraMap` object with a single building with the following features: a service building, used by entity ``Rectorate'', with code ``3A'' and coordinates `(39.4823, -0.3457)`.

c) (1.5 points) A method with header:

```
private int positionOf(Building e)
```

that given a `Building e`, returns the position of the first building (that of lowest index) in the array that is farther to rectorate than `e`, or `numBuild` if no building is farther to rectorate than `e`.

You must employ the method `closestToRectorate` from the `Building` class.

d) (1.5 points) A method with header:

```
public boolean add(Building e)
```

that, given a `Building e` that **is not in the map**, adds it if there is space in the array, in an **ordered** form according its distance to rectorate, updating the attributes `numBuild` and, if needed, `numSchools`. The method returns `true` when addition was successful, and `false` if no more buildings can be added to the map.

When `e` can enter in the array, you must use the private method `positionOf(Building)` in order to know which position of the array `buildings` is the one in which you have to put building `e`. Once that position is found, you must provide `e` the needed space in the array. Thus, you must use a private method `already implemented` with header:

```
private void moveRight(int begin, int end)
```

that moves a position to the right all the elements of the array `buildings` from position `begin` to position `end` both included ($0 \leq$ `begin` $\leq$ `end` $\leq$ `numBuild - 1` $<$ `buildings.length - 1`). According to precondition, if `begin` $>$ `end`, no movement is done.

e) (1.5 points) A method with header:

```
public Building[] filterSchoolType()
```

that returns an array of `Building` with the teaching building (or schools) from the map. The size of this array must be equal to the number of teaching buildings in the map, or 0 if no building of that type is present in the map.

Solution:

```
public class VeraMap {
    public static final int MAX_BUILD = 50;
    private int numBuild;
    private Building[] buildings;
    private int numSchools;

    public VeraMap() {
        buildings = new Building[MAX_BUILD];
        buildings[0] = new Building("3A", "Rectorate",
```

```
            Building.SERVICES, new RealPoint(39.4823, -0.3457));
        numBuild = 1;
        numSchools = 0;
    }

    private int positionOf(Building e) {
        int i = 1;
        while (i < numBuild && buildings[i].closestToRectorate(e) <= 0) { i++; }
        return i;
    }

    /** Precondition: 0 <= begin <= end <= numBuild - 1 < buildings.length - 1*/
    private void moveRight(int begin, int end) {
        for (int pos = end + 1; pos > begin; pos--) {
            buildings[pos] = buildings[pos - 1];
        }
    }

    /** Precondition: e is not in the map */
    public boolean add(Building e) {
        boolean res = false;
        if (numBuild != MAX_BUILD) {
            int pos = positionOf(e);
            moveRight(pos, numBuild - 1);
            buildings[pos] = e;
            numBuild++;
            if (e.getType() == Building.SCHOOL) { numSchools++; }
            res = true;
        }
        return res;
    }

    public Building[] filterSchoolType() {
        Building[] aux = new Building[numSchools];
        int k = 0;
        for (int i = 1; i < numBuild && k < numSchools; i++) {
            if (buildings[i].getType() == Building.SCHOOL) {
                aux[k] = buildings[i];
                k++;
            }
        }
        return aux;
    }
}
```

2. 2 points Let be an integer `n` $\geq$ 2. **You must:** implement a class (`static`) method that, for all integers between 2 and `n` both included, returns a `String` with the list of their proper divisors. Remember that *proper divisors* of an integer are all its divisors except itself and the unit. For example, for `n = 18`, the method must return the following `String`:

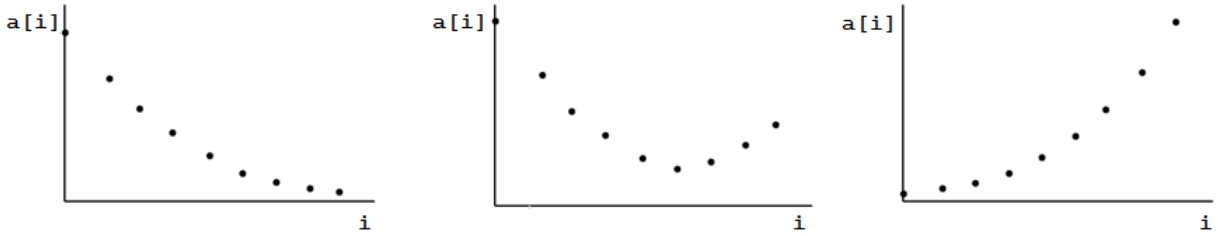
```
Proper divisors of 2:
Proper divisors of 3:
Proper divisors of 4: 2
Proper divisors of 5:
Proper divisors of 6: 2 3
Proper divisors of 7:
Proper divisors of 8: 2 4
```

```
Proper divisors of 9: 3
Proper divisors of 10: 2 5
Proper divisors of 11:
Proper divisors of 12: 2 3 4 6
Proper divisors of 13:
Proper divisors of 14: 2 7
Proper divisors of 15: 3 5
Proper divisors of 16: 2 4 8
Proper divisors of 17:
Proper divisors of 18: 2 3 6 9
```

Solution:

```
/** Precondition: n >= 2 */
public static String properDivisors(int n) {
    String res = "";
    for (int i = 2; i <= n; i++) {
        res += "Proper divisors of " + i + ": ";
        for (int j = 2; j <= i / 2; j++) {
            if (i % j == 0) { res += j + " "; }
        }
        res += "\n";
    }
    return res;
}
```

3. 2 points We have an array `a` of real numbers and length `n` $\geq$ 2, such that its components fits to the profile of a concave curve, i.e., there exists a minumum in a position `k`, 0 $\leq$ `k` $<$ `n` (thus, the values in `a[0..k]` strictly decrease and the values in `a[k..n - 1]` strictly increase); minimum can be in an extreme of the array. **You must:** implement a class (`static`) method such that, given the array, returns the position of its minimum. For example, for the arrays of the following figures, the method must return 8, 5, and 0, respectively.



Solution:

```
/** Precondition: components in a, a.length >= 2,
 *  fit the profile of a concave curve.
 */
public static int minConcave(double[] a) {
    int i = 0;
    while (i < a.length - 1 && a[i] > a[i + 1]) { i++; }
    return i;
}
```