Aitana González Cárdenas - Álvaro Jiménez Barreno
U166651 - U186356

REPORT PRÀCTICA 3

1. ***An introduction where the problem is described. For example, what should the program do? Which classes do you have to define? Which methods do you have to define? Which methods do you have to implement for these classes?***

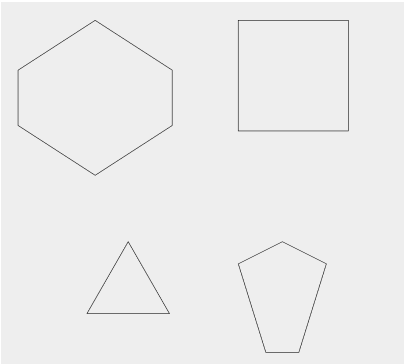
En aquesta tercera pràctica, la idea principal ha estat estendre el disseny de la segona pràctica i afegir més classes per fer el nostre món una mica més complex i sofisticat.

Inicialment, en el laboratori 2, teníem les classes territorials Point, PolygonalRegion, Continents i World. A més, totes les classes entre elles estaven relacionades mitjançant composició. Ara, en aquesta pràctica, hem implementat més classes: GeoPoint, City, Region i Country. A més, unes classes hereten d'altres class, pel qual també treballem el concepte d'*inherithance* en Java, analitzant i implementant en codi el que comporta.

Addicionalment, també hem fet la part opcional. Això implica que també hem implementant dues classes extra (pertinents al Seminari 3): Lake i Ocean. L'objectiu d'aquestes classes ha estat saber jugar amb les funcions proporcionades per la classe Graphics, així com canviar el color del contorn de les figures imprimides per pantalla, o emplenar-les d'un color en concret.

Com a objectiu general, en fer RUN al programa, s'havia de mostrar el nostre mapa, ara més complex, amb les regions agrupades (com a la pràctica 2), però ara amb ciutats (punts geomètrics) i el nom d'aquestes (labels). A més, hem fet figures amb les vores blaves, dins de instàncies Country per referir-nos a les instàncies Lake i figures amb color de fons blau entre instàncies Continent per referir-nos a les instàncies Ocean.

1. ELEMENTS DE LA PRÀCTICA 2: CANVIS A FER



Com a primer punt introductori, volem esmentar l'estat final de la pràctica 2.

Veiem que una regió poligonal està composada d'instàncies de tipus Point. Diferents regions poligonals estan agrupades per continents i, finalment, diferents continents formen el nostre món.

Ara això canviarà, perquè la classe geogràfica més petita que hi ha és City. Aquesta classe hereta de GeoPoint i aquesta de Point. Per tant, veiem que la primera classe que haurem de definir serà GeoPoint (ja que Point ja estava creada).

Seguidament, veiem que hem de definir Country amb totes les característiques que s'esmenten en l'enunciat de la pràctica. Tot i així, a l'enunciat es veu que Country hereta de PolygonalRegion i, aquesta mateixa, hereta de Region. Per tant, l'ordre de definició de classes ha estat de Region, PolygonalRegion, Country i, un cop definides totes aquestes classes, hem modificat Continent.

Finalment, hem afegit les dues classes extra, Lake i Ocean. Aquestes, però, tenen certes restriccions esmentades en el Seminari previ al laboratori. Per tant, un cop creades, hem hagut de fer canvis a altres classes ja definides prèviament. Aquestes són Country i World.

2. DISSENY DE GEOPOINT

Un cop hem organitzat el procés que haurem de fer per realitzar la pràctica, comencem dissenyant la classe GeoPoint. Aquesta, com hereta de la classe Point (concepte que explicarem al punt 2), només cal afegir *extends* a la classe i definir els atributs exclusius de GeoPoint.

```
public class GeoPoint extends Point {...}
```

D'aquesta forma, GeoPoint té els atributs de la superclasse Point i els seus. Això, però, comporta canvis en la creació del constructor. Ara, per crear instàncies de tipus GeoPoint, passem per paràmetre tots els atributs que té i, en el constructor, cridem als atributs de Point amb la funció **super**. Després simplement actualitzem el name.

```
public GeoPoint (double x, double y, String name) {
    super(x, y);
    this.name = name;
}
```

L'únic mètode que necessitem a part del constructor en aquesta classe és el mètode Draw. Necessitem una funció que ens permeti dibuixar tot el que seran ciutats amb punts i una etiqueta que indiqui el nom de la ciutat. Per fer-ho, creem 4 variables:

- Dues d'elles (height i width) seran per declarar els paràmetres d'alçada i amplada per la funció fillOval(), que més endavant utilitzarem. Les hem establert a 10. Les establim com a int, ja que la funció només accepta variables de tipus int.

- Les altres dues són les coordenades del punt. En aquestes variables cridem als mètodes getX() i getY(), definits a Point, amb un cast també de tipus int per la mateixa raó.

```
public void drawGP(Graphics g){

    int pointX = (int) getX();
    int pointY = (int) getY();
    int width = 10;
    int height = 10;

    g.setColor(Color.black);
    g.fillOval(pointX, pointY, width, height);
    g.drawString(name, pointX, pointY-5);
}
```

Finalment, cridem a la funció fillOval per dibuixar el punt i a drawString() per mostrar per pantalla el nom de la ciutat.

3. DISSENY DE CITY

Un cop hem definit la classe GeoPoint, dissenyem el codi de la classe City. Aquest mòdul és més simple que l'anterior, ja que ho hem definit tot a GeoPoint. Sabem que City hereta de GeoPoint, per tant, tota instància de tipus City tindrà les característiques de GeoPoint (ho fem amb *extends*).

```
public class City extends GeoPoint {...}
```

A més, declarem un atribut extra, numhab, que ens indica el nombre d'habitants d'una ciutat. Cal dir que no l'hem utilitzat, però l'hem volgut deixar ja que estava especificat a l'esquema de la pràctica.
Pel que fa als mètodes, al constructor de City fem el mateix que hem fet amb GeoPoint. Els atributs passats per paràmetre d'altres superclasses els incloem tots en la funció super i actualitzem el nombre d'habitants.

```
public City (double x, double y, String name, int numhab) {
    super(x, y, name);
    this.numhab = numhab;
}
```

Per dibuixar ciutats no cal dissenyar una funció específica de draw que ens dibuixi instàncies de classe City, ja que com tota instància City és de classe GeoPoint, l'únic que fem és cridar a **drawGeoPoint(g);** amb un super (mètode de la superclasse).

```
public void drawCity(Graphics g) {
    super.drawGP(g);
}
```

4. DISSENY DE REGION I CANVIS A POLYGONALREGION

Si pensem la classe Region de forma teòrica, veiem que tota regió pot ser o PolygonalRegion o EllipsoidalRegion (seguint l'esquema, tot i que no definim res relacionat amb EllipsiodalRegion ja que no s'especifica). Per tant, tota instància Region realment serà PolygonalRegion o EllipsoidalRegion.

És per això que al codi fem Region abstracte. Realment no fem instàncies de classe Region, sinó que aquestes seran instàncies de classe PolygonalRegion. Això implicarà, per tant, que PolygonalRegion heretarà de Region.

A REGION:

Declarem la classe abstracta amb un *public abstract class* i definim els mètodes que tindrà la seva subclasse, PolygonalRegion amb un *public abstract*.

```
import java.awt.Graphics;

public abstract class Region {

    public abstract double getArea();
    public abstract void drawPolyRegion(Graphics g);
}
```

A POLYGONALREGION:

Creem l'herència amb la línia de codi:

```
public class PolygonalRegion extends Region {...}
```

5. DISSENY DE COUNTRY I CANVIS A CONTINENT

Country és la següent classe que definim. Aquesta hereta de PolygonalRegion, per tant, declarem la herència de la mateixa forma que ho hem fet amb la resta de classes.

```
public class City extends GeoPoint{...}
```

D'altra banda, creem diversos atributs que pertanyen a la classe: un nom, una llista de ciutats, una llista de països veïns i una instància de tipus City, que serà la capital.

El constructor el definim de la mateixa forma. Actualitzem tots els atributs que passem per paràmetre i points, l'atribut provinent de la classe PolygonalRegion, el definim amb un super.

```
public Country (LinkedList<Point> points, String name, City capital){
    super(points);
    this.name = name;
    cities = new LinkedList<City>();
    neighbors = new LinkedList<Country>();
    this.capital = capital;
```

Inicialment, tindrem en aquest mòdul tres mètodes:

- **void addCity (City city):** Afegim una instància de tipus City a la llista de ciutats.

```
public void addCity (City city) {
    cities.add(city);
}
```

- **void addNeighbor (Country country):** Afegim una instància de tipus Country a la llista de països veïns.

```
public void addNeighbor (Country country) {
    neighbors.add(country);
}
```

- **void drawCountry (Graphics g):** En aquest mètode fem inicialment dues tasques. Per una part imprimim per pantalla totes les regions del nostre mapa, cridant a drawPolyRegion() amb un super. D'altra banda, amb la llargada de la llista de ciutats, fem un for() i, per cadascuna de les instàncies de la llista de ciutats, cridem a drawCity().

```
public void drawCountry (Graphics g) {

    int nCities = cities.size();
    super.drawPolyRegion(g);

    for(int i = 0; i < nCities; i++){
        cities.get(i).drawCity(g);
    }
```

6. DISSENY DE LAKE I OCEAN; CANVIS EN COUNTRY I WORLD

En aquest punt, vam fer una pausa en la implementació del codi i vam veure que tot ens funcionava al MyMap. Quan vam realitzar aquest *punt de check*, vam implementar les dues classes extra, Lake i Ocean.

6.1. Disseny de Lake

Per fer-ho més simplificat, vam establir que un Lake es formava a partir d'una llista d'instàncies de tipus Point. La idea principal era que es mostressin per pantalla les diferents instàncies de tipus Lake. Per tant, vam fer només dos mètodes

- El constructor de la classe: Només actualitzem la llista de punts que li passem per paràmetre, és a dir, l'atribut de la classe.

```
public Lake (LinkedList<Point> points){
    this.points = points;
}
```

- drawLake(): Aquest mètode és molt similar al mètode draw de PolygonalRegion, ja que també li passem una llista de punts. Per diferenciar llavors les instàncies Lake de les PolygonalRegion, vam afegir les següents línies de codi:

```
g.setColor(Color.blue);
g.drawPolygon(xPoints, yPoints, nPoints);
```

Això, amb la resta del codi del mètode, ens permet mostrar per pantalla figures blaves (només el contorn), que identifiquem com instàncies de classe Lake.

```
public void drawLake(Graphics g){

    int nPoints = points.size();
    int xPoints[] = new int[nPoints];
    int yPoints[] = new int[nPoints];

    for (int i = 0; i < points.size(); i++) {
        xPoints[i] = (int) points.get(i).getX();
        yPoints[i] = (int) points.get(i).getY();
    }

    g.setColor(Color.blue);
    g.drawPolygon(xPoints, yPoints, nPoints);
}
```

Com a conseqüència, vam haver de fer certs canvis a la classe Country, ja que la classe Lake està relacionada amb aquesta (mitjançant composició).

- Hem afegit un nou atribut, que fos una llista d'instàncies de tipus Lake.

```
private LinkedList<Lake> lakes;
```

- Al constructor, creem una llista buida de llacs. Comencem suposant que un país no te llac. Si el té, l'afegim amb un mètode específic.

```
neighbors = new LinkedList<Country>();
```

```
public Country (LinkedList<Point> points, String name, City capital) {
    super(points);
    this.name = name;
    cities = new LinkedList<City>();
    neighbors = new LinkedList<Country>();
    lakes = new LinkedList<Lake>();
    this.capital = capital;
}
```

- Creem un mètode per afegir llacs a la llista d'instàncies de tipus Lake, addLake().

```
public void addLake (Lake lake){
    lakes.add(lake);
}
```

- Al drawContry(), ara també necessitem una variable per saber la mida de la llista de llacs i, per cada element de la llista, amb un bucle for, cridem a la funció drawLake().

```
public void drawCountry (Graphics g) {

    int nCities = cities.size();
    super.drawPolyRegion(g);

    for(int i = 0; i < nCities; i++){
        cities.get(i).drawCity(g);
    }

    int nLakes = lakes.size();

    for(int i = 0; i < nLakes; i++){
        lakes.get(i).drawLake(g);
    }
}
```

6.2. Disseny d'Ocean

Un cop feta la classe Lake, també ens cal fer la classe Ocean. Aquesta té la mateixa dinàmica que la classe Lake. Per tant, tindrà només un atribut, una llista d'instàncies de tipus Point:

```
private LinkedList<Point> points;
```

Com a mètodes, tindrà un constructor i un draw().

- El constructor només ens servirà per inicialitzar instàncies de classe Ocean, actualitzant només la llista de punts.

- La funció drawOcean() serà igual que la funció drawLake() però afegim al final del mètode la instrucció que ens permetrà emplenar la figura d'un color, en aquest cas, blau. Com hem fet servir la funció drawPolygon(), podem emplenar-lo amb la funció fillPolygon(), amb els mateixos paràmetres.

Un cop creada la classe, hem decidit implementar-la a World. Aquesta classe, per tant, tindrà una llista de continents i una llista d'oceans com a paràmetres.

```
private LinkedList<Continent> conts;
private LinkedList<Ocean> oceans;
```

Addicionalment, al constructor de la classe, haurem d'afegir per paràmetre una llista d'oceans i actualitzar-los. Això ho fem a World i no a Country amb Lake degut a que al seminari se'ns va especificar que un món havia de tenir com a mínim una instància de classe Ocean, però un país podia no tenir cap instància de classe Lake.

7. REDISSENY DE MYMAP

Tal com vam fer a la pràctica anterior, un cop definides i implementades totes les classes necessàries, creem al mòdul de MyMap tot d'instàncies per poder agrupar-les entre elles segons la jerarquia establerta i esmentada al principi de l'informe. Veiem que un cop arribem a crear una instància de tipus World (que anomenem world), la incloem en la funció paint per imprimir per pantalla tot el nostre món.

La idea és fer llistes de tipus Point agrupades en instàncies Lake i crear instàncies de tipus City. Un cop tenim aquestes establertes, les agrupem en instàncies Country (delimitades per figures PolygonalRegion), agrupant aquestes últimes en instàncies de tipus Continent. Tot seguit, tornem a crear instàncies Point que agrupem en instàncies Ocean. Finalment, agrupem les instàncies Continent i Ocean (independents entre elles) en una instància World.

```
LinkedList< Point > points1 = new LinkedList< Point >();
points1.add( new Point( 100, 190 ) );
points1.add( new Point( 240, 100 ) );
points1.add( new Point( 380, 190 ) );
points1.add( new Point( 380, 290 ) );
points1.add( new Point( 240, 380 ) );
points1.add( new Point( 100, 290 ) );

LinkedList< Point > points2 = new LinkedList< Point >();
points2.add( new Point( 500, 540 ) );
points2.add( new Point( 580, 500 ) );
points2.add( new Point( 660, 540 ) );
points2.add( new Point( 610, 700 ) );
points2.add( new Point( 550, 700 ) );


LinkedList< Point > points3 = new LinkedList< Point >();
points3.add( new Point( 500, 100 ) );
points3.add( new Point( 700, 100 ) );
points3.add( new Point( 700, 300 ) );
points3.add( new Point( 500, 300 ) );


LinkedList< Point > points4 = new LinkedList< Point >();
points4.add( new Point( 300, 500 ) );
points4.add( new Point( 375, 630 ) );
points4.add( new Point( 225, 630 ) );
```

```
City barcelona = new City(350, 170, "Barcelona", 1620000);
City pontevedra = new City(170, 140, "Pontevedra", 83029);
City malaga = new City(235, 375, "Màlaga", 569005);
City madrid = new City(235, 240, "Madrid", 3223000);

City bangkok = new City(580, 600, "Bangkok", 2161000);
City huahin = new City(540, 670, "Hua Hin", 513275);
City pai = new City(540, 515, "Pai", 471941);

City miami = new City(525, 295, "Miami", 1472000);
City washingtonDC = new City(595, 200, "Washington DC", 3645000);

City laHabana = new City(295, 580, "La Habana", 821752);
```

```
Country cuba = new Country(points4, "Cuba", laHabana);
cuba.addCity(laHabana);

LinkedList< Country > countries1 = new LinkedList< Country >();
countries1.add(espanya);

LinkedList< Country > countries2 = new LinkedList< Country >();
countries2.add(eeuu);
countries2.add(cuba);

LinkedList< Country > countries3 = new LinkedList< Country >();
countries3.add(tailandia);

Continent continent1 = new Continent(countries1);
Continent continent2 = new Continent(countries2);
Continent continent3 = new Continent(countries3);

LinkedList< Continent > continents = new LinkedList<Continent>();
continents.add(continent1);
continents.add(continent2);
continents.add(continent3);
```

```
LinkedList< Point > points5 = new LinkedList< Point >();
points5.add( new Point( 285, 210 ) );
points5.add( new Point( 360, 210 ) );
points5.add( new Point( 360, 270 ) );
points5.add( new Point( 285, 270 ) );

LinkedList< Point > points6 = new LinkedList< Point >();
points6.add( new Point( 570, 610 ) );
points6.add( new Point( 620, 640 ) );
points6.add( new Point( 545, 640 ) );

LinkedList< Point > points7 = new LinkedList< Point >();
points7.add( new Point( 520, 540 ) );
points7.add( new Point( 550, 525 ) );
points7.add( new Point( 580, 540 ) );
points7.add( new Point( 580, 570 ) );
points7.add( new Point( 550, 585 ) );
points7.add( new Point( 520, 570 ) );

Lake covadonga = new Lake(points5);
Lake khao = new Lake(points6);
Lake pang = new Lake(points7);
```

```
LinkedList< Point > points8 = new LinkedList< Point >();
points8.add( new Point( 380, 330 ) );
points8.add( new Point( 520, 400 ) );
points8.add( new Point( 660, 330 ) );
points8.add( new Point( 660, 440 ) );
points8.add( new Point( 520, 500 ) );
points8.add( new Point( 380, 440 ) );

LinkedList< Point > points9 = new LinkedList< Point >();
points9.add( new Point( 410, 780 ) );
points9.add( new Point( 485, 650 ) );
points9.add( new Point( 335, 650 ) );

Ocean ocean1 = new Ocean(points8);
Ocean ocean2 = new Ocean(points9);

LinkedList< Ocean > oceans = new LinkedList<Ocean>();
oceans.add(ocean1);
oceans.add(ocean2);
```

I, finalment,

```
world = new World(continents, oceans);
```

A la funció paint:

```
super.paint(g);
world.drawW(g);
```

2. ***A description of possible alternative solutions that were discussed, and a description of the chosen solution and the reason for choosing this solution rather than others. It is also a good idea to mention the related theoretical concepts of object-oriented programming that were applied as part of the solution.***

Durant la pràctica, ens hem trobat amb diverses dificultats i incògnites que ens han fet redefinir certs conceptes.

Per entendre bé la pràctica, vam revisar el concepte d'herència, i quins conceptes s'havien de treballar quan hi havia una relació d'aquest tipus. Veiem que quan existeix una relació d'herència entre classes (Class A is a Class B), a l'heretar d'una classe base heretem tant els atributs com els mètodes (els constructors són utilitzats però no heretats). Això ens ajudar a crear classes especialitzades, més concretes. Al cap i a la fi, estem creant classes molt semblants entre elles (amb un mateix concepte), però amb certes característiques especials per cada classe. És a dir, una instància de classe City té totes les característiques d'una instància de classe GeoPoint (mateix concepte) i, a més, té característiques pròpies de la classe City.

Amb aquest concepte, hem après a portar a codi la relació entre superclasses i subclasses, a relacionar-les entre elles amb la instrucció **extends**, i a programar els constructors amb atributs que provenen de superclasses o implementar mètodes també de la superclasse (amb la instrucció super()).

D'altra banda, a l'hora d'implementar els colors en les funcions draw(), no ens servia directament escriure les instruccions (setColor(), fillPolygon()...). Per que ens funcionés, vam haver d'importar *java.awt.Color*. Principalment, als exemples, ens sortia que havíem d'importar *java.awt.**. Tot i així, vam veure que amb color també ens funcionava i, com no vam saber trobar el significat de *java.awt.** i ens semblava més coherent *java.awt.Color*, vam decidir canviar-ho.

A l'hora d'implementar les classes Lake i Ocean, vam passar primerament per paràmetre una LinkedList de Lake a Country, i una LinkedList d'Ocean a World i actualitzàvem les llistes al constructor. Tot i així, vam veure més endavant que a Lake no estava del tot optimitzat. Podria existir el cas que un país no tingués cap Lake. Per tant, hauríem de crear una llista buida per tots aquelles instàncies de Country que no tinguessin cap llac i passar-la per paràmetre. En comptes de fer això, vam decidir crear directament una llista buida al constructor de Country i, addicionalment, crear una funció per afegir instàncies de tipus Lake a la llista lakes.
Això no ho vam fer a Ocean ja que un món ha de tenir com a mínim un oceà (informació proporcionada al seminari 3).

3. ***A conclusion that describes how well the solution worked in practice, i.e. did the tests show that the classes vere correctly implemented? You can also mention any difficulties during the implementation as well as any doubts you might have had.***

Com a conclusions generals de la pràctica, tot i que en un principi ens ha costat més enfrontar-nos a aquesta pel fet d'haver d'aplicar nova teoria realitzada a classe al nostre codi, com ha sigut el cas del mecanisme d'herència, gràcies a la visualització de les diferents classes i relacions entre aquestes realitzades al seminari, creiem que ens n'hem sortit força bé.

Així mateix, la redacció d'un informe després de cada pràctica ens ajuda a assimilar tant els continguts teòrics com els continguts pràctics realitzats a les diferents classes de l'assignatura.

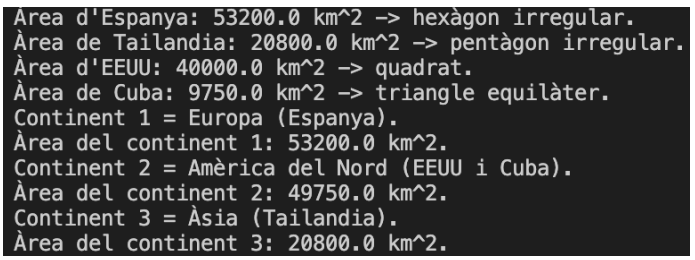
No obstant, volem destacar un parell d'inconvenients que s'han generat al llarg de la realització de la pràctica que podem dir que hem resolt.

Principalment, ens vam quedar una mica estancats quan vam haver d'implementar la classe Region, ja que inicialment no la vam pensar com una classe abstracta. Quan vam deduir aquest fet, vam haver de repassar les nomenclatures, les declaracions i les crides de funcions de sub-classes i super-classes per la correcta realització del nostre codi.
Cal afegir que ens va passar algo semblant a l'hora d'implementar els diferents mètodes de la classe Country. En un principi no no vam pensar que la funció de drawCountry() havia de cridar a la funció drawP(), implementat a PolygonalRegion, mitjançant la forma super.drawP().

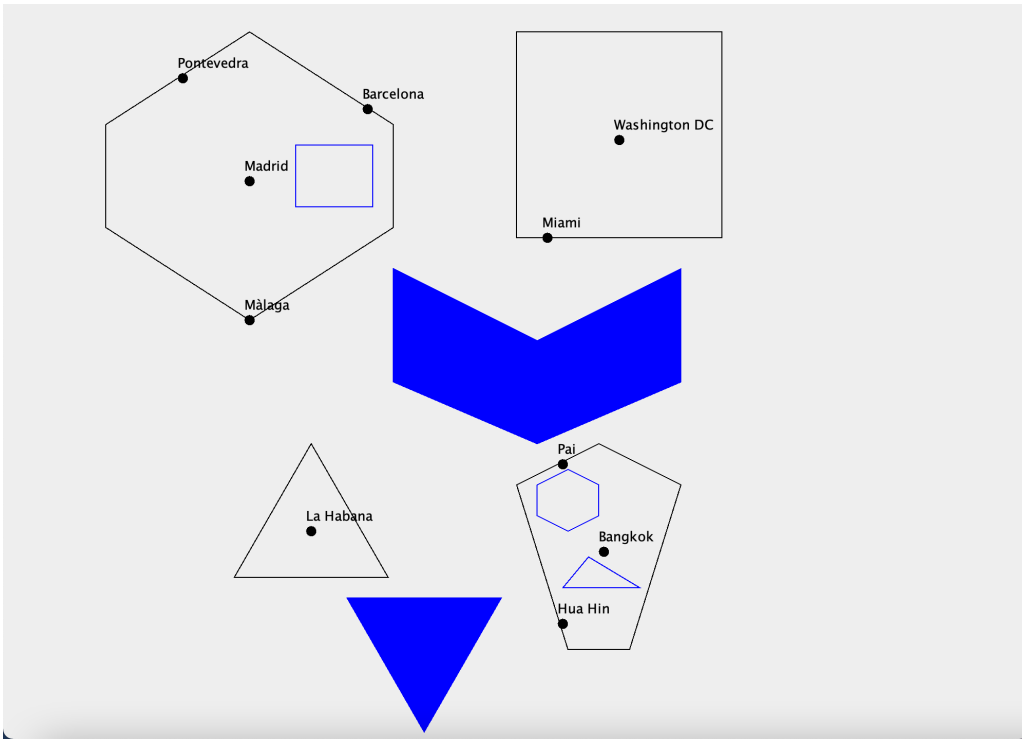
Altrament, volíem comentar que a l'hora d'implementar les classes Lake i Ocean, vam voler "jugar" amb els seus mètodes de dibuix (els draws) i mitjançant les funcions de Graphics setColor() i fillColor(), vam decidir mostrar per pantalla els lakes amb la vora de color blava i els oceans establir-los amb els color de fons blau, per així poder diferenciar-ho. No obstant, quan vam cridar a les diferents funcions de Graphics per poder canviar els colors, ens vam adonar que el color de la regió i les etiquetes de les ciutats també ens canviava. Per aquest motiu, vam decidir incorporar aquestes funcions en els mètodes drawP() i drawGP(), implementats en les classes PolygonalRegion i GeoPoint respectivament, per "obligar" a que tant les regions com els punts geomètrics i les seves etiquetes es mostressin de color negre.

En darrer lloc, volem afegir que, tal i com vam realitzar a la pràctica anterior, volíem demostrar que les funcions getArea() i getTotalArea() funcionaven correctament. Per tant, vam printejar les àrees dels diferents països i dels diferents continents.

Finalment, el resultat del nostre programa apareix de la següent forma:



PRINT DEL NOSTRE MÓN



INTERPRETACIÓ DE LA DISTRIBUCIÓ DEL NOSTRE MÓN