Classes i encapsulació en Java

Definició de classes

A la definició de la classe hem d'especificar la seva declaració (tipus d'accés, identificador, herència) i el cos, que n'és el contingut.

Les classes descendents o derivades hereten les variables i mètodes de les seves ascendents o classes base.

El cos de les classes conté les declaracions i inicialitzacions dels membres de la classe, siguin mètodes, variables de classe o variables d'instància i la descripció de la feina que fan els mètodes.

A la declaració de la classe es poden incloure diversos **modificadors**:

- **public**: accessibles arreu
- **final**: no pot tenir classes derivades
- **abstract**: no se'n poden instanciar objectes. Només és una plantilla per derivar altre classes
- **syncronizable**: els seus mètodes i atributs no són accessibles simultàniament per diversos fils.

Els tipus abstractes de dades (TAD)

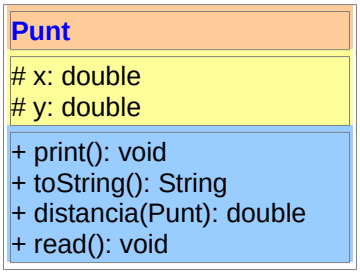
Anem a definir la classe Punt per representar els punts del pla cartesià. Els atributs d'un Punt són les seves dues coordenades (x,y). Necessitem mètodes per llegir aquests atributs (getX(), getY()), mètodes per modificar-los (setX(), setY()), constructors per crear objectes de tipus Punt i, si és el cas, inicialitzar a voluntat els atributs, així com altres mètodes per serialitzar l'objecte (print()), calcular la distància entre dos punts, etc.

Per representar el TAD d'una manera ràpida i visual, utilitzarem la **notació UML**.

Un TAD es descriu amb una caixa dividida en tres zones. A la superior, s'indica el nom de la classe. A la segona (intermitja) es relacionen els atributs. Per últim, a la zona inferior es relacionen els mètodes. Cada identificador es precedeix d'un símbol que indica la seva accessibilitat, d'acord amb el següent conveni:

private (-)	només accés a la classe
protected (#)	accés a la classe, el paquet i les classes derivades
public (+)	accés a la classe, el paquet, les classes derivades i qualsevol altra classe

Per facilitar la lectura de la representació i simplificar-la, en general no hi posarem ni els mètodes constructors ni els mètodes d'accés (accessors de lectura i escriptura).



A continuació teniu la implementació de la classe Punt.

```
/**
 * Punt.java
*/
import java.io.*;
public class Punt
{
     protected double x;  // coordenades (x,y)
     protected double y;
/**
 * Punt(): constructor amb paràmetres
 */
     public Punt(double x, double y)
     {
          this.x = x;
          this.y = y;
     }
/**
 * Punt(): constructor sense paràmetres
 */
     public Punt()
     {
          this.x = 0.0;
          this.y = 0.0;
     }
/**
 * Punt(): constructor de còpia
 */
     public Punt(Punt B)
     {
          this.x = B.getX();
          this.y = B.getY();
     }
/**
 * accessors de lectura i escriptura dels atributs
 */
     public double getX()
     {
          return x;
     }
     public double getY()
     {
          return y;
     }
     public void setX(double x)
     {
          this.x = x;
     }
     public void setY(double y)
     {
          this.y = y;
     }
/**
 * print(): serialitza l'objecte per la sortida estàndard
 */
     public void print()
     {
          System.out.print("("+x+","+y+")");
     }
```

```
/**
 * toString(): serialitza l'objecte en un String
 */
    public String toString()
    {
        return ("("+Double.toString(x)+","+Double.toString(y)+")");
    }
/**
 * distancia(): calcula la distancia al punt B
 */
    public double distancia(Punt B)
    {
        return (Math.hypot(B.getX()-x,B.getY()-y));
    }
/**
 * read(): llegeix Punt des de l'entrada estàndard
 */
    public void read() throws IOException, NumberFormatException
    {
        BufferedReader cin=
            new BufferedReader(new InputStreamReader(System.in));
        setX(Double.parseDouble(cin.readLine()));
        setY(Double.parseDouble(cin.readLine()));
    }
}
```

El mètode constructor principal és el que defineix les dues coordenades. Aquest mètode està sobrecarregat amb un altre sense paràmetres que assigna valors per defecte. Si no es defineix cap constructor, el compilador en genera un per defecte sense paràmetres i sense cos, però en el moment que se'n defineix un, es perd el que es genera per defecte i, per tant, es fa necessari definir-ne un sense paràmetres si és que es necessita. Els constructors no tenen tipus de retorn i tenen el mateix nom que la classe.

Per tal de provar el funcionament de la classe Punt, programem la classe PuntTestMain.

```
/**
 * PuntTestMain.java
 * @description: Test per a la classe Punt
 * per a l'entrada estàndard
 */
import java.io.*;
public class PuntTestMain
{
    public static void main(String[] args)
        throws IOException, NumberFormatException
    {
        BufferedReader cin =
            new BufferedReader(new InputStreamReader(System.in));
        Punt a = new Punt(2,3);
        Punt b = new Punt();
        System.out.print("\na: "); a.print();
        System.out.print("\nb: "); b.print();
        System.out.print("\nEntra b: "); b.read();
        System.out.print("\na: "); a.print();
        System.out.print("\nb: "); b.print();
        System.out.print("\nDistancia entre a i b: "+a.distancia(b));
        System.out.println("\nCoordenades de a: ("+a.getX()+","+a.getY()+")");
        b.setX(4.0); b.setY(2.0);
        System.out.print("\nPunt b modificat: "); b.print();
        Punt c=new Punt(c);
```

```
        System.out.print("\nPunt c: "); c.print();
        System.out.print("\nEntra c: "); c.read();
        System.out.print("\nPunt c: "); c.print();
    }
}
```

Aquesta aplicació il·lustra l'ús dels mètodes de la classe Punt i permet verificar-ne el seu funcionament.

Els atributs i els mètodes poden tenir el modificador **static**. En aquest cas, aquest atribut o mètode es diu "de classe" és compartit per tots els objectes de la classe. De fet, no cal instanciar cap objecte de la classe per tenir-hi accés. En el cas dels atributs static, el seu valor és el mateix per a tots els objectes de la classe. L'accés es pot fer a través de l'identificador d'un objecte qualsevol de la classe i també a través de l'identificador de la classe.

Per exemple, introduïm la propietat de classe

```
public static int numDimensions = 2;
```

Podem accedir des de fora de la classe Punt d'aquestes formes:

```
int a = Punt.numDimensions;   //utilitzem l'dentificador de la classe
// o bé instanciem un Punt...
Punt p = new Punt();
// ... i accedim a la propietat
int b = p.numDimensions;
```

Podem modificar el valor de l'atribut

```
//amb l'identificador de la classe
Punt.numDimensions = 3;
//amb l'identificador d'un objecte
p.numDimensions = 3;
```

Tots els objectes de classe Punt compartiran el nou valor de l'atribut.

L'encapsulació de dades

El principi d'encapsulació de les dades estableix l'ocultació de la implementació de la classe per tal que sigui fàcil d'usar i de canviar, aplicant el model de caixa negra que ofereix serveis. Serveix per implementar el principi d'obertura-tancament, separar les dades i el comportament de l'aplicació i controlar com cada part és usada per la resta de l'aplicació.

Així, l'accés a les propietats de les classes es definirà de la manera més restrictiva possible per tal de controlar de manera precisa l'accés a la informació i la seva modificació. A més, el disseny evitarà la necessitat de conèixer detalls concrets sobre la implementació de la classe, de manera que el programador que utilitza la classe es pugui centrar només en els serveis que proveeix i la manera d'usar-los.

Considerem, com a exemple, la següent definició de la classe Rectangle per representar un rectangle. Cada rectangle vindrà definit per la seva base i la seva altura.

```
/**
 * Rectangle.java
 * TAD Rectangle
 */

public class Rectangle {
    //Propietats o atributs
    public double base;       //base del rectangle (lectura/escriptura)
    public double altura;    //altura del rectangle (lectura/escriptura)
```

```
    /**
     * Contructor per defecte (sense parametres)
     **/
   public Rectangle() {
     base = 0.0;
     altura = 0.0;
   }

    /**
     * Contructor amb inicialització
     **/
   public Rectangle( double base, double altura )  {
     this.base = base;
     this.altura = altura;
   }

   //Comportament (mètodes)
    /** area()
     * @return area del rectangle
     */
   public double area() {
     return ( base*altura );
   }

    /** perimetre()
     * @return perimetre del rectangle
     */
   public double perimetre() {
     return ( 2*(base+altura) );
   }

    /** toString()
     * @return conversió a String del rectangle
     */
   public String toString() {
     return ( "Rectangle: base="+base+" altura="+altura );
   }

    /** equals()
     * @return comparació amb un altre Rectangle
     */
   public boolean equals( Rectangle r ) {
     return ( ((this.base==r.base) && (this.altura==r.altura)) ? true : false );
   }
}
```

Per tal de provar la classe Rectangle, utilitzem la classe RectangleMain.

```
/**
 * RectangleMain.java
 * Aplicació de càlcul amb rectangles
 */
import java.io.*;
public class RectangleMain {

   public RectangleMain() {
   }

   public static void main(String[] args) {
```

```
        Rectangle a = new Rectangle();
        Rectangle b = new Rectangle(5.0, 2.0);
        Rectangle c = new Rectangle(3.0, 4.0);

        System.out.println("Mostrem amb toString()");
        System.out.println("Rectangle a: "+a.toString());
        System.out.println("Rectangle b: "+b.toString());
        System.out.println("Rectangle c: "+c.toString());
         System.out.println("Mostrem perimetres i arees");
        System.out.println("El rectangle a te perimetre="+a.perimetre()+" i
area="+a.area());
        System.out.println("El rectangle b te perimetre="+b.perimetre()+" i
area="+b.area());
        System.out.println("El rectangle c te perimetre="+c.perimetre()+" i
area="+c.area());
        System.out.println("Comparacions");
        if ( a.equals(b) ) System.out.println("Els rectangles a i b són iguals");
        else  System.out.println("Els rectangles a i b són diferents");
         System.out.println("Canviem el rectangle b");
         b.base = -4;
         System.out.println("Rectangle b: "+b.toString());
        System.out.println("El rectangle b te perimetre="+b.perimetre()+" i
area="+b.area());
   }
}
```

Observem que el programa permet introduir dades incorrectes (no vàlides) per als atributs dels objectes. En aquest cas hem posat un valor negatiu per a una de les dimensions d'un rectangle. També es pot fer a través del constructor. Aquest fet és molt perillós. Cal definir de quina manera es pot accedir a l'estat dels objectes (atributs/propietats), en quines condicions i què fer amb els valor no vàlids. Això s'aconsegueix amb el mecanisme de l'encapsulació. Consisteix a tancar l'accés a les propietats de manera directa, i proveir quan sigui necessari mètodes públics d'accés per a la lectura (get) i escriptura (set) dels atributs. A través d'aquests mètodes podem evitar que els objectes quedin en estats inconsistents.

Així, podem encapsular la propietat *base* de tipus *double* definint els mètodes d'accés següents:

```
public double getBase()
{
    return this.base;
}
public void setBase(double base)
{
    this.base = base;
}
```

De moment, encara no hem fet cap control sobre el valor que *setBase*() assigna a la propietat. Suposem que el valor de *base* no ha de ser negatiu. En cap que el valor del paràmetre base que rep *setBase*() sigui invàlid, podem optar per dues possibilitats:

a) Descartar el valor invàlid i posar-hi un valor per defecte.

```
public void setBase(double base)
{
    if (base >= 0) this.base = base;
    else this.base = 0.0;  //valor per defecte
}
```

Aquesta acció és transparent a la classe invocadora. Per tant, no tenim cap avís que el valor era invàlid i que s'ha descartat.

b) Llançar una excepció o error que es pugui capturar (si es vol) des de la classe invocadora

```
public void setBase(double base)
{
    if (base >= 0) this.base = base;
    else throw new RuntimeException();  //llancem RuntimeException
}
```

Les RuntimeException són excepcions de temps d'execució i no cal declarar que la funció *setBase* pot llançar-la (en certes situacions) ni estem obligats a capturar-la a la classe invocadors (tot i que en aquest cas el programa es detendrà).

Més endavant aprofundirem sobre el mecanisme de les excepcions.

El mateix procediment s'ha de fer sobre la propietat *altura*.

Cal també encapsular l'accés a les propietats en el constructor d'inicialització.

```
public Rectangle(double base, double altura)
{
    setBase(base);
    setAltura(altura);
}
```

Com a conclusió, hem d'encapsular l'accés a les dades de les classes i atorgar l'accés més restrictiu possible.

Nivells d’accés

El nivell d’accés a atributs i mètodes depèn del modificador utilitzat en la seva definició i del punt des del qual el volem utilitzar.

Modificador	Classe	Paquet	Subclasse	Resta de classes
Sense modificador	Sí	Sí	No	No
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
private	Sí	No	No	No

En general, convé utilitzar el nivell més restrictiu d’accés.