DESARROLLO DE APLICACIONES WEB

Unidad 8

Programación Orientada a Objetos

1r DAW

IES La Mola de Novelda

Departament d'informàtica

Profesor: Joan Carbonell Picó

ÍNDEX

1.- Introducción....3
2.- POO....3
2.1.- Clases y Objetos....4
2.1.1.- Definición de una clase en Java....6
2.1.2.- Objetos....7
2.1.2.1.- Creación de un objeto....7
2.1.2.2.- Operador new....8
2.1.3.- Atributos (campos)....9
2.1.4.- Métodos....10
2.1.5.- Ejemplos atributos y métodos estáticos....12
2.1.6.- Sobrecarga de métodos....13
2.1.7.- Ámbito de la variables y atributos....13
2.1.8.- Ocultación de atributos....14
2.1.9.- Objeto *this*....14
2.1.10.- Constructores....15
2.1.11.- *this*()....16
2.1.12.- Constructor de copia....17
2.2.- Paquetes....18
2.3.- Modificadores de acceso....18
2.3.1.- Modificadores de acceso para clases....18
2.3.2.- Modificadores de aceso para miembros....21
2.3.3.- Métodos getter /setter....23
3.- Ejercicios - 1....25
4.- Ejercicios - 2....25
5.- Ejercicios - 3....28

Unidad 8: Programación Orientada a Objetos

1.- INTRODUCCIÓN

Hasta ahora, hemos utilizado el paradigma de programación llamado **Programación estructurada**, que emplea las estructuras de control (condicionales y bucles), junto a datos y funciones. Una de sus principales características es que no existe un vínculo fuerte entre funciones y datos. Para desarrollar un proyecto utilizando este paradigma debemos:

1. Identificar el problema a resolver.
2. Descomponerlo en diferentes módulos o funciones simples.
3. Combinar los módulos simples para resolver el problema complejo.

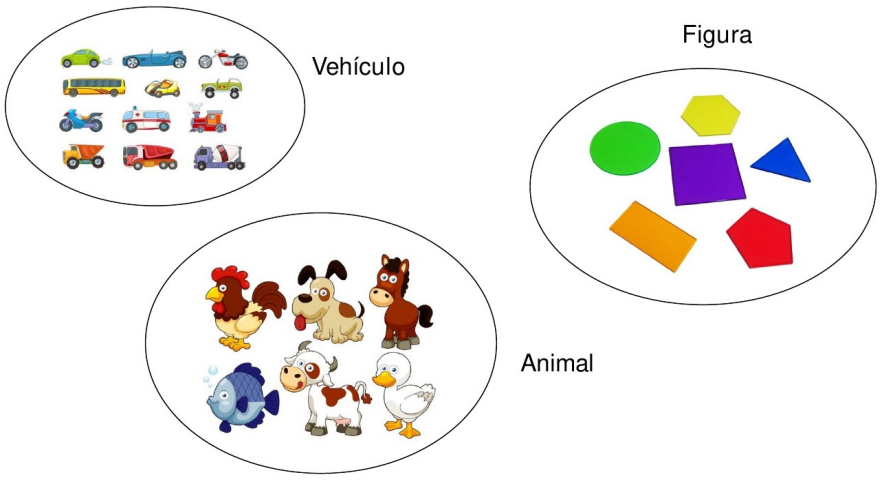
En cambio, con el paradigma de programación **Programación Orientado a Objetos** (**POO**) obtenemos nuevas herramientas con las cuales podemos afrontar problemas más complejos. A diferencia del anterior paradigma, para desarrollar proyectos utilizando la POO debemos:

1. Identificar los elementos que van a formar parte del sistema (clases)
2. Definir la información que se almacena de cada elemento (atributos) y las operaciones que va a poder realizar (métodos)
3. Establecer las conexiones o interrelaciones entre elementos

Como podemos ver, con la POO no pensamos en la funcionalidad de una aplicación o proyecto si no en los elementos que formaran parte de él y de sus iteraciones.

2.- POO

La POO se inspira en la abstracción del mundo real, en la que los objetos se clasifican en grupos. Por ejemplo:



Los seres humanos percibimos el mundo como si estuviera formado por objetos: mesas, sillas, computadoras, coches, cuentas bancarias, etc, donde consciente o inconscientemente tendemos a organizarlos, clasificarlos, relacionarlos entre si, y hasta extraer las características más importantes dependiendo de lo que queramos hacer con ellas.

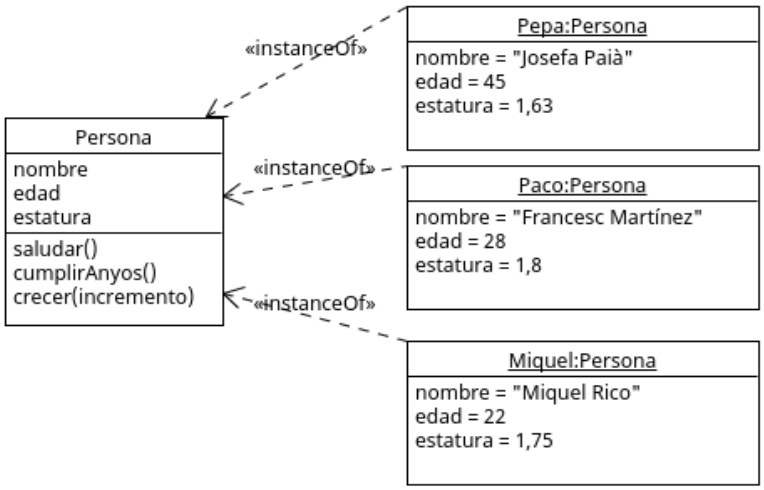
Podemos decir que la POO es un estilo de programación, donde todos los elementos que forman parte del problema se conciben como objetos, definiendo cuales son sus atributos y comportamiento, como se relacionan entre sí y como están organizados. Por lo tanto, la estructura Interna de un Objeto estará formada por:

- **Atributos**: Define el estado del objeto
- **Métodos**: Define el comportamiento del objeto

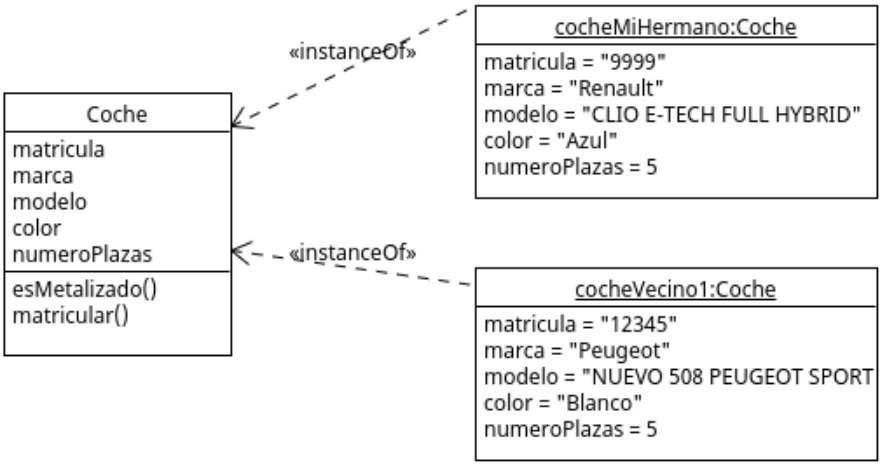
2.1.-CLASES Y OBJETOS

Una **clase** describe un grupo de objetos que comparten propiedades y métodos comunes. Por lo tanto, las clases son plantillas que categorizan los diferentes elementos de la aplicación. En cambio, un **objeto** es una instancia a una clase, es decir, una concreción o representación de esa clase. Ejemplos:

- Pepa, Paco y Miquel pertenecen al mismo grupo ya que los tres son personas. Pepa es una persona, Paco es una persona y Miquel es una persona. Por lo tanto, los tres pertenecen al grupo personas. En el argot de la POO, a cada uno de estos grupos se le llama **clase**. En cambio. Pepa, Paco y Miquel son **objetos** (**instancias**) ya que cada uno de ellos es una concreción de la clase persona.



- Mi coche, el coche de mi hermano y el coche de cada uno de mis vecinos tienen algo en común, que todos son coches. Por lo tanto podemos decir que todos ellos son instancias de la clase coche, es decir, objetos de la clase coche. Cada uno de esos coches son objetos palpables ya que son concreciones de la clase coche. La palabra coche define algo genérico, es una abstracción, no es un coche concreto sino que hace referencia a unos elementos que tienen una serie de propiedades en común: matrícula, marca, modelo, color, etc..; este conjunto de propiedades se denominan **atributos** o **variables de instancia**.



En los dos casos podemos ver que los **atributos** son todo aquello que identifica y que es propio de una persona o de un coche, mientras que los **métodos** son las acciones que podemos hacer con un coche o con una persona.

En Java, los nombres de la clases se escriben con la primera letra en mayúscula mientras que las instancias comienzan con una letra en minúscula.

Definiremos cada clase en un fichero con el mismo nombre y extensión .java. Por ejemplo, la definición de la clase Coche debe estar contenida en un fichero de nombre Coche.java

2.1.1.- Definición de una clase en Java

A la hora de crear una clase deberemos de definir un conjunto de propiedades (atributos) y comportamientos (métodos) y la sintaxis utilizada en Java es:

```
[modificadores] class NombreClase{
        //definición de los atributos
        [modificadores] tipo atributo1;
        [modificadores] tipo atributo2;
        .....
        //definición de los métodos
        [modificadores] tipo nombreMetodo (parámetros){
                cuerpo del método
        }
        ......
}
```

- Ejemplo: Vamos a implementar en Java la clase Persona descrita anteriormente en notación UML. Por el momento, no pondremos el modificador *static* delante de los métodos. Crearemos un nuevo proyecto (Unidad-8) y en él crearemos las dos siguientes clases:

```java
/**
 * Persona.java
 * Definición de la clase Persona
 * @author Joan
 */
2 usages
public class Persona {
    2 usages
    String nombre;
    2 usages
    byte edad;
    1 usage
    double estatura;

    1 usage
    void saludar(){
        System.out.println("Hola. Mi nombre es "+nombre);
        System.out.println("Encantado de conocerte");
    }
    1 usage
    void cumplirAnyos(){
        edad++;
    }
    void crecer(double incremento){
        estatura += incremento;
    }
}
```

```java
/**
 * PruebaPersona.java
 * Programa que prueba la clase Persona
 * @author Joan
 */
2 usages
public class PruebaPersona {
    public static void main(String[] args) {
      PruebaPersona programa = new PruebaPersona();
      programa.start();
    }
    1 usage
    public static void start(){
        Persona p;
        p = new Persona();
        p.edad = 18;
        p.nombre = "Maria";
        p.saludar();
        p.cumplirAnyos();
    }
}
```

A partir de ahora los objetos de tipo Persona pueden invocar sus métodos utilizando un punto (.), al igual que se hace con los atributos.

Tanto a los atributos como a los métodos de una clase se les llama de forma genérica **miembros**. De esta forma, al hablar de miembros de una clase, hacemos referencia a los elementos declarados en su definición, ya sean atributos o métodos.

Los métodos de una clase tienen acceso a las siguientes variables: variables locales declaradas dentro del método, parámetros de entrada y atributos de la clase.

2.1.2.- Objetos

Los elementos que pertenecen a una clase se denominan **instancias** u **objetos**. Cada uno de ellos tiene sus propios valores de los atributos definidos en la clase.

Ejemplo: Supongamos que una clase son los planos para construir una casa. Estos planos se pueden utilizar en repetidas ocasiones, siendo cada una de las casa construidas, casas reales, un objeto de dicha clase. Todas la casas construidas (objetos) tendrán la misma distribución ya que utilizamos el mismo plano (clase). Sin embargo, cada una de las casas (objetos) tendrán distintas particularidades: distinto color de fachada, distinto modelo de puertas, etc.

En el anterior ejemplo, Pepa, Paco i Miquel son instancias u objetos de la clase Persona.

2.1.2.1.- Creación de un objeto

La sintaxis para crear un objeto de una clase es la siguiente:

```
NombreClase  nombreObjeto;
nombreObjeto  =  new NombreClase()
```

Normalmente se utilizan las dos líneas juntas:

```
NombreClase  nombreObjeto  =  new NombreClase()
```

Para entender como se crean los objetos, a diferencia de como se creaban las variables de tipo primitivo, vamos a fijarnos en el siguiente ejemplo:

Con tipos primitivos

```
int var;
var = 23;
```

O bien:

```
int var = 23;
```

Con clases

Dos formas de crear un objeto:

- ```
  MiClaseCoche cocheDemo1;
  cocheDemo1 = new MiClaseCoche();
  ```
- ```
  MiClaseCoche cocheDemo1 = new MiClaseCoche();
  ```
- ```
  cocheDemo1.marca = "Kia";
  ```

- Con `int var;` estamos declarando la variable **var** como entera. Eso significa que se reserva un espacio de 32b (4B) en la memoria principal. A continuación `var = 23;`, asignamos el número 23 a la variable **var**, es decir, se almacena en esos 32b el número 23. La variable que existe y se llama **var**, tiene ahora el valor 23.

- Con la instrucción `MiClaseCoche cocheDemo1;` se declara la variable **cocheDemo1** del tipo
  ```

MiClaseCoche. Con la instrucción **cocheDemo1** = new MiClaseCoche(); estamos creando una instancia u objeto y asignamos su referencia a la variable **cocheDemo1**.

El operador new, primero busca en memoria un hueco disponible donde construir el objeto. Dependiendo de su tamaño (depende del tipo y la cantidad de atributos que tenga la clase) ocupará cierto número consecutivos de bloques de memoria (bytes). Por último, devuelve la referencia del objeto creado, que se le asigna a la variable **cocheDemo1**.

Por último, la instrucción **cocheDemo1.**marca **= "Kia";** asigna el valor **"Kia"** al atributo marca. Estamos guardando valores para el objeto **cocheDemo1**, que en realidad, es dar valor a los atributos del objeto. Una vez tenemos un objeto, podemos accede a sus atributos mediante un punto (**.**) a continuación de la variable que referencia a ese objeto.

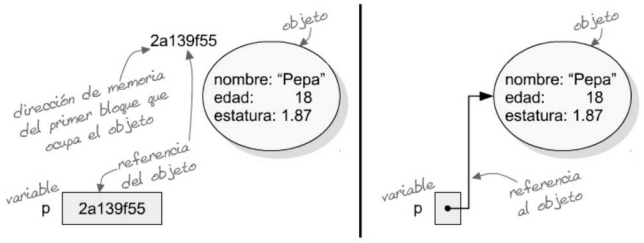
CONCLUSIÓN: La diferencia entre una variable de tipo primitivo y una variable de tipo Clase es que, mientras una variable de tipo primitivo almacena directamente un valor, una variables del tipo clase almacena la referencia de un objeto.

2.1.2.2.- Operador new

Como hemos visto anteriormente, la forma de crear objetos es mediante el operador new, el cual devuelve la referencia del objeto creado en memoria. Para entender mejor su funcionamiento, veamos los siguientes ejemplos:

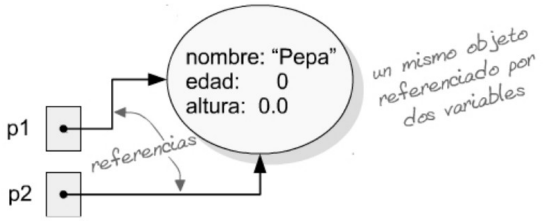
```
Persona p;
p = new Persona();
p.nombre = "Pepa";
p.edad = 18;
p.estatura = 1,87;
```

p es una variable del tipo **Persona** que referencia al objeto **Persona** creado con el operador **new**. Podemos representarlo gráficamente de dos maneras:




```
Persona p1, p2;
p1 = new Persona(); // p1 referencia al objeto creado
p2= p1; // p2 referencia al mismo objeto que p1
p2.nombre = "Pepa"; // es equivalente a p1.nombre
```

Ahora podemos acceder al objeto de dos maneras: mediante **p1** o mediante **p2**. En ambos casos estamos referenciando al mismo objeto:



A partir de los ejemplos podemos ver que una vez se ha creado una variable de referencia, al principio ésta no referencia a ningún objeto (**Persona p**;). En estos casos, se inicializan por defecto a **null**. Con el valor literal **null** hacemos referencia a ningún bloque de memoria. Ejemplo:

```
Persona p;
p.nombre = "Pepa";
```

Estamos accediendo a los miembros de una referencia nula, lo cual produce el error: *Null pointer exception*. Su significado es que estamos intentando acceder a los atributos de un objeto nulo

También se puede dar el caso de que un objeto no esté referenciado por ninguna variable de referencia. Esto sucede cuando se crea un objeto y no se le asigna a ninguna variable (no tiene mucho sentido hacerlo):

```
new Persona();
```

No tiene mucho sentido ejecutar esta instrucción sin la asignación a una variable de referencia.

O también, si asignamos **null** a una variable que ya contenía una referencia a un objeto:

```
Persona p = new Persona();
p = null;
```

El objeto se queda sin referenciar, ya no podemos acceder a él y está ocupando memoria.

Si esos dos comportamientos se repiten con mucha frecuencia, es posible que se agote toda la memoria libre disponible, lo que impedirá el normal funcionamiento del ordenador. Para evitar este problema, Java dispone del mecanismo llamado recolector de basura (*garbage collection*), que se ejecuta periódicamente de forma transparente al usuario, y que se encarga de comprobar, uno a uno, todos los objetos de la memoria. Si algunos de ellos no estuviera referenciado por nadie, lo cual implica la imposibilidad de utilizarlo, se destruye, liberando la memoria que ocupa.

2.1.3.- Atributos (campos)

Los objetos almacenan la información en sus atributos o campos, los cuales son el conjunto de características que comparten los objetos de una clase.

Existen dos tipos de atributos o campos:

- Atributo o campo de instancia: Hay una copia de un atributo o campo de instancia por cada objeto de la clase. El atributo o campo de instancia es accesible a través del objeto al que pertenece.
- Atributo o campo de clase (***static***): Hay **una única copia de un atributo** o campo *static* en el sistema (equivale a lo que en otros lenguajes es una variable global). El campo *static* es accesible a través de la clase (sin necesidad de instanciar la clase).

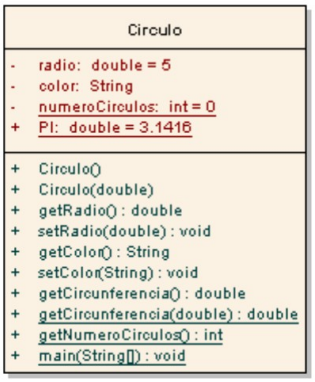
Ejemplo:

Atributos (campos)

```
class Circulo {
    // atributos
    double radio = 5;
    String color;
    static int numeroCirculos = 0;
    static final double PI = 3.1416;
    // métodos
    // constructores
    // main( )
}
```

radio y color son variables de instancia, hay una copia de ellas por cada objeto Circulo

numeroCirculos y PI son variables static, están sólo una vez en memoria; PI además es constante (final): no puede modificarse



Para acceder a los atributos o campos diferenciamos si son de instancia o de clase:

- Acceso a variables de instancia: se utiliza la sintaxis "**objeto.**".

```
Circulo c1 = new Circulo();
c1.radio = 5;
c1.color = "rojo";
```

Recordar que si c1 es null, se genera una excepción NullPointerException

- Acceso a variables *static*: se utiliza la sintaxis "**clase.**".

```
Circulo.numeroCirculos++;
System.out.println(Circulo.PI);
```

2.1.4.- Métodos

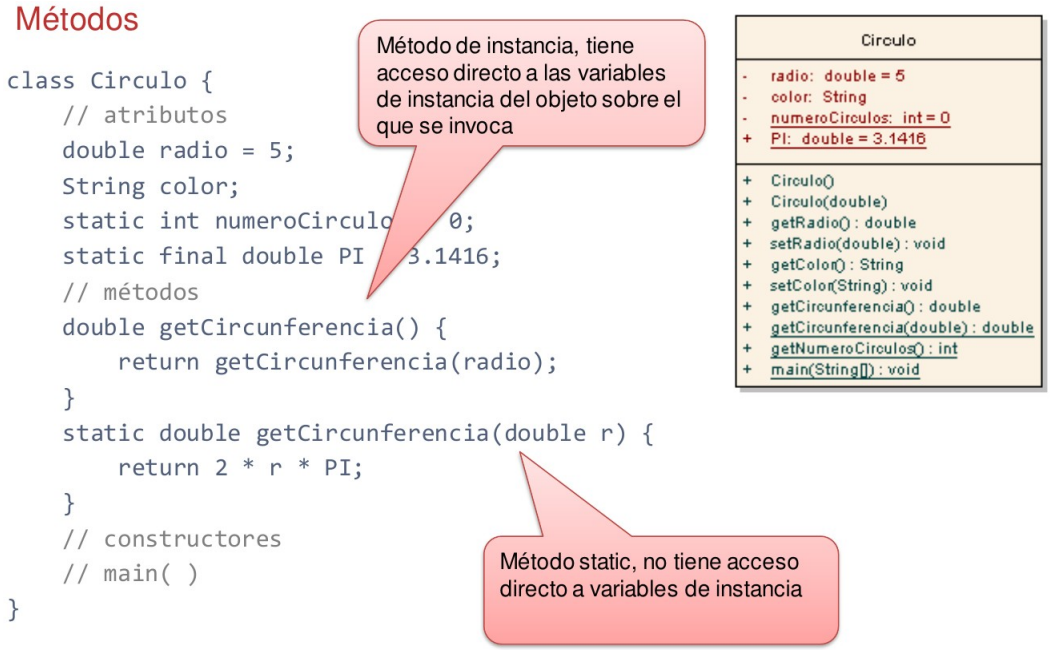
Hemos declarado clases con atributos pero, como hemos visto, las clases también disponen de comportamientos. En la POO, a los comportamientos u operaciones que pueden realizar los objetos de una clase se les denomina **métodos**. Los métodos son la acciones asociadas a una clase y se definen dentro del cuerpo de la clase y se suelen colocar a continuación de los atributos. Su sintaxis es la siguiente:

```
[static] <tipo retorno> <nombre método> (<tipo> parámetro1,
...)
{
    // cuerpo del método
    return <valor de retorno>;
}
```

Los métodos contendrán las instrucciones que operan sobre los datos de una objeto para obtener resultados. Pueden tener cero o más parámetros y pueden retornar una valor o ser declarados como *void* para indicar que no retornan ningún valor. Los métodos pueden ser de instancia o de clase (*static*):

- Un método de instancia se invoca sobre un objeto de la clase, al cual tiene acceso mediante la palabra this (y sus variables de instancia son accesibles de manera directa)
- Un método de clase (*static*) no opera sobre un objeto de la clase, y la palabra this no es válida en su interior.

Ejemplo:



Para acceder tanto a los atributos como a los métodos diferenciaremos si son de instancia o de clase:

- Acceso a atributos y métodos de instancia: se utiliza la sintaxis "**objeto.**".

```
Circulo c1 = new Circulo();
c1.radio = 5;
c1.color = "rojo";
double d = c1.getCircunferencia();
```

Si c1 es null, se genera una excepción NullPointerException

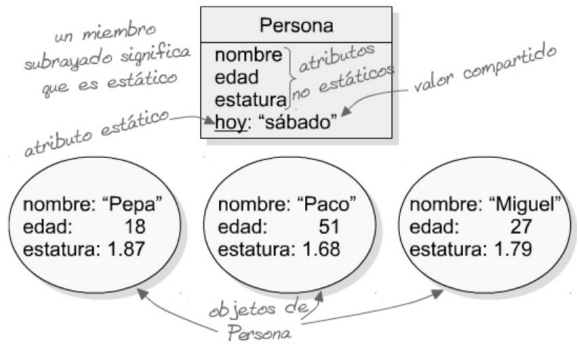
- Acceso a atributos y métodos *static*: se utiliza la sintaxis "**clase.**".

```
Circulo.numeroCirculo++;
int n = Circulo.getNumeroCirculos();
System.out.println(Circulo.PI);
```

2.1.5.- Ejemplos atributos y métodos estáticos

Un atributo estático o de clase es aquel del que no existe una copia en cada objeto. Todos los objetos de una misma clase comparten su valor. Veamos el siguiente ejemplo:

- Queremos añadir a la clase **Persona** el atributo hoy que nos indique qué día de la semana es hoy. Evidentemente, si hoy es lunes, será lunes para todo el mundo; e igualmente si es martes, será martes para todos. Por lo tanto, el valor del atributo hoy será compartido por todos los objetos. La siguiente figura representa este concepto:




```
class Persona{
        . . .
        static String hoy = "Dilluns";
        . . .
}

Persona p = new Persona();
System.out.println(Persona.hoy);
Persona.hoy = "Diumenge";
System.out.println(p.hoy);
```

- De igual manera, podemos declarar métodos estáticos. Son aquellos que no requieren de ningún objeto para ejecutarse y, por tanto, no pueden utilizar ningún atributo que no sea estático. En el caso de que un método estático intente utilizar un atributo no estático se producirá un error.

```
static void hoyEs(int dia){
        switch(dia){
                case 1: hoy="Dilluns";
                        break;
                case 2: hoy="Dimarts";
                        break;
                        . . .
                case 7: hoy="Diumenge";
                        break;
        }
}
```

La forma de invocar un método estático es, igual que con los atributos estáticos, mediante el nombre de la clase o cualquier objeto.

Si queremos actualizar el dia a "Dimarts", cualquiera de las dos instrucciones seria válida

```
Persona.hoyEs(2);

p.hoyES(2);
```

2.1.6.- Sobrecarga de métodos

De la misma manera que estudiamos en la unidad de la programación modular, los métodos de una clase pueden tener el mismo nombre pero diferentes parámetros. Cuando se invoca un método, el compilador compara el número y tipo de los parámetros y determina que método debe invocar o ejecutar. La cuestión es que la firma de los métodos que tienen el mismo nombre sean diferentes. La firma de una método es:

Firma (signature) = nombre del método + lista de parámetros.

Ejemplo:

```
class Cuenta{
        public void depositar(double monto){
                this.depositar(monto,"$");
        }
        public void depositar(double monto,String moneda){
                //procesa el depósito
        }
}
```

- Decimos que un **método** está **sobrecargado** cuando admite más de una combinación de tipos y/o cantidades de argumentos. Esto se logra escribiendo el método tantas veces como tantas combinaciones diferentes queremos que el método pueda admitir.

2.1.7.- Ámbito de la variables y atributos

El ámbito de una variable define en qué lugar puede utilizarse. Éste coincide con el bloque en el que se declara la variable, como hemos visto en las unidades anteriores. Puede ser:

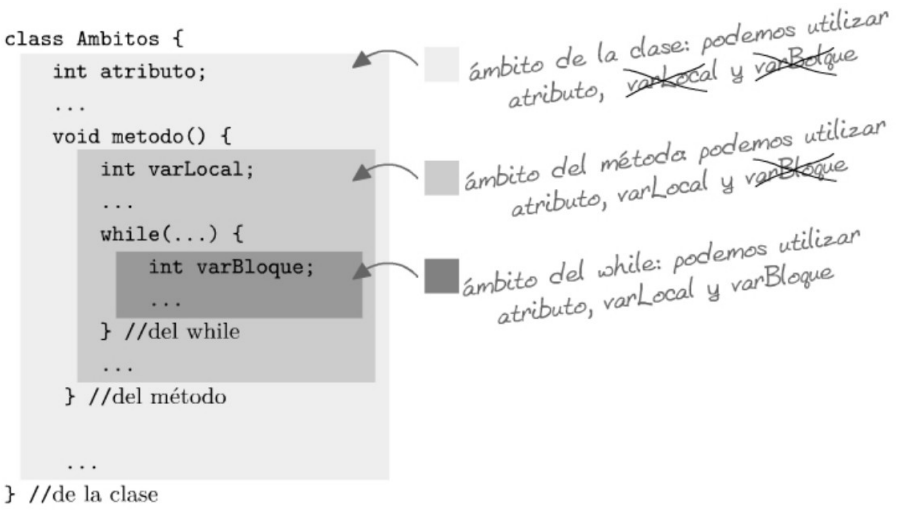
- Bloque de una estructura de control (*if, if-else, swtich, while, do-while, for*). Las variables declaradas en este ámbito se denominan variables de bloque.

- Una función o método. A las variables declaradas aquí se conocen como variables locales.

Con la POO, aparece un nuevo ámbito:

- La clase. Cualquier miembro, atributo o método definido en una clase podrá ser utilizado en cualquier lugar de la misma. A la variables declarada en su definición se les llama atributos.

Hay que tener en cuenta que un ámbito puede contener otros, formando una estructura jerárquica. Ejemplo:



En el código de la figura podemos ver el ámbito de tres variables, donde **atributo** puede utilizarse en cualquier lugar de la clase, **varLocal** en cualquier lugar dentro del método en el que se declara; por último, **varBloque** puede usarse solo dentro del bloque de instrucciones de **while**.

2.1.8.- Ocultación de atributos

Dos variables declaradas en ámbitos anidados no pueden tener el mismo identificador, ya que esto genera un error. Existe una excepción, cuando una variable local en un método tiene el mismo identificador que un atributo de la clase. En este caso, la variable local tiene prioridad sobre el atributo, lo que provoca que al utilizar el identificador se acceda a la variable local y no al atributo. En la jerga de la POO se dice que la variable local oculta al atributo. Ejemplo:

```
public class Ambito{
        int edad; // atributo entero

        void metodo(){
                double edad; //variable local. Oculta el atributo edad (entero)
                edad = 8.2; //variable local, no el atributo de la clase
                . . .
        }
}
```

2.1.9.- Objeto *this*

De la misma manera que nos podemos referirnos a nosotros mismos como "*yo*", aunque tengamos un nombre que los demás utilizan para identificarnos, las clases se refieren a si mismas como ***this***, que es una referencia al objeto actual y funciona como una especie de "*yo*" para clases. Al escribir ***this*** en el ámbito de una clase se interpreta como la propia clase, y permite acceder a los atributos aunque se encuentren ocultos. Ejemplo:

```
public class Ambito{
        int edad; // atributo entero

        void metodo(){
                double edad; //variable local. Oculta el atributo edad (entero)
                edad = 8.2; //variable local, no el atributo de la clase
                this.edad = 30; //atributo de la clase
                . . .
        }
}
```

La palabra ***this*** representa una referencia al objeto sobre el cual se invoca un método de instancia.

2.1.10.- Constructores

En la creación de un objeto, los atributos a los que no se les asigna un valor en su declaración se inicializan por defecto, dependiendo de su tipo de la siguiente manera: cero para valores numéricos, *null* para referencias y **false** para booleanos.

Antes de utilizar un objeto tendremos que asignar valores a cada uno de sus atributos. Por ejemplo:

```
Persona p = new Persona(); //creamos el objeto
p.nombre = “Maria”;
p.edad = 23;
p.estatura = 1,20;
```

Si no queremos trabajar con los valores por defecto, la asignación de valores será necesario cada vez que creamos un objeto. El operador *new* facilita esta tarea mediante los **constructores**.

Un **constructor** es un método especial que debe tener el mismo nombre que la clase, se define sin tipo devuelto (ni siquiera *void*), y se ejecuta inmediatamente después de crear el objeto. Por lo tanto, los constructores son métodos especiales invocados al instanciar una clase.

El principal cometido de un constructor es asignar valores a los atributos, aunque también se pueden utilizar para ejecutar cualquier código necesario: crear tablas, mostrar cualquier tipo de información, crear otros objetos que necesitemos, etc.

Al constructor, como cualquier otro método, se le pueden pasar parámetros y se puede sobrecargar. Veamos el siguiente ejemplo:

- Vamos a implementar un constructor para **Persona**, que asigne los valores iniciales de sus atributos: nombre, edad y estatura. Dentro de la definición de la clase escribiremos el siguiente constructor:

```
Persona(String nombre, byte edad, double estatura){
    this.nombre = nombre;
    this.edad = edad;
    this.estatura = estatura;
 }
```

- Los valores de los parámetros de entrada se especifican al crear el objeto con *new* en la llamada al constructor. Si deseamos crear un objeto Persona con los datos que hemos utilizado anteriormente sería:

```
Persona p = new Persona("Claudia",8,1.2);
```
Creación del objeto y lo inicializamos con el constructor

- Si queremos sobrecargar un método tendremos que asegurarnos de que se puedan distinguir entre ellos mediante el número y/o tipo de parámetros de entrada. La sobrecarga de constructores es útil cuando necesitamos inicializar objetos de varias formas. Por ejemplo:

```
  //constuctor sobrecargado que sólo asigna el nombre
 Persona(String nombre){
    this.nombre = nombre;
    estatura = 1.0; //valor arbitrario para la estatura
    //al no asignar la edad se iniciliza por defecto a 0
 }
 // constructor sobrecargado por defecto
 Persona() {


 }
```

- Ahora disponemos de tres constructores que se utilizan de la siguiente manera:

```
Persona p = new Persona("Claudia",8,1.2);
Persona b = new Persona("Joan");
Persona c = new Persona();
```

2.1.11.- *this*()

Cuando la clase dispone de un conjunto de constructores sobrecargados, es posible que un constructor invoque a otro y así reutilizar su funcionalidad. El problema es que los constructores no se pueden invocar por su nombre directamente, sino mediante el constructor genérico ***this()***, que es como se denomina a los constructores desde dentro de su clase. La forma de distinguir los distintos constructores, al igual que en cualquier método sobrecargado, es mediante el número y el tipo de los

parámetros de entrada. Veamos el siguiente ejemplo:

```
// constructor que asigna valores a todos los atributos
  Persona(String nombre, int edad, double estatura){
    this.nombre = nombre;
    this.edad = edad;
    this.estatura = estatura;
  }

  //constuctor sobrecargado que sólo asigna el nombre
  Persona(String nombre){
    this(nombre,0,1.0); //invoca al primer constructor
    //la edad se pone a 0 y la estatura a 1.0
  }
```

Hay que tener presente que en caso de utilizar ***this()***, tiene que ser la primera instrucción de un constructor; en otro caso producirá error.

2.1.12.- Constructor de copia

Una forma de iniciar un objeto podría ser mediante la copia de los valores de los atributos de otro objeto ya existente. Imagina que necesitas varios objetos iguales (con los mismos valores en sus atributos) y que ya tienes uno de ellos perfectamente configurado (sus atributos contienen los valores que tú necesitas). Estaría bien disponer de un constructor que hiciera copias idénticas de ese objeto.

Durante el proceso de creación de un objeto puedes generar objetos exactamente iguales (basados en la misma clase) que se distinguirán posteriormente porque podrán tener estados distintos (valores diferentes en los atributos). La idea es poder decirle a la clase que además de generar un objeto nuevo, que lo haga con los mismos valores que tenga otro objeto ya existente. Es decir, algo así como si pudieras clonar el objeto tantas veces como te haga falta. A este tipo de mecanismo se le suele llamar constructor copia o constructor de copia.

Un constructor copia es un método constructor como los que ya has utilizado pero con la particularidad de que recibe como parámetro una referencia al objeto cuyo contenido se desea copiar. Este método revisa cada uno de los atributos del objeto recibido como parámetro y se copian todos sus valores en los atributos del objeto que se está creando en ese momento en el método constructor.

Veamos el siguiente ejemplo:

Los dos constructores de copia hacen lo mismo

```
Persona(Persona p){
    this(p.nombre,p.edad,p.estatura);
 }
```

```
Persona(Persona p){
    this.nombre = p.nombre;
    this.edad = p.edad;
    this.estatura = p.estatura;
 }
```

2.2.-PAQUETES

En Java es importante controlar la accesibilidad de unas clases desde otras por razones de seguridad y eficiencia. Esto se consigue por medio de los paquetes, que son contenedores que nos permiten guardar las clases en compartimientos separados, de modo que podamos decidir, a través de la importación, qué clases son visibles desde una clase que estemos implementando.

Los **paquetes** se organizan de forma jerárquica, del mismo modo que las carpetas, y puede haber paquetes que contienen paquetes. Un archivo fuente de Java es una archivo de texto con extensión .java, que se guarda en un paquete y que contiene los siguientes elementos:

- Una sentencia donde se especifica el paquete al que pertenece, que empieza con la palabra clave ***package*** seguida del nombre del paquete.
- Una serie opcional de sentencias de importación, que empiezan con la palabra ***import*** que nos permiten importar las clases definidas en otros paquetes.
- La definición de una o más clases, de la cuales solo una puede ser declarada pública (por medio del modificador de acceso *public*). De todas formas, es recomendable que, en cada archivo fuente se defina una sola clase, que debe tener el mismo nombre que el archivo.

No debemos olvidar que se importan las clases, no los paquetes. Si queremos importar todas las clases públicas de un paquete en una sola sentencia de importación, se usa el asterisco (*****).

2.3.-MODIFICADORES DE ACCESO

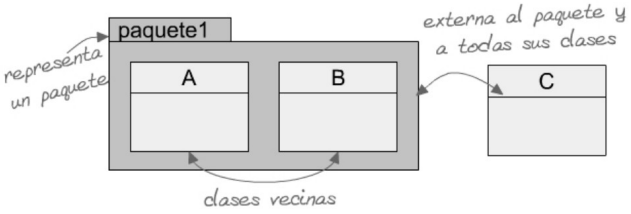
Una clase será visible por otra dependiendo de si se ubican en el mismo paquete y de los modificadores de acceso que utilice. Estos alteran su visibilidad, permitiendo que se muestre u oculte.

De igual manera que podemos modificar la visibilidad entre clases, es posible modificar la visibilidad a nivel de miembros, es decir, qué atributos y métodos son visibles para otras clases

2.3.1.- Modificadores de acceso para clases

Debido a la estructura de clases, organizadas en paquetes, que utiliza Java, dos clases cualesquiera pueden definirse como:

- **Clases vecinas**: cuando ambas están definidas dentro del mismo paquete. Todas las clases que pertenecen a un paquete son vecinas entre sí.
- **Clases externas**: en el caso de que se encuentren definidas en paquetes distintos. Además, una clase definida fuera de una paquete dado, no solo es externa a todas sus clases, sino al propio paquete.



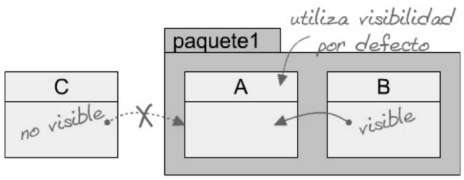
La clases siguen el lema si lo ves, puedes utilizarlo. Utilizar una clase significa crear objetos de ella o acceder a sus miembros estáticos.

Visibilidad por defecto

Cuando definimos una clase sin utilizar ningún modificador de acceso,

```
package paquete1;

class A{ //sin modificador de acceso
   . . .
}
```



se dice que utiliza visibilidad por defecto, que hace que sólo sea visible por sus clases vecinas. En nuestro caso, A es visible por B, pero no será visible por C.

Visibilidad total

La clase A es invisible para todas las clases externas. Si queremos que la clase A sea visible desde C, utilizaremos el modificador de acceso ***public***, que produce el siguiente efecto: la clase marcada como pública será visible, además, desde cualquier clase externa con una sentencia de importación. El modificador ***public*** proporciona visibilidad total a la clase.

```
package paquete1;

public class A{
   . . .
}
```

Hemos redefinido la clase A para que utilice visibilidad total

A partir de ahora, cualquier clase, vecina o externa, puede crear objetos o acceder a los miembros estáticos de la clase A. Lo único que necesita una clase externa, como C, para utilizar A es importarla.

```
import paquete1.A;
class C{
   . . .
}
```

Ahora C puede utilizar la clase A

No debemos olvidar que se importan las clases, no los paquetes. Si queremos importar todas las clases públicas de un paquete en una sola sentencia de importación, se usa el asterisco.

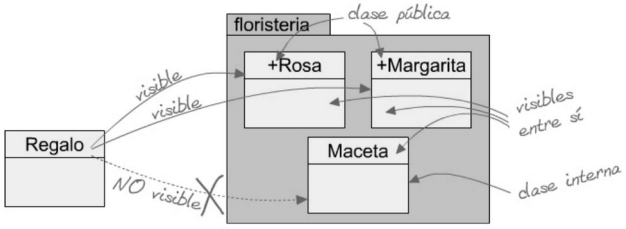
```
import paquete1.*;
class C{
   . . .
}
```

Ahora C puede utilizar cualquier clase visible en paquete1

Resumen de la visibilidad entre clases:

	Visible desde...	
	clases vecinas	clases externas
sin modificador	✓	
`public`	✓	✓

- **Ejemplo**: Necesitamos modelar el funcionamiento de una floristería, donde se venden rosas y margaritas, cortadas o plantadas en una maceta. En la floristería nunca se venden macetas vacías → Una posible solución es definir las clases **Rosa**, **Margarita** y **Maceta** dentro del paquete floristería. Las dos primeras serán visibles desde fuera del paquete; y la clase Maceta será de uso interno al paquete. Cualquier clase externa, como la clase Regalo, podrá utilizar la clase Rosa o la clase Margarita (que vendrán cortadas o en una maceta) pero no será posible regalar una maceta vacía. En la siguiente figura se representa esta posible solución:



I el código que modela esta comportamiento es:

```
package floristeria;

public class Rosa{
   . . .
}
```

```
package floristeria;

public class Margarita{
   . . .
}
```

```
package floristeria;

class Maceta{
   . . .
}
```

2.3.2.- Modificadores de aceso para miembros

También podemos regular la visibilidad de los miembros de una clase. Que un atributo sea visible significa que podemos acceder a él tanto para leer como para modificarlo. Que un método se visible significa que puede invocarse.

Para que un miembro ser visible, es indispensable que su clase también lo sea. Es evidente que si no podemos acceder a una clase, tampoco lo podremos hacer a sus miembros.

Cualquier miembro es siempre visible dentro de su propia clase, indistintamente del modificador de acceso que utilicemos. Por lo tanto, dentro de la definición de una clase siempre tendremos acceso a todos los atributos y podremos invocar cualquiera de sus métodos.

```
public class A{ //clase pública
        int a; //su ámbito es toda la clase: a es accesible desde cualquier lugar de A
        . . .
}
```

Visibilidad por defecto

Cuando queramos acceder a miembros de otra clase hay diversos grados de visibilidad. La visibilidad por defecto es aquella que se aplica a miembros declarados sin ningún modificador de acceso, como el atributo **a** en el código anterior.

La visibilidad por defecto implica que un miembro es visible desde las clases vecinas, pero invisible desde clases externas. En el ejemplo anterior, el atributo **a** será:

- Visible por clases vecinas, lo que les permite acceder tanto a la clase A como acceder al atributo **a**. Acceder a una clase significa utilizar sus miembros visibles, incluidos los constructores que permiten crear objetos.
- Invisible desde clases externas. Cualquier clase externa podrá acceder a la clase **A**, pero no al atributo **a**.

Hay que tener en cuenta que la clase **A** se ha definido ***public***, lo que permite que sea visible desde clases externas. Si **A** no fuera visible desde el exterior, tampoco lo serían sus miembros, sin importar el modificador utilizado en su declaración.

Modificador de acceso *private* y *public*

Con el modificador ***private*** obtenemos una visibilidad más restrictiva que por defecto, ya que

impide el acceso incluso para las clases vecinas. Un miembro, ya sea una atributo o un ,método, declarado privado es invisible desde fuera de la clase.

El modificador ***public*** hace que un miembro sea visible incluso desde clases externas. Otorga visibilidad total.

El modificador ***private*** lo utilizamos cuando queremos controlar los cambios de un atributo o cuando deseamos que no se conozca directamente su valor, o bien cuando queremos que un método se invocado desde otros métodos de la clase, pero no fuera de ella. El acceso a esos miembros privados deberá hacerse a través de algún método ***public*** de la misma clase.

- Ejemplo: Implementar la clase Alumno con los atributos nombre y nota media. Éste será un atributo privado, ya que interesa controlar el rango de valores válidos que estarán comprendidos entre 0 y 10, inclusive. El método publico asignarNota() será el encargado de controlar el valor asignado a la nota:

```
public class Alumno{
        public String nombre;
        private double notaMedia;

        public void asignarNota(double notaMedia){
                if(notaMedia < 0 || notaMedia > 0)
                        System.out.println("Nota incorrecta");
                else
                        this.notaMedia = notaMedia;
        }
}
```

De este modo, sólo es posible modificar la nota a través del método que la controla.

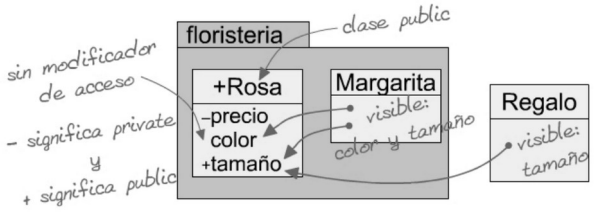
Resumen del alcance de la visibilidad según el modificador de acceso:

	Visible desde...		
	la propia clase	clases vecinas	clases externas
`private`	✓		
sin modificador	✓	✓	
`public`	✓	✓	✓

- Ejemplo: Visibilidad de los diferentes atributos de la clase Rosa vista anteriormente, dependiendo del tipo de modificador de acceso que pongamos:

```
public class Rosa{
        private double precio; //invisible fuera de la clase
        String color; //visible por clases vecinas
        public double tamanyo; //visibilidad total
}
```

En la siguiente figura podemos ver la visibilidad de ***private***, ***public*** y **por defecto**.



2.3.3.- Métodos getter /setter

Un atributo público puede ser modificado desde cualquier clase, lo que a veces tiene sus inconvenientes, ya que es imposible controlar los valores asignados, que pueden no tener sentido. Por ejemplo, nada impide que se asigne a un atributo edad un valor negativo, al incoherente. Para ello, existe una convención para ocultar atributos públicos, y en su lugar, crear dos métodos: el primer (***set***) permite asignar un valor al atributo, controlando el rango válido de valores, Y el segundo (***get***) devuelve el atributo, lo que permite conocer su valor. Los métodos ***get*** / ***set*** hacen, en la práctica, que un atributo no visible se comporte como si lo fuera. Estos métodos se identifican con ***get*** / ***set*** seguido del nombre del atributo.

- Ejemplo: Podemos ver en el siguiente ejemplo que las ventajas de utilizar métodos set / get son que la implementación de la clase se encapsula, ocultando los detalles, y por otro lado permite controlar los valores asignados a los atributos. Al ejemplo se ha limitado el uso de valores negativos para la edad.

```
class Persona{
        private int edad;
        . . .
        public void setEdad(int edad){
                if(edad >= 0)//solo los valores positivos tienen sentido
                        this.edad = edad;
                //en caso contrario no se modifica la edad
        }

        public int getEdad(){
                return edad;
        }

}
```

Uno de los pilares en los que se basa la POO es el **ENCAPSULAMIENTO**. Básicamente, el **encapsulamiento** consiste en definir todas las propiedades y el comportamiento de una clase dentro de esa clase (). Nunca deberemos mezclar parte de una clase con otra distinta para resolver una problema puntual.

El concepto de proteger los valores de una clase también va asociado al encapsulamiento. Con ello se impide que los atributos de una clase sean directamente modificados o consultados desde fuera de la clase. Los atributos serán ***private*** y se usan ***getters*** y ***setters*** como métodos de acceso.

La **ocultación** facilita el **encapsulamiento**.

- Ejemplo:

```
class Guerrero{
    attribute vitalidad
}
class Monstruo {
    method atacar(Guerrero barbaro){
        barbaro.vitalidad = barbaro.vitalidad - 10
    }
}
```

Cuando el monstruo ataca al guerrero lo debilita cambiando el valor de la propiedad vitalidad directamente. Funciona, pero a nivel conceptual no es conveniente hacerlo así ya que el objeto monstruo no tiene porque conocer como el bárbaro maneja su vida. El monstruo sólo tiene que mandar el mensaje y el bárbaro lo tiene que entender

Desde el punto de vista de la escalabilidad y de un lenguaje de programación, si el día de mañana tenemos diferentes objetos que tienen que modificar dicha propiedad, nuestro código se puede volver difícil de cambiar. Si tenemos encapsulado ese comportamiento, sólo vamos a tener que hacer un cambio en un solo lado.

La manera más amigable de hacerlo sería:

```
class Guerrero {
    attribute vitalidad
    method recibirAtaque(int puntuacionDeAtaque){
        self.vitalidad = self.vitalidad - puntuacionDeAtaque
    }
}

class Monstruo {
    attribute puntuacionDeAtaque
    method atacar(Guerrero barbaro){
        barbaro.recibirAtaque(puntuacionDeAtaque)
    }
}
```

3.- EJERCICIOS - 1

1. Crear la clase GatoSimple. Sus atributos serán las características que tienen los gatos: son de una color determinado, pertenecen a una raza, tienen una edad, tienen un determinado sexo (machos, hembras o hermafrodita) y tienen un peso que se puede expresar en kilogramos.

   Respecto a sus métodos, serán acciones que están asociadas a los gatos. Los gatos maullan, ronronean, comen y si son machos se pelean entre ellos.

2. Crear la clase Cubo. Todos los cubos tienen una determinada capacidad, un color, están hechos de un determinado material (plástico, latón, etc) y pueden que tengan asa o puede que no. Un cubo se fabrica con el propósito de contener líquido; por lo tanto otra característica es la cantidad de líquido que contiene un un momento determinado. Por ahora, sólo nos interesa saber la capacidad máxima y los litros que contiene el cubo en cada momento.

4.- EJERCICIOS - 2

1. Diseñar la clase CuentaCorriente, sabiendo que los datos necesarios son: saldo, límite de descubierto (cantidad de dinero que se permite sacar de una cuenta que ya está a cero), nombre y DNI del titular. Las operaciones típicas con una cuenta corriente son:
   - **Crear la cuenta**: se necesita el nombre y DNI del titular. El saldo inicial será 0 y el límite de descubierto será de -50 euros.
   - **Sacar dinero**: sólo se podrá sacar dinero hasta el límite de descubierto. El método debe indicar si ha sido posible llevar a cabo la operación.
   - **Ingresar dinero**: se incrementa el saldo.
   - **Mostrar información**: muestra la información disponible de la cuenta corriente.
2. En la clase CuentaCorriente sobrecargar los constructores para que permitan crear los siguientes objetos:
   - Sólo con el saldo inicial, no serán necesarios los datos del titular. Por defecto el límite de descubierto será de 0 euros.
   - Con un saldo inicial, con límite de descubierto y con el DNI del titular de la cuenta.
3. Para la clase CuentaCorriente escribir un programa que compruebe el funcionamiento de sus

métodos, incluido los constructores.

4. Modificar la visibilidad de la clase CuentaCorriente para que sea visible desde clases externas y la visibilidad de sus atributos para que:
   - saldo y límite no sean visibles para otras clase.
   - nombre sea público para cualquier clase.
   - dni sólo sea visible por clases vecinas.

   Realizar un programa para comprobar la visibilidad de los atributos.

5. Todas las cuentas corrientes con las que vamos a trabajar pertenecen al mismo banco. Añadir un atributo que almacene el nombre del banco (que es único) en la clase CuentaCorriente. Diseñar un método que permita modificar el nombre del banco (al que pertenecen todas las cuentas corrientes).

6. Diseñar la clase Texto que gestiona una cadena de caracteres con algunas características:
   - La cadena de caracteres tendrá una longitud máxima, que se especifica en el constructor.
   - Permite añadir un carácter, al principio o al final, siempre y cuando exista espacio disponible.
   - Igualmente, permite añadir una cadena, al principio o al final del texto, siempre y cuando no se rebase el tamaño máximo establecido.
   - Es necesario saber cuántas vocales (mayúsculas y minúsculas) hay en el texto.

7. Diseñar la clase Banco de la que interesa guardar su nombre, capital y la dirección central, Los bancos tienen las siguientes restricciones:
   - Siempre tienen que tener un nombre que no puede ser modificado.
   - Si no se especifica, todos los bancos tienen un capital por defecto de 5,2 millones de euros al crearse.
   - El capital y la dirección de un banco son modificables.

   Modificar la clase CuentaCorriente para que cada una esté vinculada a un objeto de tipo Banco. Escribir los métodos necesarios en la clase CuentaCorriente para gestionar el banco

al que pertenece. Existe la posibilidad de que una cuenta corriente no está vinculada a ningún banco

8. Se quiere definir una clase que permita controlar un sintonizador digital de emisoras FM; concretamente, se desea dotar al controlador de una interfaz que permita subir (up) o bajar (down) la frecuencia (en saltos de 0,5 MHz) y mostrar la frecuencia sintonizada en un momento dado (display). Supondremos que el rango de frecuencias a manejar oscila entre los 80 MHz y os 108 MHz y que, al inicio, el controlador sintonice a 80 MHz. Si durante una operación de subida o bajada se sobrepasa uno de los dos límites, la frecuencia sintonizada debe pasar a ser la del extremo contrario. Escribir un pequeño programa principal para comprobar su funcionamiento.

9. Modelar una casa con muchas bombillas, de forma que cada bombilla se pueda encender o apagar individualmente. Para ello hacer una clase Bombilla con una variable privada que indique si está encendida o apagada, así como un método que nos diga el estado de una bombilla concreta. Además, queremos poner un interruptor general, de forma que si saltan los fusibles, todas las bombillas quedan apagadas. Cuando el fusible se repara, las bombillas vuelven a estar encendidas o apagadas, según estuvieran antes del percance. Cada bombilla se encienda y se apaga individualmente, pero sólo responde que está encendida si su interruptor particular está activado y además hay luz general.

10. Hemos recibido un encargo para definir los paquetes y las clases necesarias (sólo implementar los atributos y los constructores) para gestionar una empresa ferroviaria, en la que se distinguen dos grandes grupos: el personal y la maquinaria. En el primero se ubican todos los empleados de la empresa, que se clasifican en tres grupos: los maquinistas, los mecánicos y los jefes de estación. De cada uno de ellos es necesario guardar:

    - Maquinistas: su nombre completo, su documento nacional de identidad, su sueldo mensual y el rango que tienen adquirido.
    - Mecánicos: su nombre completo, teléfono (para contactar en caso de urgencia) y en qué especialidad desarrollan su trabajo (frenos, hidráulica, electricidad, etc)
    - Jefes de estación: su nombre completo y DNI.

    En la parte de maquinaria podemos encontrar trenes, locomotoras y vagones. De cada uno de ellos hay que considerar:

- Vagones: tienen una capacidad máxima de carga (en kilos), una capacidad actual (de la carga que tienen en un momento dado) y el tipo de mercancía con el que están cargados.

- Locomotoras: disponen de una matrícula (que las identifica), la potencia de sus motores y una antigüedad (año de fabricación). Además, cada locomotora tiene asignado un mecánico que se encarga de su mantenimiento.

- Trenes: están formados por una locomotora y un máximo de 5 vagones. Cada tren tiene asignado un maquinista que es responsable de él.

Todas las clases correspondientes al personal (Maquinista, Mecanico y JefeEstacion) serán de uso público. Entre las clases relativas a la maquinaria sólo será posible construir, desde clases externas, objetos de tipo Tren y de tipo Locomotora. La clase Vagón será sólo visible por sus clases vecinas.

5.- EJERCICIOS - 3

1. En un punto fronterizo recogemos diariamente la información referente al tránsito de personas. Para una persona nos interesa almacenar su DNI y su nombre completo. Se pide una aplicación que presente las siguientes opciones:

    1. Paso de frontera.
    2. Mostrar todas las personas.
    3. Búsqueda por nombre.
    4. Búsqueda por DNI.
    5. Salir

    Mediante la opción 1 se recogen los datos de la persona que transita a través de la frontera. La opción 2 muestra la información de todas las personas que han pasado por la frontera. Mediante la opción 3 se solicita un nombre por teclado y se muestra la información de todas las personas cuyo nombre coincide con el introducido. Y por último, en la opción 4 introducimos el DNI de una persona de la que obtendremos toda su información

2. Diseñar la clase Calendario que representa una fecha concreta (año, mes y día). La clase debe disponer de los métodos:

    - Calendario (int año, int mes, int dia) → Crea un objeto con los datos pasado como parámetros, siempre y cuando, la fecha que representa sea correcta.

    - void incrementarDia (int cantidad) → Incrementa, si resulta correcto, la fecha del

calendario en el número de días especificados.

- void incrementarMes (int cantidad) → Incrementa la fecha del calendario en el número de meses especificados, siempre que la fecha resultante sea correcta.
- void incrementarAño (int cantidad) → Incrementa la fecha del calendario en el número de años especificado, salvo que el año resultante sea el 0 (que no existió).
- void mostrar () → Muestra la fecha por consola.
- boolean iguales (Calendario otraFecha) → Determina si la fecha que invoca la función y la que se pasa como parámetro son iguales o distintas.

3. Escribir la clase Punto que representa un punto de dos dimensiones (con una componente x y una componente y), con los métodos:
   - Punto (double x, double y) → Construye un objeto con los datos pasados por parámetros.
   - void desplazaX (double dx) → Desplaza la componente x la cantidad dx.
   - void desplazaY (double dy) → Desplaza la componente y la cantidad dy.
   - void desplaza (double dx, double dy) → Desplaza ambas componentes las cantidades dx en el eje x y dy en la componente y.
   - void distanciaEuclidea (Punto otro) → Calcula y devuelve la distancia euclídea entre el punto que invoca la función y el punto otro.
   - void muestra () → Muestra por consola la información relativa al punto.