

## Tema 05 - POO, Profundización

Objetivos del Tema:

- \* Profundizar en la creación de **paquetes**
- \* Reforzar el concepto de **clase**
- \* Profundizar en el diseño de las **clases**
- \* Reforzar el concepto de **constructor**
- \* Introducir el concepto de **interface**
- \* Introducir el concepto **clase abstracta** y profundizar en los de **herencia** y **polimorfismo**
- \* Introducir el concepto de **clases wrapper**
- \* Introducir el concepto del modificador **final**
- \* Profundizar en los conceptos de **sobrecarga (overload)** y **sobreescritura (override)**
- \* Reforzar el concepto de **super**
- \* Introducir el concepto de **Casting entre objetos**
- \* Trabajar con fechas **LocalDate** y **ChronoUnit**

### 1.- Paquetes

En el capítulo 3 ya hablamos de los *paquetes* y de la variable de entorno *CLASSPATH*. A modo de repaso recordaremos algunos conceptos básicos:

- Los paquetes son un conjunto de clases que físicamente se encuentran en la misma carpeta y que guardan relación entre sí.
- Los paquetes pueden a su vez contener a otros paquetes.
- Java mantiene sus clases estándar organizadas en paquetes, p.ej.: **java.util.\***, del cual hemos utilizado alguna clase como **java.util.Scanner**.
- Para utilizar una clase contenida en un paquete tenemos dos formas de hacerlo, lo vemos con un ejemplo:

**java.util.Scanner s= new java.util.Scanner(System.in);**

o incluyendo el correspondiente *import* al principio del fichero:

**import java.util.Scanner;**

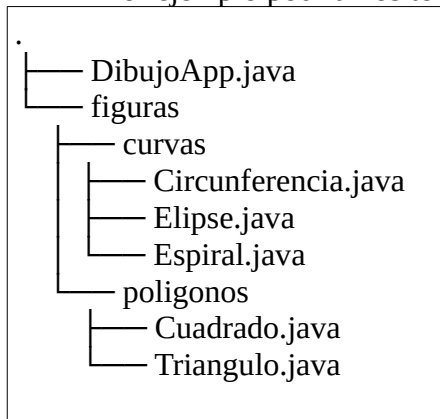
.....

**Scanner s=new Scanner(System.in);**

- Los paquetes nos permiten poder tener clases con el mismo nombre si se encuentran en distintos paquetes (recordad que una clase tiene que estar guardada en un fichero con el mismo nombre de la clase *.java* y el Sistema Operativo no permite tener 2 ficheros con el mismo nombre en la misma carpeta).
- Los paquetes nos permiten tener acceso *de paquete* entre las clases del mismo. Las variables miembro de las clases que no tengan modificador de acceso estarán directamente accesibles para otras clases del mismo paquete.

Para crear un paquete lo único que tenemos que hacer es crear una nueva carpeta con el nombre del paquete dentro del raíz de nuestro proyecto y depositar en la misma todos los ficheros de las clases que queremos que estén incluidas en el paquete. Cada fichero de cada clase debe empezar con la declaración de paquete **package nombre\_del\_paquete;**

Por ejemplo podríamos tener una estructura de paquetes como la siguiente:

	<p>Tenemos en este ejemplo el paquete por defecto representado por el ".", dentro de éste el paquete <b>figuras</b>, dentro de éste tenemos dos paquetes <b>curvas</b> y <b>poligonos</b>, respectivamente. Cada uno de éstos últimos con sus clases. Por ejemplo la clase <b>Circunferencia</b>, comenzará con la sentencia <b>package figuras.curvas;</b> y la clase <b>Cuadrado</b> con la sentencia <b>package figuras.poligonos;</b></p> <p>En la clase <b>DibujoApp</b>, que en este caso se encuentra en el raíz del proyecto (paquete por defecto) haremos uso de estas clases mediante la correspondiente sentencia import:</p> <p><b>import figuras.curvas.*; import figuras.poligonos.*;</b></p>
---	---

## 2.- Clases

Cuando se diseña una aplicación en POO se persigue encontrar un conjunto de clases que representen una **abstracción** del problema a resolver. Una vez hecho el diseño de esas clases, se procederá a la implementación de las mismas siguiendo el principio de **caja negra**, ocultando los detalles de la implementación al "mundo exterior".

El principio de **ocultación** consiste en prohibir el acceso a los detalles de implementación de las clases ocultando las variables miembro de las mismas (modificador de acceso **private**) y dando acceso a éstas solamente a través de la *interfaz* de la clase (*setters* y *getters*).

También se persigue el objetivo de que si hay que hacer modificaciones a una clase si no modificamos su *interfaz* otras clases que hagan uso de ella no se verán afectadas. Buscamos la independencia de la implementación.

El nivel de acceso a los miembros de una clase viene dado por los modificadores de acceso:

- **public** cualquier clase tiene acceso a este miembro
- **protected** las clases que heredan y comparten paquete tienen acceso a este miembro
- **default (si se omite el modificador de acceso)** las clases que estén en el mismo paquete tienen acceso a este miembro
- **private** sólo la clase a la que pertenece este miembro tiene acceso al mismo

Por descontado que una clase siempre tiene acceso a sus miembros cualquiera que sea el modificador de acceso que tengan.

La siguiente tabla pretende esclarecer cualquier duda:

Modificador/Acceso	Clase	Paquete	Subclase	Todos
<b>public</b>	S	S	S	S
<b>protected</b>	S	S	S	N
<b>default</b>	S	S	N	N
<b>private</b>	S	N	N	N

A las clases también le son aplicables los mismos modificadores de acceso, sólo que **private** y **protected** quedan reservados para las **inner classes**. Lo normal es que le asignemos **public** o **nada (default)**. En el caso de **public** esa clase es reutilizable en cualquier otra clase o aplicación y en el caso de **default (nada)** la clase sólo estará accesible dentro del mismo paquete.

Para profundizar en el tema, ver los siguientes vídeos:

<https://drive.google.com/open?id=1C-eY7MVOhRoQ4RaAtKwsk3gjrI5lFn8E>

<https://drive.google.com/open?id=1ud0za-eS6Tt0G7llrE0v8mgJHMYLjoXo>

## 3.- this

Cuando se *instancia* un objeto de cualquier clase, en los métodos no estáticos de la misma se puede hacer referencia a dicho objeto mediante **this**. **this** es una palabra reservada del lenguaje que hace referencia al objeto actual que llama al método.

```
public class Persona {
    private String nombre, apellidos;
    private int edad;
    ....
    public void setEdad(int edad) {
        this.edad=edad; // Es necesario el uso de this, porque hay ambigüedad
    }
    public void setNombre(String nom) {
        nombre=nom; //aquí no es necesario usar this, porque no hay ambigüedad
    }
    ....
}
```

**this** también puede ser usado en un constructor para llamar a otro constructor, veremos un ejemplo más adelante cuando hablemos de constructores. En el ejemplo de la clase Estudiante, hay un constructor que llama a otro constructor **this(nombre,ap1,ap2);**. Si un constructor hace una llamada a otro constructor mediante **this()** esta instrucción tiene que ser la primera que ejecute dicho constructor.

#### 4.- java.lang.Object

Java define una clase especial llamada **Object** que es una superclase implícita de todas las demás clases, es decir, todas las clases *heredan* de **Object**. Esto significa que una variable de referencia de tipo **Object** puede referirse a un objeto de cualquier otra clase. Además, dado que los *arrays* se implementan como clases, una variable de tipo **Object** también puede referirse a cualquier *array*. **Object** define los siguientes métodos que son heredados por todos los objetos en **JAVA**:

Método	Propósito
<b>Class getClass()</b>	Obtiene la clase de un objeto en tiempo de ejecución
<b>int hashCode</b>	Devuelve el código hash asociado con el objeto invocado
<b>boolean equals(Object obj)</b>	Determina si un objeto es igual a otro
<b>Object clone()</b>	Crea un nuevo objeto que es el mismo que el objeto que se está clonando
<b>String toString()</b>	Devuelve una cadena que describe el objeto
<b>void notify()</b>	Reanuda la ejecución de un hilo esperando en el objeto invocado
<b>void notifyAll()</b>	Reanuda la ejecución de todo el hilo esperando en el objeto invocado
<b>void wait(long timeout)</b>	Espera en otro hilo de ejecución
<b>void wait(long timeout,int nanos)</b>	Espera en otro hilo de ejecución
<b>void wait()</b>	Espera en otro hilo de ejecución
<b>void finalize()</b>	Determina si un objeto es reciclado (obsoleto por JDK9)

\* Los métodos **getClass()**, **notify()**, **notifyAll()** y **wait()** se declaran como **final** lo cual quiere decir que no pueden ser sobreescritos.

A continuación echaremos un vistazo a los métodos más relevantes.

##### > **String toString()**

Proporciona la representación **String** de un Objeto y se usa para convertir un objeto a Cadena (**String**). El método predeterminado **toString()** para la clase **Object** devuelve una cadena que consiste en el nombre de la clase de la cual el objeto es una instancia, el carácter arroba '@' y la representación hexadecimal sin signo del código hash del objeto. En otras palabras, se define como:

```
//El comportamiento predeterminado de toString () es
//imprimir el nombre de la clase, luego @, luego la representación hexadecimal
//sin firmar del código hash del objeto
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Se recomienda sobrescribir el método **toString()** para obtener nuestra propia representación **String** de los objetos de la clase que estamos implementado. Este método es llamado automáticamente al imprimir mediante **System.out.println()**.

##### > **boolean equals(Object obj)**

Compara el objeto desde el que se llama al método ("**this**") con el objeto pasado por parámetro. Devuelve **true** si se "da" la igualdad o **false** en caso contrario. Lo que hace es comparar los "hashcodes" de ambos objetos, con lo cual lo que en realidad nos indica es si ambas referencias "apuntan" al mismo objeto.

#### 4.- java.lang.Object

Java define una clase especial llamada **Object** que es una superclase implícita de todas las demás clases, es decir, todas las clases *heredan* de **Object**. Esto significa que una variable de referencia de tipo **Object** puede referirse a un objeto de cualquier otra clase. Además, dado que los *arrays* se implementan como clases, una variable de tipo **Object** también puede referirse a cualquier *array*. **Object** define los siguientes métodos que son heredados por todos los objetos en **JAVA**:

Método	Propósito
<b>Class getClass()</b>	Obtiene la clase de un objeto en tiempo de ejecución
<b>int hashCode</b>	Devuelve el código hash asociado con el objeto invocado
<b>boolean equals(Object obj)</b>	Determina si un objeto es igual a otro
<b>Object clone()</b>	Crea un nuevo objeto que es el mismo que el objeto que se está clonando
<b>String toString()</b>	Devuelve una cadena que describe el objeto
<b>void notify()</b>	Reanuda la ejecución de un hilo esperando en el objeto invocado
<b>void notifyAll()</b>	Reanuda la ejecución de todo el hilo esperando en el objeto invocado
<b>void wait(long timeout)</b>	Espera en otro hilo de ejecución
<b>void wait(long timeout,int nanos)</b>	Espera en otro hilo de ejecución
<b>void wait()</b>	Espera en otro hilo de ejecución
<b>void finalize()</b>	Determina si un objeto es reciclado (obsoleto por JDK9)

\* Los métodos *getClass()*, *notify()*, *notifyAll()* y *wait()* se declaran como **final** lo cual quiere decir que no pueden ser sobreescritos.

A continuación echaremos un vistazo a los métodos más relevantes.

##### > **String toString()**

Proporciona la representación String de un Objeto y se usa para convertir un objeto a Cadena (String). El método predeterminado **toString()** para la clase *Object* devuelve una cadena que consiste en el nombre de la clase de la cual el objeto es una instancia, el carácter arroba '@' y la representación hexadecimal sin signo del código hash del objeto. En otras palabras, se define como:

```
//El comportamiento predeterminado de toString () es
//imprimir el nombre de la clase, luego @, luego la representación hexadecimal
//sin firmar del código hash del objeto
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Se recomienda sobrescribir el método **toString()** para obtener nuestra propia representación String de los objetos de la clase que estamos implementado. Este método es llamado automáticamente al imprimir mediante **System.out.println()**.

##### > **Class getClass()**

Devuelve la clase de “este” objeto y se utiliza para obtener la clase en tiempo de ejecución real del objeto. También se puede usar para obtener metadatos de esta clase.

##### > **Object clone()**

Devuelve un nuevo objeto (no una referencia) que es una copia exacta del mismo objeto desde el que se llama al método. Mejor que usar este método sería implementar un **constructor de copia** en la clase correspondiente. Para usar este método y que no se produzca una excepción

***java.lang.CloneNotSupportedException*** la clase en la que vayamos a utilizarlo debe implementar el interfaz ***Cloneable***, y sobrescribir el método ***Clone*** que es heredado de la clase ***Object***, superclase de todas las clases en JAVA. Dicho método es declarado como ***protected*** por lo cual estamos obligados a implementarlo.

## 5.- Atributos y métodos de una clase

### 5.1.- static

Ya se ha hablado sobre como afecta el uso de la palabra reservada de JAVA ***static*** a los miembros de una clase. Cuando hablamos de los miembros de la una clase nos referimos tanto a las variables miembro o atributos, que definen el ***estado*** de un objeto en concreto de dicha clase como a sus ***métodos***, que definen el ***comportamiento*** de los objetos que se ***instancien*** (que se creen) de la clase.

***static*** aplicado a un atributo significa que no habrá una copia de dicho atributo en cada uno de los objetos que se instancien de la clase, sino que ese atributo será compartido por todos ellos.

***static*** aplicado a un método de la clase significa que para ejecutar dicho método, no hace falta instanciar un objeto de la clase, sino que puede ser ejecutado directamente, utilizando el nombre de la clase "." nombre del método. Por ejemplo, hemos utilizado el método ***Math.sqrt*** para calcular la raíz cuadrada de un número. El método ***sqrt*** pertenece a la clase ***Math*** que pertenece a la ***API*** de JAVA y para invocarlo no necesitamos crear objetos de la clase ***Math***, simplemente lo invocamos de la siguiente forma: ***double raiz2=Math.sqrt(2);***

Detalles a tener en cuenta:

- Un método ***static*** no puede hacer uso de ***this***
- Un método ***static*** no puede acceder a los atributos de la clase que no sean ***static***
- Cuando una clase tiene todos o casi todos sus miembros (atributos y métodos) ***static*** se la suele llamar ***utility class*** o ***helper class***. Por ejemplo, la clase ***java.lang.Math*** (<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Math.html>) que viene con la API de JAVA es una utility class que trae todas las funciones matemáticas estándar.

Para profundizar más ver los siguientes vídeos:

[https://drive.google.com/open?id=1hhE\\_nDHbe9TAErN4PS2CTVCsiLAEQCzd](https://drive.google.com/open?id=1hhE_nDHbe9TAErN4PS2CTVCsiLAEQCzd)

<https://drive.google.com/open?id=19KXVl3qBN3iP6TevUeVsxYthSX3nweg2>

### 5.2.- atributos y métodos de instancia

Cuando declaramos variables dentro de una clase y fuera de cualquier método, estas variables son llamadas variables miembro o de instancia o atributos de la clase. Estos atributos son únicos para cada objeto. Por ejemplo, si creamos una clase ***Persona*** y, dependiendo del problema del mundo real que estemos modelando, nos interesa guardar su ***dni***, ***nombre***, ***apellidos***, ***direccion*** y ***teléfono***, está claro que estas variables no deberían estar compartidas por todos los objetos de la clase ***Persona***. Cada objeto debería tener sus propios valores para estos atributos y así cuando instanciamos una persona cuyo ***dni*** sea 12345678Z, ***nombre*** Luis, ***apellidos*** González Fernández, ..., está claro que ese objeto es único y se diferencia de otros objetos de la clase por su ***estado***, es decir, los valores de sus atributos.

Cuando dentro de una clase declaramos métodos no ***static*** necesitamos ***instanciar*** un objeto de la clase para poder invocarlos. Por ejemplo, si tenemos una clase ***Empleado*** que guarda ***dni***, ***nombre***, ***apellidos***, ***direccion***, ***teléfono***, ***numHijos***, ***sueldo***, que tiene un método ***getIRPF*** que calcula el tanto por ciento de IRFP que le corresponde pagar a un empleado en función de su sueldo y el número de hijos que tiene, está claro que este método no puede ser ***static*** (=método de clase) ya que necesitamos un objeto que guarde el ***sueldo*** y ***numHijos*** para poder calcularlo.

Un método de instancia puede acceder a los atributos y métodos ***static*** de la clase.

## 6.- Métodos de una clase

### 6.1.- Parámetros

Cuando un método recibe parámetros de tipo básico estos parámetros se reciben por ***copia*** mientras que si los parámetros son tipo objeto lo que se recibe es una ***referencia*** al objeto. La diferencia entre paso de parámetros por ***copia*** y paso de parámetros por ***referencia*** está en que en el primer caso los cambios que se hagan dentro del método sobre el parámetro no afectan a la variable externa que se pudiera utilizar en la llamada, mientras que en el segundo caso si.

Veámoslo con un ejemplo:

```
public class Prueba {
    public static double cuadrado(double x) {
        x=x*x; // modificamos el parámetro x (variable local)
        return x;
    }
    public static void main(String[] args) {
        double x=5;
        double y=cuadrado(x); //(*)
        System.out.println(x);
        System.out.println(y);
    }
}
```

(\*) Aunque dentro del método ***cuadrado*** se asigna un nuevo valor a *x*, *x* dentro de éste método es una copia de la *x* con la que se llamó al mismo, por lo tanto los cambios sobre el parámetro *x* (variable local), no afectan a la variable *x* con la que se llamó al método, que seguirá teniendo el valor 5 después de la llamada. El resultado es que se imprime 5.0 (*x* que no cambia su valor) y 25.0 valor asignado a *y*.

Recuerda son tipos de datos básicos ***byte, short, int, long, float, double, char y boolean***.

Sin embargo veamos lo que ocurre en el siguiente ejemplo cuando se pasa un objeto a un método.

```
public class Rectangulo {
    private double ancho, alto;
    public Rectangulo(double w, double h) {
        this.ancho=w; this.alto=h;
    }
    public void setAncho(double ancho) { this.ancho=ancho; }
    public void setAlto(double alto) { this.alto=alto; }
    public static void cambiaRectangulo(Rectangulo r) {
        r.setAlto(0); r.setAncho(0);
    }
    public String toString() {
        return "Soy un rectángulo de "+this.ancho+" de ancho y "+this.alto+" de alto.";
    }
    public static void main(String[] args) {
        Rectangulo r=new Rectangulo(4.3,5.6);
        System.out.println(r);
        Rectangulo.cambiaRectangulo(r);
        System.out.println(r);
    }
}
```

La ejecución del programa anterior imprimirá por pantalla:

***Soy un rectángulo de 4.3 de ancho y 5.6 de alto.***

***Soy un rectángulo de 0.0 de ancho y 0.0 de alto.***

Se ve que después de llamar al método ***Rectangulo.cambiarRectangulo(r)*** el objeto ***r*** ha quedado modificado pasando su ancho de ser **4.3** a **0.0** y su alto de **5.6** a **0.0**.

### 6.2.- Recursividad

Se dice que un método es recursivo si dentro del método hay una llamada a sí mismo.

Ejemplo:

```

class Prueba {
    public static int factorial(int n) {
        if (n==0) {
            return 1;
        } else {
            return n*factorial(n-1);
        }
    }
    public static void main(String[] args) {
        int f=factorial(3);
        System.out.println(f);
    }
}

```

Como ves dentro del método *factorial* hay una llamada a él mismo *factorial(n-1)*. Para profundizar más en el tema, podéis ver los siguientes vídeos:

[https://drive.google.com/open?id=1qbs2c-qFlXVt\\_7TiUo72KTyRWS0Zg2dD](https://drive.google.com/open?id=1qbs2c-qFlXVt_7TiUo72KTyRWS0Zg2dD)

<https://drive.google.com/open?id=1Bx2kPSSlkj5D1mQ9KTStJFxfHXYmpXSm>

Otro ejemplo de método recursivo sería el cálculo del término n-ésimo de la sucesión de *fibonacci* (ejercicio resuelto en los primeros temas):

```

class Prueba {
    public static int fibonacci(int n) {
        if (n==0 || n==1) {
            return 1;
        } else {
            return fibonacci(n-1)+fibonacci(n-2);
        }
    }
    public static void main(String[] args) {
        int f=fibonacci(3);
        System.out.println(f);
    }
}

```

Si no lo entiendes no pasa nada en Desarrollo de Aplicaciones Web rara vez vas a tener que utilizar la recursividad.

## 7.- Constructores

En JAVA un constructor de una clase es un método especial que tiene el mismo nombre de la clase. No se especifica que tipo de dato devuelve porque va implícito (devuelve un objeto de la clase). Si que puede llevar modificador de acceso, que normalmente será **public**. El constructor se suele utilizar para inicializar los atributos de la clase.

Para *instanciar* o crear un objeto de una clase se utiliza el operador **new**. Este operador va seguido de una llamada al constructor de la clase. **new** construye un objeto de la clase que especificamos tras él invocando al constructor que indicamos. Para ello reserva la cantidad de memoria necesaria para almacenar el objeto que se crea en la zona de memoria llamada **heap** y devuelve una referencia a esa zona de memoria ocupada por el objeto.

Si no especificamos un constructor en JAVA se crea uno vacío (constructor que no recibe ningún parámetro) por defecto. Por ejemplo:

```

public class Persona {
    String nombre;
    int edad;
}

```



```

        public static void main(String[] args) {
            Persona p=new Persona();
        }
    }

```

En este sencillo ejemplo hemos *instanciado* un objeto de la clase persona mediante la Instrucción:

```

Persona p=new Persona()

```

Si ponemos un constructor distinto del constructor vacío "desaparece" el constructor vacío

```

public class Persona {
    String nombre;
    int edad;
    public Persona() {}
    public Persona(String nombre, int edad) {
        this.nombre=nombre;
        this.edad=edad;
    }
}

```

```

    public static void main(String[] args) {
        //Persona p=new Persona(); <-- daría un error
        Persona p2=new Persona("Luis",33); <-- ahora es la única forma de
construir un objeto de la clase Persona, ya que el constructor vacío ha "desaparecido"
    }
}

```

Lo normal es que el constructor inicialice los atributos de la clase, aunque si no lo hacemos se asigna un valor por defecto: **cero** para variables **numéricas** y **char**, **false** para variables **boolean** y **null** para variables tipo **objeto**.



## 7.1.- Sobrecarga de constructores

*Sobrecargar* un constructor nos permite inicializar un objeto de múltiples formas. Es muy común que una clase de objetos tenga dos constructores. Uno en el que se le pasan los valores más importantes de sus atributos uno a uno y otro llamado de ***copia*** en el que se le pasa un objeto de la misma clase y lo que se hace es copiar los atributos del objeto pasado por referencia en el objeto que se está creando. Veámoslo con un ejemplo:

```
//Fichero Estudiante.java
public class Estudiante {
    String nombre;
    String ap1;
    String ap2;
    int edad;
    String curso;
    public Estudiante() {

    }
    public Estudiante(String nombre, String ap1, String ap2) { //constructor
parametrizado, deja edad y curso sin inicializar
        this.nombre=nombre; this.ap1=ap1; this.ap2=ap2;
    }
    public Estudiante(String nombre, String ap1, String ap2, int edad, String curso)
{ //constructor parametrizado
        this(nombre,ap1,ap2); //llamamos al constructor anterior
        this.edad=edad; this.curso=curso;
    }
    public Estudiante(Estudiante e) { //constructor de copia
        this(e.nombre,e.ap1,e.ap2,e.edad,e.curso);
        //this.nombre=e.nombre; this.ap1=e.ap1; this.ap2=e.ap2;
        //this.edad=e.edad; this.curso=e.curso;
    }
    public String toString() {
        return String.format("Nombre: %s %s %s, Edad: %d, Curso: %s",
            this.nombre, this.ap1, this.ap2, this.edad, this.curso);
    }
}

//Fichero Main.java
public class Main {
    public static void main(String[] args) {
        Estudiante e1=new Estudiante(); // objeto sin inicializar
        Estudiante e2=new Estudiante("Luis","González","Martínez",19,"DAW1");
        Estudiante e3=new Estudiante(e2); // objeto "réplica del anterior"
        System.out.println(e1);
        System.out.println(e2);
        System.out.println(e3);
        System.out.println(e2==e3);
    }
}
```

La ejecución de la clase Main anterior producirá la siguiente salida:

```
Nombre: null null null, Edad: 0, Curso: null
Nombre: Luis González Martínez, Edad: 19, Curso: DAW1
Nombre: Luis González Martínez, Edad: 19, Curso: DAW1
false
```

En el ejemplo vemos también como un constructor puede hacer uso de otro mediante una llamada a `this(...)`. Es muy conveniente hacer esto, pues si algún atributo necesitase de una inicialización especial, sólo lo haríamos en un constructor y los demás harían uso de éste sin tener que repetir el código.

## 7.2.- Constructores de copia

Ya hemos visto un ejemplo en el punto anterior en el que se declaraba el constructor:

```
public Estudiante(Estudiente e)
```

En él se hacía una copia de las variables miembro del objeto pasado por *referencia* *e* sobre el objeto que se está creando. Por tanto, un constructor de copia, recibe un objeto de la misma clase y hace una copia de todos los atributos del objeto pasado por referencia sobre el objeto que se está creando.

## 7.3.- Referencias

Como ya hemos visto anteriormente, en JAVA para crear un objeto hacemos uso del operador **new** el cual se encarga de reservar la cantidad de memoria necesaria para almacenarlo y devuelve una **referencia** a esa zona de memoria donde se almacena el objeto. Así pues tenemos que tener mucho cuidado a la hora de asignar referencias, ya que no ocurre igual que en otros lenguajes de programación como C++ donde si que se hace una copia de objetos cuando se hace una asignación.

Siguiendo con el ejemplo de la clase Estudiante del punto anterior, si tenemos en nuestro código algo como esto:

```
Estudiante e1=new Estudiante("Luisa", "Martínez", "Pérez",21,"DAW1");  
Estudiante e2=a;
```

Ahora tanto *e1* como *e2*, hacen **referencia** al mismo objeto. Si hiciéramos cambios en el mismo tanto a través de la **referencia** *e1* como a través de *e2*, estaríamos cambiando el mismo objeto. En este caso **e1==e2** devolvería **true**, ya que ambas referencias "apuntan" al mismo objeto.

## 8.- Destruidores

En JAVA no existen *destruidores* como por ejemplo en C++, cuando un objeto deja de tener referencias al mismo, será destruido por el recolector de basura. El recolector de basura es un proceso que se ejecuta en segundo plano en la máquina virtual de JAVA que se encarga de ir eliminando aquellos objetos que ya no tienen ninguna referencia.

## 9. Interfaces

En POO el principio de **encapsulación** (caja negra) y **ocultación** (modificadores de visibilidad) nos ayuda a ocultar los detalles de implementación de una clase. Normalmente lo único que exponemos al mundo exterior es el **interface** de la clase, lo cual suele ser el conjunto de métodos public de la misma. De esta forma obligamos a quien haga uso de nuestra clase a hacer uso de la misma a través de sus métodos (interface).

En JAVA existen los **interface** que son la materialización de este concepto. En un **interface** definimos una serie de métodos que aquellas clases que lo **implementen** se ven obligadas a implementar. Dichas clases utilizarán la palabra clave **implements** nombre-interfaz, en la declaración de la clase. Veámoslo con un ejemplo:

```
//fichero Figura2D.java
public interface Figura2D {
    int RED=0xff0000; int GREEN=0x00ff00; int BLUE=0x0000ff;
    public double area();
    public double perimetro();
    public void dibujar();
}

-----
//fichero Cuadrado.java
public class Cuadrado implements Figura2D {
    double lado;
    public Cuadrado(double lado) { this.lado=lado; }
    public double area() { return this.lado*this.lado; }
    public double perimetro() { return 4*this.lado; }
    public void dibujar() { //.....
    }
    public String toString() {
        return "Soy un cuadrado de lado: "+this.lado;
    }
}

-----
//fichero Circulo.java
public class Circulo implements Figura2D {
    double radio;
    public Circulo(double radio) { this.radio=radio; }
    public double area() { return java.lang.Math.PI*this.radio*this.radio; }
    public double perimetro() { return 2*java.lang.Math.PI*this.radio; }
    public void dibujar() { //.....
    }
    public String toString() {
        return "Soy un Círculo de radio: "+this.radio;
    }
}

-----
//fichero Main.java
public class Main {
    public static void main(String[] args) {
        Figura2D f1=new Circulo(3); Figura2D f2=new Cuadrado(4);
        Figura2D f3=new Cuadrado(7); Figura2D f4=new Circulo(5);
        Figura2D[] figuras= {f1,f2,f3,f4};
        for(Figura2D f:figuras) {
            System.out.println(f.toString());
            System.out.println("Area: "+f.area());
            System.out.println("Perímetro: "+f.perimetro());
            System.out.println("-----");
        }
    }
}

-----
```

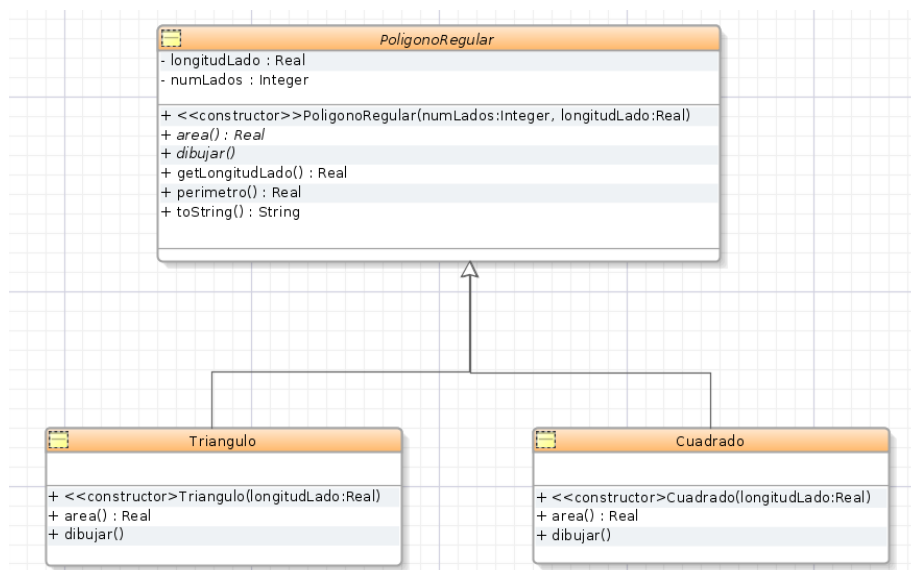
Al ejecutar la clase *Main.java* obtendríamos la siguiente salida:

```
Soy un Círculo de radio: 3.0
Area: 28.274333882308138
Perímetro: 18.84955592153876
-----
Soy un cuadrado de lado: 4.0
Area: 16.0
Perímetro: 16.0
-----
Soy un cuadrado de lado: 7.0
Area: 49.0
Perímetro: 28.0
-----
Soy un Círculo de radio: 5.0
Area: 78.53981633974483
Perímetro: 31.41592653589793
-----
```

Si en un interface declaramos variables éstas serán automáticamente final (constantes)

## 10. Clases abstractas, Herencia, Polimorfismo

Una clase **abstracta** es aquella que no implementa todos sus métodos, por no disponer de la información necesaria para implementarlos. La implementación de estos métodos declarados como **abstract** se deja para las clases que heredan de ésta. El concepto es muy parecido al de **interface** sólo que ahora sí que podemos tener variables miembro e implementar algunos métodos, salvo que habrá algunos que se declararán como **abstract** y su implementación se dejará "pendiente" para las clases que hereden que deberán implementarlos obligatoriamente. En el siguiente diagrama vemos un ejemplo de clase abstracta. La clase **PoligonoRegular** tiene dos métodos abstractos: **dibujar** y **area** (ambos aparecen en cursiva, lo cual indica que son abstractos). Está claro que no es lo mismo dibujar un triángulo que un cuadrado que cualquier otro polígono regular. E igual pasa con el cálculo del área del polígono. Por lo tanto, ambos métodos se declaran **abstract** dejando su implementación para las clases hijas, en este caso **Triangulo** y **Cuadrado**.



Cuando una clase **hereda** de otra clase *recibe* automáticamente todos sus miembros, tanto métodos como atributos. Es como si se hiciera una copia de todo lo que hay en la clase padre en la clase hija. Cuando una clase hija implementa de nuevo un método existente en el padre, se dice que está **sobreescribiendo** dicho método el cual oculta al método del padre (aunque todavía podemos acceder al método del padre usando **super**). En el ejemplo anterior tanto la clase **Triangulo** como la clase **Cuadrado**, heredan todos los atributos y métodos de la clase padre **PoligonoRegular** y además al ser ésta última una clase abstracta se ven obligados a implementar los métodos **dibujar** y **area** que están declarados como **abstract** en la clase padre.

El **Polimorfismo** consiste en que las clases hijas sobrescriben métodos del padre o implementan métodos abstractos del padre, para adecuarlos a un comportamiento especializado en las clases hijas. A continuación vemos como se implementaría el diagrama de clases anterior en JAVA:

```
/****** Fichero PoligonoRegular.java *****/
public abstract class PoligonoRegular {
    private double longitudLado;
    private int numLados;
    public PoligonoRegular(int numLados, double longitudLado) {
        this.numLados=numLados;
        this.longitudLado=longitudLado;
    }
    public abstract void dibujar();
    public abstract double area();
    public double perimetro() {
        return this.numLados*this.longitudLado;
    }
    public double getLongitudLado() {
        return this.longitudLado;
    }
    public String toString() {
        return String.format("%s de lado: %.2f y área: %.2f\n",
                               this.getClass().getName(),this.getLongitudLado(),this.area());
    }
}

/****** Fichero Triangulo.java *****/
class Triangulo extends PoligonoRegular {
    public Triangulo(double longitudLado) {
        super(3,longitudLado);
    }
    public double area() {
        double l=this.getLongitudLado();
        double h=Math.sqrt(l*l-l*l/4);
        return h*l/2;
    }
    public void dibujar() {
        double l=this.getLongitudLado();
        for(int i=0;i<l;i++) {
            System.out.print(" ".repeat((int)(l-i)));
            System.out.println("* ".repeat(i+1));
        }
        System.out.println(this);
    }
}

/****** Fichero Cuadrado.java *****/
class Cuadrado extends PoligonoRegular {
    public Cuadrado(double longitudLado) {
        super(4,longitudLado);
    }
    public double area() {
        return this.getLongitudLado()*this.getLongitudLado();
    }
    public void dibujar() {
        double l=this.getLongitudLado();
        for(int i=0;i<l;i++) {
            System.out.println("* ".repeat((int)l));
        }
        System.out.println(this);
    }
}

/****** Fichero Main.java *****/
class Main {
    public static void main(String[] args) {
        //PoligonoRegular pr=new PoligonoRegular(5,7.0);//Error no se puede instanciar una clase abstracta
        //Creemos una serie de PoligonosRegulares
        PoligonoRegular[] poligonos=new PoligonoRegular[4];
        for(int i=0; i<poligonos.length;i++) {
            if (i%2==0) {
                poligonos[i]=new Triangulo(i+1);
            } else {
                poligonos[i]=new Cuadrado(i+1);
            }
        }
        for(int i=0;i<poligonos.length;i++) {
            poligonos[i].dibujar();
        }
    }
}
```

La ejecución de la clase Main, produce el siguiente resultado:

```
*
Triangulo de lado: 1,00 y área: 0,43

* *
* *
Cuadrado de lado: 2,00 y área: 4,00

*
* *
* * *
Triangulo de lado: 3,00 y área: 3,90

* * * *
* * * *
* * * *
* * * *
Cuadrado de lado: 4,00 y área: 16,00
```

Vemos como la llamada al método polimórfico **dibujar** actúa de forma distinta según el **PoligonoRegular** sea un **Triangulo** o sea un **Cuadrado**. También vemos que, evidentemente, no se puede instanciar una clase abstracta, pues no tiene implementados sus métodos abstractos. Además el método **toString** de la clase padre hace uso de éstos métodos polimórficos, por lo que actúa de forma distinta en el **Triangulo** que en el **Cuadrado**.

## 11. Clases wrapper (envoltorio)

La filosofía de JAVA es que todo es un objeto, pero como sabemos existen 8 tipos de datos básicos que no lo son. JAVA implementa una clase para cada uno de ellos, una clase **wrapper**:

Tipo Básico	Clase Wrapper
boolean	Boolean
char	Char
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

A modo de ejemplo vamos a ver la clase **java.lang.Double**, como vemos se encuentra en el paquete **java.lang** que podemos considerar "forma parte del lenguaje". Al formar parte de este paquete podemos hacer uso de dicha clase sin necesidad de hacer **import**.

Tiene dos constructores:

**Double(double value)**

**Double(String s)**

Aunque si miramos la documentación de la API de JAVA veremos que aparecen ambos como **deprecated** (se desaconseja su uso y es posible que en el futuro desaparezca). Desde hace ya algunas versiones JAVA intenta hacer más fácil el uso de las clase **wrapper** permitiendo inicializarlas directamente con un valor del tipo de datos básico al cual "envuelven".

Veámoslo con un ejemplo:

```
Double d=new Double(5.5); //deprecated
```

```
Double d=5.5; //estamos creando un objeto de la clase Double no una variable tipo double
```

También podemos hacer uso del objeto Double directamente en expresiones aritméticas:

```
Double e=d+5; // e almacenaría el valor 10.5
```

Para crear un objeto tipo Double a partir de una cadena de caracteres podemos usar el constructor enumerado anteriormente:

```
String s="5.5";
```

```
Double f=new Double(s); //deprecated
```

Pero como está **deprecated** mejor usar el método estático de la clase para ello:

```
Double f=Double.parseDouble(s);
```

Para profundizar más sobre esta clase ver:

<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/Double.html>

Utilizaremos estas **clases wrapper** allí donde sólo se pueda emplear un objeto. Por ejemplo, en el tema anterior en el ejercicio de los secadores, empleábamos un diccionario:

```
TreeMap<Integer,Vendedor> vendedores=new TreeMap<>();
```

Al ser la clase TreeMap parametrizada (recibe dos tipos de clase como parámetros entre <>, sólo se le pueden pasar como parámetros clases de objetos no tipos de datos básicos. Por lo que **NO** podríamos hacer esto:

```
TreeMap<int,Vendedor> vendedores=new TreeMap<>();
```

**Error:**

| **unexpected type**

| **required: reference**

| **found: int**

| *TreeMap<int,Vendedor> vendedores=new TreeMap<>();*

## 12. Final

Hay tres contextos diferentes en los que se usa la palabra clave o reservada **final** en JAVA:

1) Cuando se aplica **final** a una variable:

a) Si la variable es de un tipo de dato básico, indica que a esta sólo se le puede asignar un valor inicial y a partir de ahí no se puede modificar. Ejemplo:

```
private final int a=5;
```

.....

```
a=6; //error: cannot assign a value to final variable a
```

b) Si la variable es de tipo objeto, aquí lo que indica es que la referencia al objeto no se puede cambiar, pero sí se puede modificar el objeto al que apunta la referencia. Ejemplo:

```
final Persona p1=new Persona("12345678Z","Luis","Pérez",35);
```

```
Persona p2=new Persona("23456789D","María","López",33);
```

```
p1.setNombre("Alfonso"); //bien le hemos cambiado el nombre al objeto p1 de Luis a Alfonso
```

```
p1=p2; //error: cannot assign a value to final variable p1
```

En ambos casos a) y b) el error se da en tiempo de compilación, quiere ésto decir que el programa no compilará y por tanto ni siquiera podremos llegar a ejecutarlo.

2) Cuando se aplica **final** a un método.

En este caso quiere decirse que el método no podrá ser **sobreescrito** en ninguna clase que herede de la clase donde se definió el método. De esta forma nos aseguramos que el método no podrá ser "ocultado" en las clases hijas. Si una clase hija intenta sobreescribirlo se producirá un error como el siguiente:



*error: nombreMetodo() in NombreClase cannot override NombreMetodo() in NombreClaseHija.*

3) Cuando se aplica **final** a una clase.

En este caso impedimos que se pueda haber clases que hereden de nuestra clase. Si intentamos heredar de una clase **final** obtendremos un error en tiempo de compilación como el siguiente:

*error: cannot inherit from final NombreClasePadre*

### 13.- Sobrecarga y sobreescritura de métodos

**Sobreescribimos (override)** un método cuando implementamos una clase que hereda de otra clase y "reescribimos" un método con el mismo prototipo que uno existente en el padre. Por ejemplo, en un apartado anterior la clase **Triangulo** sobreescribía el método **area** de la clase padre. Da igual que el método en la clase padre sea abstracto que no, al sobreescribirlo "desaparece" el método de la clase padre y en su lugar se invocará al método de la clase hija. El hecho de que sea **abstract** en la clase padre, obliga a la clase hija a implementarlo. A lo largo del curso hemos sobreescrito varias veces el método **toString** heredado de la clase **Object**. Sabemos que éste método imprime un **hashcode** del objeto. Pero en numerosas ocasiones esta información no nos era relevante y lo hemos sobreescrito. El hecho de sobreescribir un método no implica realmente que desaparezca en el padre, podremos invocarlo haciendo uso de la palabra reservada **super**. Por ejemplo: si en la clase **Triangulo**, **PoligonoRegular** quisiera que además de que se imprima "**Triangulo de lado: 3,00 y área: 3,90**", o "**Cuadrado de lado: 2,00 y área: 4,00**" también se imprima una referencia al objeto, podría añadirle al método una llamada al método **toString** de la clase **Object** de la siguiente forma:

```
public String toString() {  
    return String.format("Referencia objeto: %s. %s de lado: %.2f y área: %.2f\  
n",super.toString(),this.getClass().getName(),this.getLongitudLado(),this.area());  
}
```

Y ahora en el método dibujar de ambas clases **Triangulo** y **Cuadrado**, cuando se ejecuta el método dibujar, en la instrucción **System.out.println(this);** se imprimiría, en cada uno de los casos:

**Referencia objeto: Triangulo@4b1210ee. Triangulo de lado: 1,00 y área: 0,43**

**Referencia objeto: Cuadrado@3f99bd52. Cuadrado de lado: 2,00 y área: 4,00**

**Sobrecargamos** un método cuando volvemos a escribir un método con el mismo nombre pero con distinto número **y/o** tipo de parámetros. Por ejemplo en el punto 7, sobrecargábamos al constructor de la clase Estudiante. Suele utilizarse la **sobrecarga** de métodos para establecer valores por defecto en los parámetros. Por ejemplo:

```
public double precioConIva(double precio, double iva) {  
    return precio+precio*iva/100;  
}  
public double precioConIva(double precio) {  
    return precioConIva(precio,21);  
}
```

Podemos usar el método **precioConIva** llamándolo con un o dos parámetros, en el caso de que omitamos el segundo parámetro, se calcula el precio con el iva del 21% por defecto.

## 14.- Super

Lo mismo que decíamos en un punto anterior sobre **this** podemos decir sobre **super** sólo que referido a la clase Padre del objeto actual. Puede aparecer para llamar o acceder a un miembro de la superclase (de ahí la palabra **super**) o para invocar al constructor de la superclase. Ejemplo:

```
//Fichero Persona.java
public class Persona {
    private String dni, nombre, apellidos;
    private int edad;

    public Persona(String dni, String nombre, String apellidos) {
        this.dni=dni;this.nombre=nombre; this.edad=edad; this.apellidos=apellidos;
    }
    public Persona(String dni, String nombre, String apellidos, int edad) {
        this(dni,nombre,apellidos);
        this.edad=edad;
    }
    public String toString() {
        return String.format("[Dni: %s, Nombre: %s, Apellidos: %s, Edad: %d]",dni,nombre,apellidos,edad);
    }
}

//Fichero Empleado.java
public class Empleado extends Persona {
    private int numEmp;
    public Empleado(String dni, String nombre, String apellidos, int edad, int numEmp) {
        super(dni,nombre,apellidos,edad); // al ser dni, nombre, apellidos, edad
        private en el padre es la única forma de inicializarlas
        this.numEmp=numEmp;
    }
    public String toString() {
        String s=super.toString();
        return s.replace("]",",", NumEmp: "+numEmp+"]");
    }
    public int getNumEmp() {
        return this.numEmp;
    }
}

//Fichero Main.java
public class Main {
    public static void main(String[] args) {
        Empleado e=new Empleado("23456789D","María","Pérez Solís",33,1233);
        System.out.println(e);
    }
}
```

La ejecución del programa imprimirá:

**[Dni: 23456789D, Nombre: María, Apellidos: Pérez Solís, Edad: 33, NumEmp: 1233]**

Vemos en el ejemplo que el constructor de **Empleado** comienza llamando al constructor de **Persona** del que hereda, haciendo uso de **super(dni,nombre,apellidos,edad);**, y además es la única forma que tiene de poder inicializar sus variables miembro heredadas de la clase **Persona** ya que éstas son **private**. En el método **toString()** de **Empleado** hacemos uso de **super.toString();** para llamar al método **toString()** de la clase **Persona**.

## 15.- Casting

En los primeros temas ya hablamos del casting, sólo que referido a los tipos básicos de datos. Un casting es una conversión de un tipo a otro. Evidentemente ambos tienen que ser compatibles. Por ejemplo:

**double pi=Math.PI;**

**int entPi=(int)pi; //se convierte el valor de la variable pi a entero, resultado 3, ambos tipos son compatibles**

Al igual que entre tipos de datos básicos también entre objetos se puede hacer casting, pero sólo si ambos tipos son compatibles. Dos clases de objetos son compatibles si una hereda de la otra.

Siguiendo con el ejemplo del apartado anterior de las clases **Persona** y **Empleado**, los objetos de ambas clases son compatibles y por lo tanto pueden ser convertidos mediante casting, pero un objeto de la superclase no puede ser convertido a un objeto de la subclase, si al revés.

```
Persona p1=new Persona("23456789D", "María", "Pérez Solís",33);
Persona p2=new Empleado("12345678Z", "Juan", "Martínez Pérez",32,1233);
Empleado e1=new Empleado("34567890V", "Alicia", "Alvárez Bermúdez",25,1234);
Persona p3;
```

Mediante p2 sólo podremos acceder a los métodos implementados en la clase **Persona**, teniendo en cuenta que cuando hay un método polimórfico se ejecuta el método real del objeto, por ejemplo si ejecutamos **p2.toString()** se ejecutará el método toString() de **Empleado**. Pero no podremos llamar a **p2.getNumEmp()**, ya que éste no aparece en la clase **Persona**, y p2 ha sido declarado como objeto de la clase **Persona**, aunque realmente es un objeto de la clase **Empleado**. Aún así, y sabiendo que p2 en realidad es un objeto de la clase **Empleado**, podemos llamar al método mencionado de la siguiente forma:

```
((Empleado) p2).getNumEmp()
```

Estamos haciéndole un *casting* a p2 a su verdadera clase.

También podemos aplicarle un casting al objeto e1 de la clase **Empleado** a su superclase:

```
p3=(Persona) e1;
```

Aunque no es necesario hacerlo ya que todo **Empleado** es una **Persona**, por lo tanto podemos hacer:

```
p3=e1;
```

El *casting* es implícito en este caso, ya que son tipos de datos compatibles. Al igual que pasaría si asignamos un **int** a una variable tipo **float**:

```
int a=5;
```

```
float b=a;
```

## 16.- Trabajando con fechas, java.time

A partir del JDK 8 hay disponibles nuevas clases para el manejo de fechas que sustituyen a la obsoleta clase **java.util.Date**. Ahora disponemos de la clase **java.time.LocalDate** que hace mucho más llevadero trabajar con fechas que con la clase **java.util.Date**. Veamos un ejemplo:

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;
LocalDate hoy=LocalDate.now();
LocalDate fechaNac=LocalDate.of(1995,8,6);
long numDias=ChronoUnit.DAYS.between(fechaNac,hoy);
System.out.printf("Has vivido: %d días. ¡Que vivas muchos más!\n",numDias);
Imprimirá (dependiendo del día que se ejecute):
Has vivido: 9041 días. ¡Que vivas muchos más!
```

En el ejemplo:

**LocalDate.now()** es un método estático que nos devuelve un objeto **LocalDate** con la fecha de hoy.

**LocalDate.of(1995,8,6)** es un método estático que nos devuelve un objeto **LocalDate** con la fecha "06/08/1995".

**java.time.temporal.ChronoUnit** es un tipo enumerado que contiene objetos que implementan la interface **TemporalUnit** en la cual se declaran entre otros el método **between** al que le hemos pasado dos fechas y nos ha devuelto el número de días entre ambos, haciendo uso del objeto **DAYS**, podemos trabajar con las *unidades temporales* **CENTURIES**, **DECADES**, **YEARS**, **MONTHS**, **DAYS**, **HOURS**, **MINUTES**, **SECONDS**, **MILIS**, **NANOS**, entre otras.

Otros métodos interesantes de **LocalDate**:

```
jshell> LocalDate ahora=LocalDate.now()
ahora ==> 2020-05-07
jshell> ahora.plusDays(5)
$17 ==> 2020-05-12
jshell> ahora.minusDays(5)
$18 ==> 2020-05-02
jshell> ahora.plusMonths(2)
$19 ==> 2020-07-07
jshell> ahora.minusMonths(2)
$20 ==> 2020-03-07
jshell> ahora.plusYears(2)
$21 ==> 2022-05-07
jshell> ahora.minusYears(2)
$22 ==> 2018-05-07
```

Ejercicio. Implementa un método que reciba la fecha de nacimiento de una persona y diga cuantos años, meses, días, horas y segundos han transcurrido hasta el momento en que se ha ejecutado el método.

## **NOTAS**

- Documentación última versión del JDK:

<https://docs.oracle.com/en/java/javase/17/docs/api/index.html>

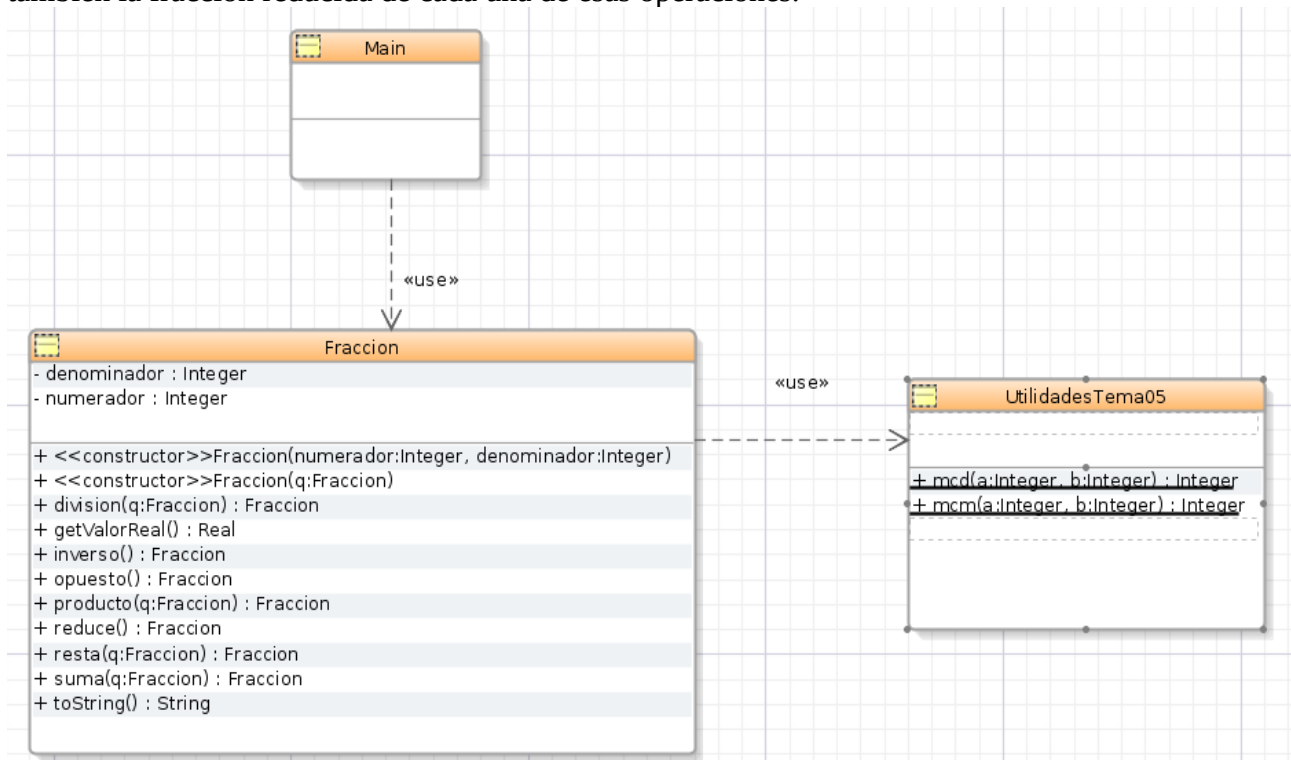
### Boletín Ejercicios

1) Implementar la clase Fraccion que representa un número racional, p.ej.: 2/3.

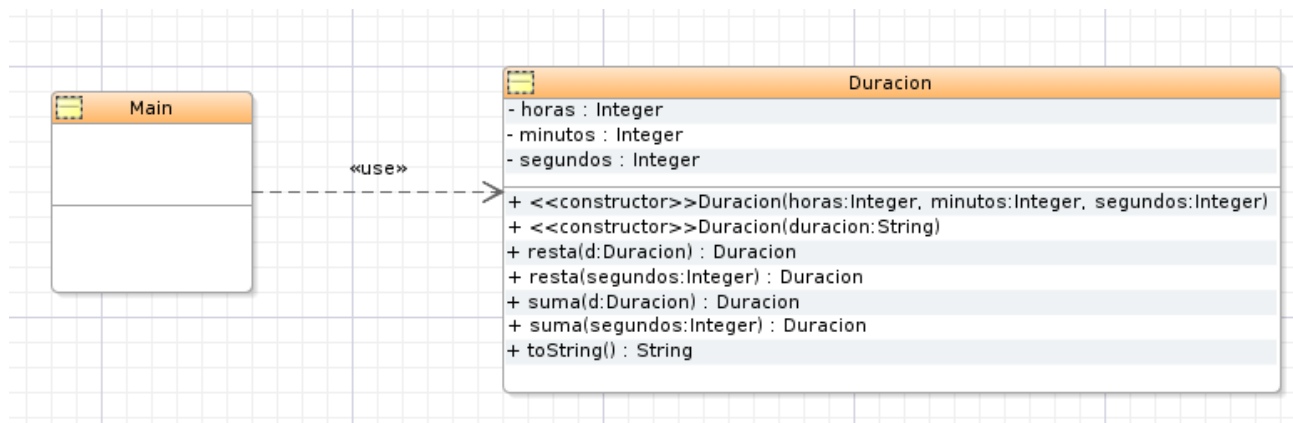
La clase **Fraccion** implementa las operaciones **suma**, **resta**, **multiplicación** y **división** de fracciones, para la suma y la resta es conveniente usar el método **mcm** (mínimo común múltiplo (usar el método de euclides)) para poner ambas fracciones en un denominador común. El método **getValorReal** nos devuelve el número double resultado de dividir *numerador/denominador*. El método **reduce** nos devuelve una nueva fracción reducida de la del objeto de que es llamado, p.ej. la fracción reducida de 4/16 sería 1/4. Y el método **toString** nos devuelve una cadena que represente a la fracción (p.ej.: 1/5).

La clase UtilidadesTema05 es una clase de utilidad que implementa los métodos estáticos **mcd** (máximo común divisor) y **mcm** (mínimo común múltiplo)

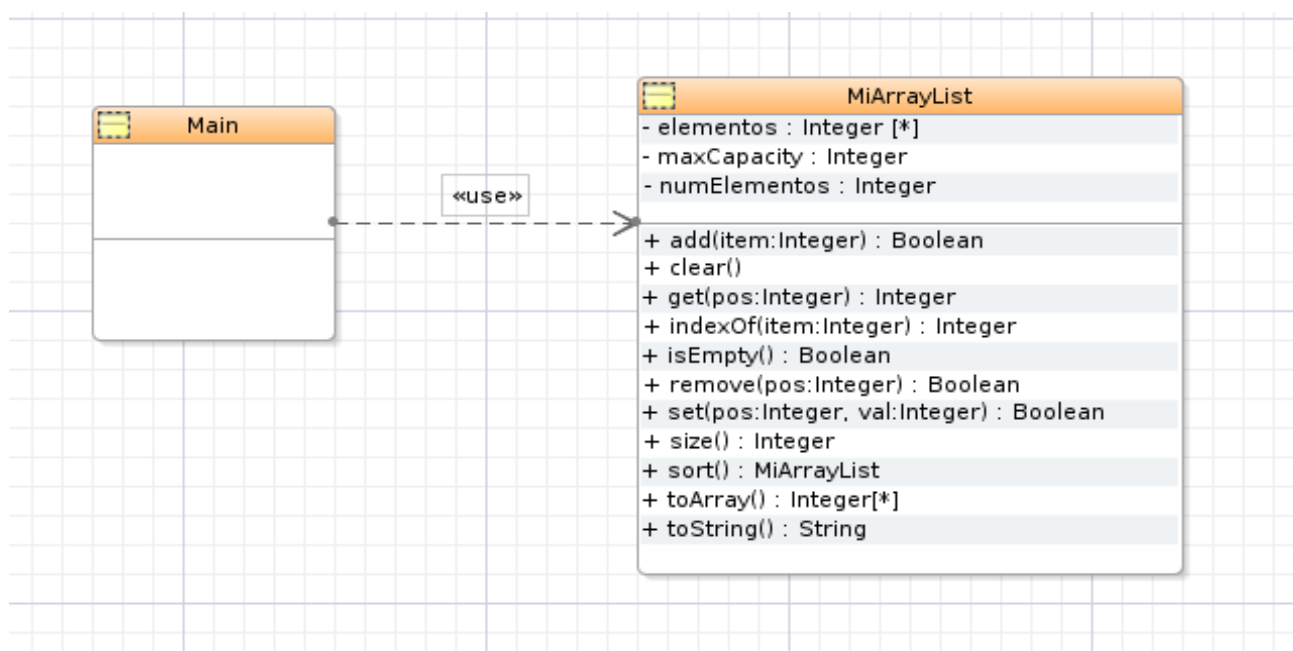
La clase **Main** la utilizaremos para poner a prueba a la clase Fraccion. Crea en ella dos objetos de tipo Fraccion, p.ej. 3/7 y 2/3. Calcula e imprime su **suma**, **resta**, **producto**, **división**. Imprime también la fracción reducida de cada una de esas operaciones.



2) Crea la clase **Duracion** con los métodos **suma** y **resta**. Los objetos de la clase **Duracion** son intervalos de tiempo y se crean de la forma **Duracion d = new Duracion(1, 20, 30)** o **t = new Duracion("01:20:30")**, donde los parámetros que se le pasan al constructor son las horas, los minutos y los segundos respectivamente, o una cadena con las horas, los minutos y los segundos cada uno con 2 dígitos separados por ":". Crea el método **toString** para ver los intervalos de tiempo de la forma 01:20:30. Si se suman por ejemplo 00:30:40 y 00:35:20 el resultado debería ser 01:06:00. Sobrecarga los métodos **suma** y **resta** de forma que también se le pueda sumar a un objeto de la clase **Duración** una cantidad de segundos. Realiza una clase **Main** de prueba para comprobar que la clase funciona bien.



3) Crear una clase **MiArrayList** que almacene una lista de 100 enteros como máximo. Deberá tener las operaciones **add** añadir un nuevo entero, **remove** borrar el entero de la posición que se le pase, **get** obtener el entero de la posición que se le pase, **sort** que devolverá un objeto **MiArrayList** ordenado, **size** que devolverá el número de elementos almacenados, **clear** que borrará todos los elementos de la lista, **indexOf** que devolverá el índice de la primera ocurrencia de un elemento en la lista o -1 si no se encuentra, **isEmpty** que devolverá **true** si la lista está vacía o **false** en caso contrario, **set** que cambia el valor de la posición que se le indique, **toArray** que devolverá un array con los elementos actualmente en la lista, **toString** que devolverá una cadena representando la lista de números.





4) Se quiere informatizar una biblioteca. Crea las clases Publicacion, Libro y Revista. Las clases deben estar implementadas con la jerarquía correcta. Las características comunes de las revistas y de los libros son el título, y el año de publicación. Los libros tienen además un isbn y un atributo prestado. Cuando se crean los libros, no están prestados. Las revistas tienen un issn y un número. La clase Libro debe implementar la interfaz Prestable que tiene los métodos presta, devuelve y estaPrestado. Haz primero el diagrama de clases.

Programa principal:

```
Libro libro1 = new Libro("123456", "La Ruta Prohibida", 2007);
Libro libro2 = new Libro("112233", "Los Otros", 2016);
Libro libro3 = new Libro("456789", "La rosa del mundo", 1995);
Revista revista1 = new Revista("444555", "Año Cero", 2019, 344);
Revista revista2 = new Revista("002244", "National Geographic", 2003, 255);
System.out.println(libro1);
System.out.println(libro2);
System.out.println(libro3);
System.out.println(revista1);
System.out.println(revista2);
libro2.presta();
if (libro2.estaPrestado()) {
    System.out.println("El libro está prestado");
}
libro2.presta();
libro2.devuelve();
if (libro2.estaPrestado()) {
    System.out.println("El libro está prestado");
}
libro3.presta();
System.out.println(libro2);
System.out.println(libro3);
Salida:
ISBN: 123456, título: La Ruta Prohibida, año de publicación: 2007 (no prestado)
ISBN: 112233, título: Los Otros, año de publicación: 2016 (no prestado)
ISBN: 456789, título: La rosa del mundo, año de publicación: 1995 (no prestado)
ISBN: 444555, título: Año Cero, año de publicación: 2019
ISBN: 002244, título: National Geographic, año de publicación: 2003
El libro está prestado
Lo siento, ese libro ya está prestado.
ISBN: 112233, título: Los Otros, año de publicación: 2016 (no prestado)
ISBN: 456789, título: La rosa del mundo, año de publicación: 1995 (prestado)
```

#### 4) Partiendo de un documento XML como el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<cds>
  <cd>
    <titulo>
      Nevermind
    </titulo>
    <año>
      1991</año>
    <artista>
      Nirvana
    </artista>
    <cancion>
      <duracion>00:05:01</duracion>
      <titulo>Smells Like Teen Spirit</titulo>
    </cancion>
    <cancion><duracion>00:04:14</duracion><titulo>In Bloom</titulo></cancion>
    <cancion><duracion>00:03:39</duracion><titulo>Come As You Are</titulo></cancion>
    <cancion><duracion>00:03:03</duracion><titulo>Breed</titulo></cancion>
    <cancion><duracion>00:04:17</duracion><titulo>Lithium</titulo></cancion>
    <sello>DGC Records Virgin Records</sello>
  </cd>
  <cd>
    <titulo>Demon Days</titulo>
    <año>2005</año>
    <artista>Nirvana</artista>
    <cancion><duracion>00:01:03</duracion><titulo>Intro</titulo></cancion>
    <cancion><duracion>00:03:10</duracion><titulo>Last Living Souls</titulo></cancion>
    <cancion><duracion>00:04:31</duracion><titulo>Kids With Guns</titulo></cancion>
    <cancion><duracion>00:03:43</duracion><titulo>0 Green World</titulo></cancion>
    <cancion><duracion>00:04:53</duracion><titulo>Dirty Harry</titulo></cancion>
    <sello>Parlophone Virgin Records</sello>
  </cd>
  <cd>
    <titulo>Goblin</titulo>
    <año>2011</año>
    <artista>Tyler, The Creator</artista>
    <cancion><duracion>00:04:11</duracion><titulo>Goblin</titulo></cancion>
    <cancion><duracion>00:07:18</duracion><titulo>Yonkers</titulo></cancion>
    <cancion><duracion>00:04:13</duracion><titulo>Radicals</titulo></cancion>
    <cancion><duracion>00:03:12</duracion><titulo>She</titulo></cancion>
    <cancion><duracion>00:05:22</duracion><titulo>Transylvania</titulo></cancion>
    <sello>XL Recordings</sello>
  </cd>
  <cd>
    <titulo>Madvillainy</titulo>
    <año>2004</año>
    <artista>MF Doom Madlib</artista>
    <cancion><duracion>00:01:55</duracion><titulo>The Illest Villains</titulo></cancion>
    <cancion><duracion>00:01:58</duracion><titulo>Accordion</titulo></cancion>
    <cancion><duracion>00:02:11</duracion><titulo>Meat Grinder</titulo></cancion>
    <cancion><duracion>00:02:35</duracion><titulo>Bistro</titulo></cancion>
    <cancion><duracion>00:03:54</duracion><titulo>Raid</titulo></cancion>
    <sello>Stones Throw Records</sello>
  </cd>
  <cd>
    <titulo>My Beautiful Dark Twisted Fantasy</titulo>
    <año>2010</año>
    <artista>Kanye West</artista>
    <cancion><duracion>00:04:40</duracion><titulo>Dark Fantasy</titulo></cancion>
    <cancion><duracion>00:05:57</duracion><titulo>Gorgeous</titulo></cancion>
    <cancion><duracion>00:04:52</duracion><titulo>Power</titulo></cancion>
    <cancion><duracion>00:01:02</duracion><titulo>All Of The Lights (Interlude)</titulo></cancion>
    <cancion><duracion>00:04:59</duracion><titulo>All Of The Lights</titulo></cancion>
    <sello>Roc-A-Fella Records</sello>
  </cd>
  <cd>
    <titulo>Random Access Memories</titulo>
    <año>2013</año>
    <artista>Daft Punk</artista>
    <cancion><duracion>00:04:35</duracion><titulo>Give Life Back To Music</titulo></cancion>
    <cancion><duracion>00:05:22</duracion><titulo>The Game Of Love</titulo></cancion>
    <cancion><duracion>00:09:04</duracion><titulo>Giorgio By Moroder</titulo></cancion>
    <cancion><duracion>00:03:48</duracion><titulo>Within</titulo></cancion>
    <cancion><duracion>00:05:37</duracion><titulo>Instant Crush</titulo></cancion>
    <sello>Columbia Sony Music</sello>
  </cd>
  <cd>
    <titulo>Good Kid, M.A.A.d City</titulo>
    <año>2012</año>
    <artista>Kendrick Lamar</artista>
    <cancion><duracion>00:03:48</duracion><titulo>Sherane a.k.a Master Splinter's Daughter</titulo></cancion>
    <cancion><duracion>00:05:37</duracion><titulo>Bitch, Don't Kill My Vibe</titulo></cancion>
    <cancion><duracion>00:05:53</duracion><titulo>Backseat Freestyle</titulo></cancion>
    <cancion><duracion>00:08:18</duracion><titulo>The Art Of Peer Pressure</titulo></cancion>
    <cancion><duracion>00:06:09</duracion><titulo>Money Trees</titulo></cancion>
  </cd>
</c>
```

```

<sello>Top Dawg Entertainment</sello>
</cd>
<cd>
<titulo>XX</titulo>
<año>2009</año>
<artista>The XX</artista>
<cancion><duracion>00:02:08</duracion><titulo>Intro</titulo></cancion>
<cancion><duracion>00:02:57</duracion><titulo>VCR</titulo></cancion>
<cancion><duracion>00:02:41</duracion><titulo>Crystalised</titulo></cancion>
<cancion><duracion>00:04:02</duracion><titulo>Islands</titulo></cancion>
<cancion><duracion>00:03:34</duracion><titulo>Heart Skipped A Beat</titulo></cancion>
<sello>XL Recordings Young Turks</sello>
</cd>
<cd>
<titulo>The Suburbs</titulo>
<año>2010</año>
<artista>Arcade Fire</artista>
<cancion><duracion>00:05:15</duracion><titulo>The Suburbs</titulo></cancion>
<cancion><duracion>00:04:16</duracion><titulo>Ready To Start</titulo></cancion>
<cancion><duracion>00:04:40</duracion><titulo>Modern Man</titulo></cancion>
<cancion><duracion>00:03:57</duracion><titulo>Rococo</titulo></cancion>
<cancion><duracion>00:02:52</duracion><titulo>Empty Room</titulo></cancion>
<sello>Merge Records Sonovox Records</sello>
</cd>
<cd>
<titulo>In Rainbows</titulo>
<año>2008</año>
<artista>Radiohead</artista>
<cancion><duracion>00:03:57</duracion><titulo>15 Step</titulo></cancion>
<cancion><duracion>00:04:02</duracion><titulo>Bodysnatchers</titulo></cancion>
<cancion><duracion>00:04:15</duracion><titulo>Nude</titulo></cancion>
<cancion><duracion>00:05:18</duracion><titulo>Weird Fishes/Arpeggi</titulo></cancion>
<cancion><duracion>00:03:49</duracion><titulo>All I Need</titulo></cancion>
<sello>TBD Records RED Distribution Virgin Records</sello>
</cd>
</cdis>

```

Se desea implementar una aplicación que procese el documento, cree una representación de objetos del mismo (D.O.M - Document Objet Model) e imprima una tabla con toda la información de cada cd contenido en el mismo. Partiremos primero del diagram a de clases y a partir de él se hará la implementación haciendo los sucesivos refinamientos (retroalimentación) que hicieran falta.