

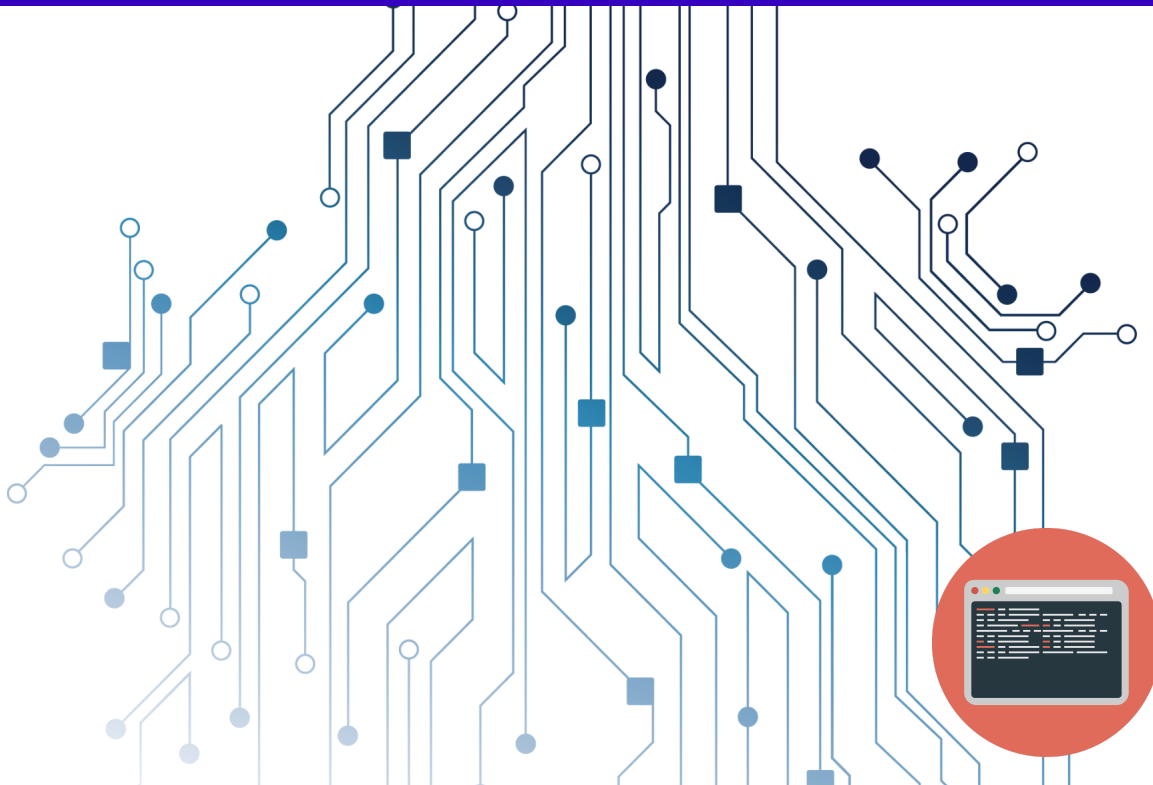


CFGS APLICACIONES MULTIPLATAFORMA - PROGRAMACIÓN



UNIDAD 3

PROGRAMACIÓN ORIENTADA A OBJETOS (POO)



DIEGO VALERO ARÉVALO

BASADO EN APUNTES Y EJERCICIOS PROPORCIONADOS POR
M^º CARMEN DÍAZ GONZÁLEZ - IES VIRGEN DE LA PALOMA

U3 - PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

ÍNDICE

3.1 - INTRODUCCIÓN A LA POO	1
>> 3.1.1- INTRODUCCIÓN	1
· Partes de un objeto	1
>> 3.1.2- TRABAJANDO CON CLASES	1
3.1.2.1 - Definición de clases	1
3.1.2.2 - Instanciación de objetos	2
>> 3.1.3- GARBAGE COLLECTOR	3
>> 3.1.4- ACCESO A CLASES	3
· public, private, protected.	
3.1.4.1 - Principio de Encapsulación	4
>> 3.1.5- CONSTRUCTORES	4
>> 3.1.6- RESTRINGIR LA MODIFICACIÓN DE DATOS	6
>> 3.1.7- ATRIBUTOS COMUNES ENTRE OBJETOS DE LA MISMA CLASE	7
3.1.7.1 - Modificador static	7
3.1.7.2 - Modificador final (constantes)	8
>> 3.1.8- DATOS ENUMERADOS	8
>> 3.1.9- PRINCIPIOS BÁSICOS DE LA POO	9
· Abstracción, encapsulación, modularidad, jerarquía.	
3.2 - DESARROLLO DE CLASES	10
>> 3.2.1- PASO DE PARÁMETROS A MÉTODOS	10
· Paso por valor y referencia.	10
>> 3.2.2- ÁMBITO DE ATRIBUTOS Y VARIABLES	11
· Global, local.	11
>> 3.2.3- SOBRECARGA DE MÉTODOS	12
3.3 - PAQUETES Y LIBRERÍAS	13
>> 3.2.1- CREAR LIBRERÍAS	13
>> 3.2.2- IMPORTAR LIBRERÍAS	13

U3 - BATERÍA DE EJERCICIOS**15**

Parte 1

15

Parte 2

16

Parte 3

16

Ejercicios complementarios

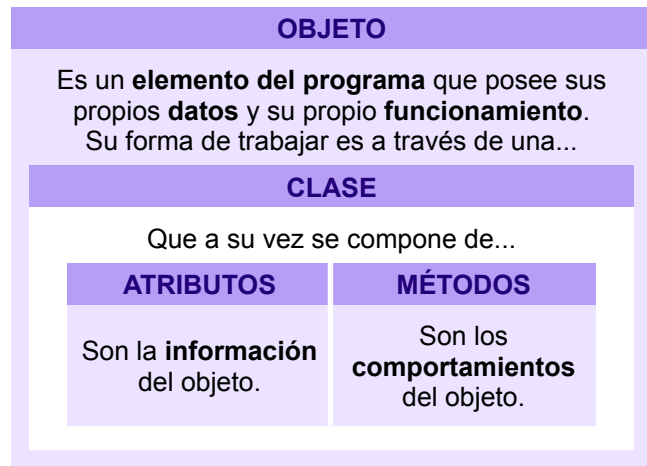
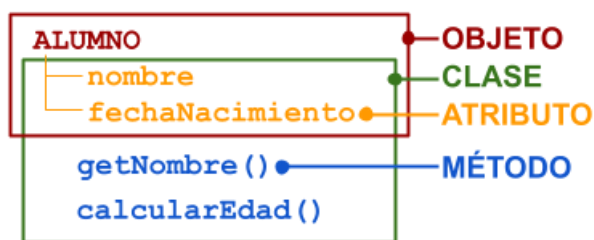
17

3.1 - INTRODUCCIÓN A LA POO

>> 3.1.1- INTRODUCCIÓN

La **programación orientada a objetos (POO)** hace referencia al tipo de programación que trata de almacenar **objetos reales de manera virtual**. Nos permite **dividir un problema en diferentes partes** que colaboran entre sí o pueden ser independientes para lograr una solución.

Estas partes se llaman **objetos**, y su estructura es la siguiente:



Antes de crear un objeto, debemos **definir su clase**, ya que esta **define el tipo de objeto**.

>> 3.1.2- TRABAJANDO CON CLASES

3.1.2.1- DEFINICIÓN DE CLASES

Una **clase** es un **archivo .java** que contiene los **atributos** y **métodos** del objeto que queremos definir. Esta es como un “molde” del que sacamos copias, cada una con sus propios datos pero respetando la estructura principal, por eso es necesario definir primero este “molde” antes de poder construir cualquier otra cosa.

La **estructura base** de una clase es la siguiente:

```
public class nombreClase{  
  
    tipo atributo;  
  
    void setAtrib(tipo parametro){  
        atributo = parametro;  
    }  
  
    tipo getAtrib(tipo parametro){  
        return atributo;  
    }  
  
    void metodo(){  
        ...  
    }  
}
```

```
public class nombreClase{}
```

Cabecera de la clase.

```
tipo atributo;
```

Declaración de variables internas (**atributos**).

```
void setAtrib(tipo parametro){
    [this.]atributo = parametro;
}
```

Método **set**. Los métodos que comienzan con este prefijo se usan para **asignar un valor a un atributo específico**.



Podemos perfectamente llamar al parámetro con el mismo nombre que el atributo de la clase, pero para que el programa no entre en conflicto, el atributo debe tener el operador **this..**

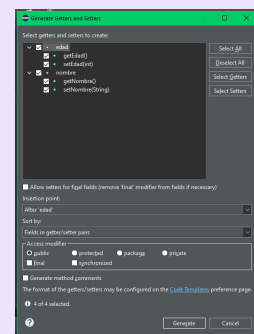
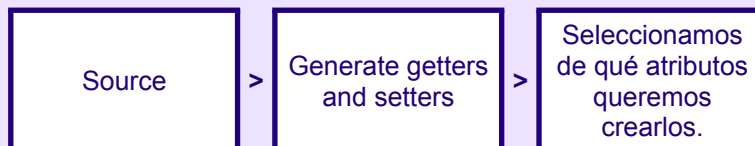
```
tipo getAtrib(tipo parametro){
    return atributo;
}
```

Método **get**. Los métodos que comienzan con este prefijo se usan para **devolver el valor de un atributo específico**.

```
void metodo() {
    ...
}
```

Un **método común** que podemos crear para la clase, que ya los conocemos. La clase puede tener todos los que queramos implementar.

Eclipse permite generar todos los **setters** y **getters** de una clase del tirón. Para ello vamos a:



3.1.2.2- INSTANCIACIÓN DE OBJETOS

Una vez hemos generado una clase, ya podemos crear un objeto a partir de esta. Debemos tener en cuenta que los objetos son **tipos de datos** y tienen sus propiedades igual que cualquier otro tipo.

```
nombreClase nombreObjeto = new nombreClase();
```

nombreClase

El tipo de nuestro objeto será la clase que lo definirá.

nombreObjeto

Asignamos un nombre al mismo.

new

El operador **new** asigna durante la ejecución del programa un **espacio de memoria al objeto**, y almacena esta **referencia** en una variable.

nombreClase();

La clase en la que se va a basar el objeto.

Si queremos **dar valores a los atributos del nuevo objeto**, podemos acceder a él a través del signo **.**:

```
nombreObjeto.atrib = valor;
```

Aunque lo más recomendable es hacerlo a través de los métodos **set**:

```
nombreObjeto.setAtrib(valor);
```

Veamos un ejemplo de una definición de clase e instanciación de objetos:

EJEMPLO	
Persona.java	PersonaMain.java
<pre> 1 public class Persona{ 2 3 String nombre; 4 int edad; 5 6 void setNombre(String nombre){ 7 this.nombre = nombre; 8 } 9 10 void setEdad(int edad){ 11 this.edad = edad; 12 } 13 14 String getNombre(){ 15 return nombre; 16 } 17 18 String getEdad(){ 19 return edad; 20 } 21 22 boolean esMayorEdad(){ 23 return (edad>=18); 24 } 25 }</pre>	<pre> public class PersonaMain{ public static void main(String[] args){ Persona p1 = new Persona(); p1.setNombre("Diego"); p1.setEdad(25); System.out.println(p1.getNombre); System.out.println("¿Es mayor de edad?"); if(p1.esMayorEdad()){ System.out.print(" Sí"); } else{ System.out.print(" No"); } } }</pre>
	<div>Console</div> <pre> Diego ¿Es mayor de edad? Sí</pre>

>> 3.1.3- GARBAGE COLLECTOR

En Java hay un **recolector de basura** (*garbage collector*) que **se encarga de gestionar los objetos que se dejan de usar y eliminarlos de memoria**. Este proceso es **automático e impredecible** y trabaja en un hilo (*thread*) de baja prioridad.

Por lo general ese proceso de recolección de basura trabaja cuando detecta que **hace demasiado tiempo que un objeto que no se utiliza en un programa**. Esta eliminación depende de la máquina virtual de Java, en casi todas las recolecciones se realiza periódicamente en un determinado lapso de tiempo.

Si queremos decirle al *garbage collector* que elimine una variable que no nos interesa, **debemos asignarle un valor null** para que la detecte y este pueda liberar recursos.

>> 3.1.4- ACCESO A CLASES

Lo óptimo es decidir y dejar claro **a qué se puede acceder y a qué no** de una clase desde el exterior. Para ello definimos la visibilidad de los atributos y métodos con los siguientes **especificadores**:

public	private	protected
Se puede acceder desde cualquier clase .	Sólo se puede acceder desde la misma clase .	Se puede acceder desde la propia clase, desde las subclases que heredan de ella y desde aquellas que estén en el mismo paquete .

Aquellas clases que tienen miembros **public** se consideran **interfaces**, ya que permiten el acceso desde fuera.

3.1.4.1- PRINCIPIO DE ENCAPSULACIÓN

Uno de los principios de la POO es la **encapsulación**, que dice que debemos aplicar el especificador **public** a las funciones miembro que formen la **interfaz pública** y **denegar el acceso a los datos miembro** usados por esas funciones mediante el especificador **private**.

Por eso lo recomendable es definir los atributos de una clase como **private** para que **no se puedan modificar directamente** y la única manera sea **a través de los métodos disponibles**. Así podemos **restringir** la modificación de ciertos datos que no nos interesa que se puedan cambiar.

EJEMPLO	
Persona.java	PersonaMain.java
<pre>1 public class Persona{ 2 3 String nombre; 4 5 void setNombre(String nombre){ 6 this.nombre = nombre; 7 } 8 }</pre>	<pre>public class PersonaMain{ public static void main(String[] args){ Persona p1 = new Persona(); p1.setNombre("Diego"); p1.nombre("Juan"); } }</pre>
Persona.java	PersonaMain.java
<pre>1 public class Persona{ 2 3 private String nombre; 4 5 public void setNombre(String nombre){ 6 this.nombre = nombre; 7 } 8 }</pre>	<pre>public class PersonaMain{ public static void main(String[] args){ Persona p1 = new Persona(); p1.setNombre("Diego"); p1.nombre("Juan"); } }</pre>

>> 3.1.5- CONSTRUCTORES

Imagina que tienes ocho atributos en una clase. Sería muy rollo ir añadiendo valores a cada atributo por separado, por eso las clases permiten crear **constructores**, que son **métodos especiales que permiten añadir valores a estos al crear el objeto**. No devuelven ningún valor pero siempre devuelven **referencias** a una instancia de la clase. Tenemos **tres tipos de valores** que podemos dar en un constructor:

VALOR POR DEFECTO	VALOR INICIAL	VALOR POR PARÁMETRO
<pre>public nombreClase(){ atributo = null; }</pre>	<pre>public nombreClase(){ atributo = valor; }</pre>	<pre>public nombreClase(tipo parametro){ atributo = parametro; }</pre>
El valor por defecto es 0 para números y null para el resto, y es el que da JAVA a cada atributo si no se especifica un valor. No es necesario escribirlos.	Podemos añadir valores iniciales a un atributo si no se proporcionan parámetros en el constructor.	También podemos añadir valores pasándolos como parámetros .

Es muy recomendable que los valores iniciales se den siempre en los **constructores** y no como inicialización del atributo:



✗ `private tipo atributo = valor;`

✓

```
public nombreClase(){
    atributo = valor;
}
```

La **única restricción** es que los constructores **deben llamarse igual que la clase**.

EJEMPLO

Persona.java

```
1 public class Persona{
2
3     private String nombre;
4     private int edad;
5
6     public Persona(String nombre, int edad){ → Esto es un constructor
7         this.nombre = nombre;
8         this.edad = edad;
9     }
10 }
```

Si tenemos un constructor con parámetros pero no necesitamos pasarle todos los datos que se piden, podemos añadir **constructores por defecto**, que dependerán del **constructor principal** (aquel que **asigna todos o la mayoría de valores de atributos por parámetros**) pero que no hará falta usar al instanciar el objeto.

```
public class nombreClase{

    private tipo atributo1;
    private tipo atributo2;

    public nombreClase(tipo parametro1, tipo parametro2){ → Esto es un constructor principal
        atributo1 = parametro1;
        atributo2 = parametro2;
    }

    public nombreClase(tipo parametro1){ → Esto es un constructor por defecto
        this(parametro1, valor);
    }
}
```

El constructor por defecto utiliza el especificador `this()`, en el cual se introducen, **en el orden de la asignación de valores del constructor principal**, los valores que va a tomar cada uno.

EJEMPLO

Persona.java

```
1 public class Persona{
2
3     private String nombre;
4     private int edad;
5
6     public Persona(String nombre, int edad){
7         this.nombre = nombre;
8         this.edad = edad;
9     }
10
11     public Persona(int edad){
12         this("Sin nombre", edad);
13     }
14 }
```

¿Es obligatorio hacer un constructor principal con todos los atributos de la clase? No, aunque es raro que no se haga, ya que si sobran atributos en el constructor, es probable que el objeto no los use y por tanto no sean necesarios.

TEN EN CUENTA



Sólo se podrán dejar datos sin añadir para que sean dados por defecto **si ese constructor existe**. En el ejemplo anterior, no podríamos crear un objeto `Persona` con **sólo el nombre** porque no tenemos un constructor que lo asigne junto a una edad por defecto.

Podemos tener **todos los constructores que necesitemos**, pero ten en cuenta que no se pueden repetir aquellos que tengan el mismo número de parámetros y que sean todos del mismo tipo.

EJEMPLO

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

✗ public Persona(String nombre){
    this(nombre, "sin apellido", 12);
}

public Persona(String apellido){
    this("sin nombre", apellido, 12);
}
```

Da igual que las instrucciones sean diferentes, en este ejemplo, como los constructores **tienen un solo dato y del mismo tipo**, si construimos un objeto dando este tipo en los parámetros, es lógico que el programa no sepa a cuál de los dos llamar.

Los constructores por defecto son una manera de **sobrecargar métodos**, algo que veremos más en profundidad en la **unidad 4, Continuación de POO**.

>> 3.1.6- RESTRINGIR LA MODIFICACIÓN DE DATOS

Ahora que conocemos los **objetos** y los **constructores**, vamos a plantear lo siguiente: Imagina que quieres crear un objeto pero no quieres que exista ninguna manera de poder modificar ciertos datos, pongamos por ejemplo que se pide la información de una persona pero una vez introducido, por ejemplo, el DNI, es un dato que no queremos que se cambie. Ya hemos visto los especificadores `private` que hacen que sólo se puedan acceder a esos atributos desde la misma clase, pero ¿cómo prevenir su acceso aún más?

Es tan sencillo como **no poner métodos set** en la clase de los datos que nos interesa.

EJEMPLO

Persona.java

```
1 public class Persona{
2
3     private String nombre;
4     private String dni;
5     private int edad;
6
7     public Persona(String nombre, String dni, int edad){
8         this.nombre = nombre;
9         this.dni = dni;
10        this.edad = edad;
11    }
12
13    public void setNombre(String nombre){
14        this.nombre = nombre;
15    }
16}
```

```

17     public void setEdad(int edad) {
18         this.edad = edad;
19     }
20
21     public void getNombre() {
22         return nombre;
23     }
24
25     public void getDni() {
26         return dni;
27     }
28
29     public void getEdad() {
30         return edad;
31     }
32 }

```

En esta clase, el único atributo que no se podría modificar es `dni`, ya que no hay manera de darle un valor salvo cuando se le llama en el constructor.

Lo normal en una clase es que encontremos tanto **constructores** como métodos `get` y `set` aparte de los **métodos** que necesite la clase, ya que así podemos:

- **Crear objetos** con atributos con valores propios y/o por defecto.
- **Modificar** los datos existentes (`set`).
- **Consultar** los datos que queramos (`get`).
- **Usar los datos del objeto** en los métodos para conseguir ciertos resultados.

Si **no incluimos ciertos métodos set** junto a **que los atributos sean private**, no habrá manera externa de hacer modificaciones donde no queramos.

>> 3.1.7- ATRIBUTOS COMUNES ENTRE OBJETOS DE LA MISMA CLASE

· 3.1.7.1- MODIFICADOR STATIC

Si queremos hacer que **el valor de un atributo o un método lo utilicen todos los objetos creados con la misma clase**, debemos usar el modificador `static`. Este modificador hace que aquello que lo lleva no sea una instancia del objeto, sino que forma parte de su **propio tipo**, por lo que no hace falta crear un objeto para acceder a ello. Se establecen como `public`.

```

public static tipo atributoComun;

public static tipo / void metodoComun() {
    ...
}

```

Para asignar un valor a un **atributo común** podemos acceder directamente a él a través del nombre de la clase. Este valor se aplica a todos los objetos sin necesidad de ir asignándolo uno por uno:

```

nombreClase.atributoComun = valor;

```

3.1.7.2- MODIFICADOR FINAL (CONSTANTES)

Podemos establecer **atributos con valores que no cambian**, llamados **constantes**. Para ello debemos añadirle los modificadores **static** y **final**. Las constantes se suelen nombrar en **mayúsculas** y se les asigna el valor en su declaración.

```
public static final tipo CONSTANTE = valor;
```

Para usar las constantes fuera de la clase debemos acceder a la **clase** donde se ha declarado seguida de punto (.) y el **nombre de la constante**. Dentro de la misma clase podemos usarla directamente como cualquier otro atributo.

FUERA DE LA CLASE	DENTRO DE LA CLASE
<code>nombreClase.CONSTANTE;</code>	<code>CONSTANTE;</code>

Tienes como ejemplo las constantes predefinidas en java como **PI** de la librería **Math**.

EJEMPLO	
Articulo.java	ArticuloMain.java
<pre>1 public class Articulo{ 2 3 public static final double IVA = 1.21; 4 5 public static double calcularIVA(double precio){ 6 return precio*IVA; 7 } 8 }</pre>	<pre>public class ArticuloMain{ public static void main(String[] args){ System.out.print(Articulo.calcularIVA(15)); } }</pre>
	Console
	18.25

>> 3.1.8- DATOS ENUMERADOS

Una manera de evaluar los datos asignados a un atributo es con las **listas de datos enumerados**. Cuando tenemos datos de este tipo sólo se aceptarán como valores aquellos definidos en la lista.

```
enum nombreLista{
    valorA,
    valorB
}
```

```
enum nombreLista{}
```

El **tipo** es **enum** y se acompaña del **nombre** y las **llaves**.

```
valorA,
valorB
```

Los **valores** que queremos que contenga la lista. Podemos añadir todos los que queramos. Se enumeran seguidos de ,. Podemos definir el elemento final con ; o no.

Su **declaración** como atributo es como cualquier otro, usando como **tipo** el **nombre de la lista**.

```
private nombreLista atributo;
```

Para **declarar el valor de un atributo enumerado**, se accede a través del **nombre de la lista**, punto (.) y el **valor registrado en la lista** que queramos darle.

```
nombreLista.valorListado;
```

Los enum se declaran **fuera** de la declaración de la clase, preferiblemente **antes** de la cabecera de la misma.

EJEMPLO

Pelicula.java

```
1 enum formatoPelicula{
2     DVD,
3     BluRay
4 }
5
6 public class Pelicula{
7     private int codigo;
8     private String titulo;
9     private formatoPelicula formato;
10    private double precio;
11
12    public Pelicula(int codigo, String titulo, formatoPelicula formato,
13                    double precio){
14        this.codigo = codigo;
15        this.titulo = titulo;
16        this.formato = formato;
17        this.precio = precio;
18    }
19 }
```

PeliculaMain.java

```
1 public class PeliculaMain{
2     public static void main(String[] args){
3         Pelicula peli = new Pelicula(1, "Inside Out", formatoPelicula.DVD, 7.50);
4     }
5 }
```

>> 3.1.9- PRINCIPIOS BÁSICOS DE LA POO

ABSTRACCIÓN

Definición de las características de los objetos y de sus diferencias con los demás. Se centra en **qué hace un objeto** más que en **cómo lo hace**.

Nos encontramos con la importancia de una **buena definición de la interfaz de una clase** que nos permita **modificar el cómo sin cambiar el qué**. La abstracción se puede interpretar como **qué hace una clase vista desde fuera**.

ENCAPSULACIÓN

Se refiere a **ocultar el estado de un objeto a los demás**. Los atributos que representan el estado del objeto **sólo serán accesibles mediante operaciones que presente el propio objeto**. No necesariamente todos los elementos del estado del objeto deberán poder consultarse y/o modificarse (**restricción de modificación de datos**).

MODULARIDAD

Permite **dividir el programa objetivo en diferentes partes más pequeñas e independientes entre sí**. Idealmente, cada parte (por ejemplo una clase) será **completamente independiente de otra** pero no siempre es posible. Es necesario tener en cuenta a la hora de diseñar la división en módulos de nuestro programa aumentar la independencia y mantener la cohesión interna de las clases. La cohesión es la **"cercanía"** conceptual de los componentes de una clase.

JERARQUÍA

Organización de las clases, separar el **"es"** del **"es parte de"**:

ES	ES PARTE DE
EJEMPLO Un coche es un vehículo y por lo tanto tiene las características concretas de un vehículo .	EJEMPLO Las ruedas forman parte de un vehículo .

3.2 - DESARROLLO DE CLASES

>> 3.2.1- PASO DE PARÁMETROS A MÉTODOS

Hasta ahora sabemos que JAVA tiene **dos tipos de variables**:

PRIMITIVAS	OBJETOS
Que almacenan un valor .	Que guardan la referencia a su dirección en la memoria .

Para utilizar estas variables como parámetros en un método, se puede hacer de **dos maneras**:

POR VALOR	POR REFERENCIA
Significa que se realiza una copia de la variable y esta es enviada al método y no la original, entonces todos los cambios realizados dentro del método sólo afectan a la copia actual .	Significa que la referencia o el puntero a la variable original son pasadas a los métodos y no el dato original .

JAVA **siempre pasa los argumentos por valor**, sea la variable que sea. Puedes comprobar que esto se cumple en el siguiente ejemplo:

EJEMPLO
<div>Ejemplo.java</div> <pre> 1 public class Ejemplo{ 2 public static void main(String[] args){ 3 int x = 1; 4 int y = 2; 5 6 System.out.println("-Valores antes de modificar-"); 7 System.out.println("x: "+x+", y: "+y); 8 9 modificarValores(x,y); 10 11 System.out.println("-Valores después de modificar-"); 12 System.out.println("x: "+x+", y: "+y); 13 } 14 15 public static void modificarValores(int x, int y){ 16 x = 100; 17 y = 200; 18 System.out.println("-Valores dentro del método-"); 19 System.out.println("x: "+x+", y: "+y); 20 } 21 } </pre> <div>Console</div> <pre> -Valores antes de modificar- x: 1, y: 2 </pre>

```
-Valores dentro del método-
x: 100, y: 200
-Valores después de modificar-
x: 1, y: 2
```

¿Qué nos demuestra esto? Que como lo que se pasan al método son las **copias** de las variables, cualquier cambio que se les haga, si no se devuelven y almacenan, **no afectará a las variables originales**.

Veamos qué ocurre si lo que le pasamos es un **objeto**:

EJEMPLO
<pre>Ejemplo.java 1 public class Ejemplo{ 2 public static void main(String[] args){ 3 Persona datosPersona = new Persona(); 4 5 System.out.println("Nombre antes de asignar: "+datosPersona.getNombre()); 6 7 datosPersona.setNombre("Juan"); 8 System.out.println("Nombre después de asignar: "+datosPersona.getNombre()); 9 10 modificaNombre(datosPersona); 11 System.out.println("Nombre después de método: "+datosPersona.getNombre()); 12 } 13 14 public static void modificaNombre(Persona datosPersona){ 15 datosPersona.setNombre("Ana"); 16 } 17 }</pre>
<pre>Console Nombre antes de asignar: null Nombre después de asignar: Juan Nombre después de método: Ana</pre>

¡Vaya! Aquí el método sí que ha afectado a la variable original. Esto es así porque lo que le estamos pasando al ser un **objeto** es la **referencia a la variable**, la cual dirige a la dirección guardada en la memoria y por tanto cuando accede a ella **puede modificar directamente el objeto original**.

>> 3.2.2- ÁMBITO DE ATRIBUTOS Y VARIABLES

El ámbito define el **alcance** (hasta dónde se puede usar) de una variable, o con más palabras, en **qué secciones de código estará disponible la misma**. Fuera de su ámbito, **no se puede acceder a ella** porque **no existe**. Tenemos **dos tipos** de ámbitos:

GLOBAL	LOCAL
<p>Son aquellas variables declaradas después de la cabecera de la clase, fuera de cualquier método.</p> <p>Su ámbito es global porque pueden utilizarse e inicializarse en cualquier parte del código.</p> <p>GLOBAL = ATRIBUTO</p>	<p>Aquellas variables que se declaran dentro de un método o bloque específico (como un bucle), restringen su ámbito al interior del mismo. No admiten modificadores.</p> <p>Estas se crean cuando se llama al método o se activa el bloque y se destruyen cuando finaliza.</p> <p>Las variables locales siempre deben inicializarse antes de usarse.</p> <p>LOCAL = MÉTODO / BLOQUE</p>

>> 3.2.3- SOBRECARGA DE MÉTODOS

Con la POO podemos tener en una misma clase **varios métodos con el mismo nombre** que pueden tener diferentes funciones dependiendo del **tipo de dato que se les pase**. A esto se le llama **sobrecarga de métodos**. Para que esto sea consistente, **todos deben devolver el mismo tipo de dato**, si no lo que tenemos son **tipos del mismo método**.

EJEMPLO
<div>Ejemplo.java</div> <pre>1 public class Ejemplo{ 2 public static void main(String[] args){ 3 4 muestra(); 5 muestra("Qué tal"); 6 muestra(12); 7 8 } 9 10 public static void muestra(){ 11 System.out.println("Hola"); 12 } 13 14 public static void muestra(String dato){ 15 System.out.println(dato); 16 } 17 18 public static void muestra(int dato){ 19 System.out.println(dato); 20 } 21 }</pre> <div>Console</div> <pre>Hola Qué tal 12</pre>

Como ves en el ejemplo, tenemos **tres métodos que se llaman igual** pero tratan **diferentes tipos de datos** para poder funcionar con cualquiera de los que le pasemos y tengamos sobrecargados, por eso no tenemos ningún error.

Por ejemplo, como vimos en la **unidad 3, Programación Orientada a Objetos**, los **constructores por defecto** son un tipo de sobrecarga. Podemos tener todas las sobrecargas que queramos, pero si recuerdas las limitaciones que vimos, al usar esta técnica:

No se pueden repetir aquellos que tengan **el mismo número de parámetros y que además sean del mismo tipo**, da igual que cambie el tipo del método.

EJEMPLO
<pre>public static void muestra() { System.out.println("Sin número"); } * public static void muestra(String dato) { System.out.println(dato); } public static void muestra(int dato) { System.out.println(dato); }</pre>

```
public static int muestra(int numero){ → Este método no se aceptaría, porque  

    return numero; aunque no lo parezca, su cabecera es igual que el anterior.  

}
```


3.3 - PAQUETES Y LIBRERÍAS

En muchos lenguajes de programación existe el concepto de **librerías**, que en general son **un conjunto de clases** (llamados **paquetes**) donde **cada una contiene ciertos métodos y atributos almacenados** a los que se pueden acceder y utilizar desde cualquiera de nuestras clases. Llevamos usando varias librerías de JAVA desde que empezamos a programar, como por ejemplo la línea `System.out.println()`; utiliza la clase `System` que pertenece a la librería `java.lang`, una de las librerías por defecto de JAVA.

Al igual que podemos usar las librerías por defecto que trae JAVA, también podemos **importar librerías externas** y **crear las nuestras propias**.

>> 3.3.1- CREAR LIBRERÍAS

✓  miLibreria

>  MetodosUtiles.java

>  OtrosMetodos.java

Si queremos tener a mano **ciertos métodos** que nos resultan útiles (como por ejemplo, métodos para operar dos números o crear un array) podemos **guardarlos en una clase** y estos a su vez en un **paquete**. De esta manera hemos creado una **librería** que podemos importar desde otras clases para acceder a sus métodos.

>> 3.3.2- IMPORTAR LIBRERÍAS

Para importar una librería debemos usar la palabra clave **import** seguido de la **ruta del paquete o clase** que queremos que esté presente en el proyecto.





```
import paquete. ... .Clase;
```

Podemos importar todas las librerías y clases que necesitemos, lo importante es que si usamos **librerías externas**, estas **deben estar incluidas en el propio proyecto**, ya que **no se pueden importar librerías entre proyectos directamente**. Como las librerías de JAVA ya vienen por defecto, las tendremos que importar en la clase que las vayamos a usar pero sus paquetes ya están en el proyecto desde que lo creamos.



Los **import** se deben escribir debajo de las especificaciones **package**, que indican en qué lugar de la estructura del proyecto se encuentra la clase donde estamos trabajando.

EJEMPLO

✓  miLibreria
 >  MetodosUtiles.java
 >  OtrosMetodos.java
✓  ejerciciosU4
 >  Ejercicio1.java

Ejercicio1.java

```
1 package ejerciciosU4;  
2  
3 import miLibreria.MetodosUtiles;  
4  
5 public class Ejercicio1{  
6     ...  
7 }
```

Esta clase se encuentra en el paquete `ejerciciosU4`, e importa la clase `MetodosUtiles.java` de la librería que hemos creado antes, además de la clase `Scanner`. Lo normal es importar sólo las clases con las que vamos a trabajar.

Si queremos importar **todas las clases** de la librería, en vez de especificar cuál queremos debemos poner `*`.

U3 - BATERÍA DE EJERCICIOS

Los siguientes ejercicios se van a enfocar de forma que inicialmente se crearán las clases sencillas con los atributos y getters y setters, y posteriormente se irán modificando para ir añadiendo mejoras y funcionalidades. Desde el inicio vamos a aplicar el principio de encapsulamiento.

En cada ejercicio es necesario crear dos archivos, uno para la clase que contiene el main (clase principal) y otro para la clase a desarrollar, que se utilizará desde el main.

>> PARTE 1

Creación de clases con atributos privados y getters y setters. Creación de los objetos y utilización de los métodos creados.

Punto_1	<p>Crea un programa con una clase llamada Punto que representará un punto de dos dimensiones en un plano. Solo contendrá dos atributos enteros llamadas x e y (coordenadas).</p> <p>En el main de la clase principal instancia 3 objetos Punto con las coordenadas (5,0), (10,10) y (-3, 7). Muestra por pantalla sus coordenadas (utiliza un println para cada punto). Modifica todas las coordenadas (prueba distintos operadores, modificando el valor directamente, operando sobre el valor que tiene el punto, como puede ser sumar, restar, multiplicar o dividir, etc) y vuelve a imprimirlas por pantalla.</p>
Persona_1	<p>Crea un programa con una clase llamada Persona que representará los datos principales de una persona: dni, nombre, apellidos y edad.</p> <p>En el main de la clase principal instancia dos objetos de la clase Persona. Luego, pide por teclado los datos de ambas personas (guárdalos en los objetos). Por último, imprime dos mensajes por pantalla (uno por objeto) con un mensaje del estilo "Azucena Luján García con DNI ... es / no es mayor de edad". (se comprobará si es mayor de edad o no en el main)</p>
Rectangulo_1	<p>Crea un programa con una clase llamada Rectangulo que representará un rectángulo mediante dos coordenadas (x1,y1) y (x2,y2) en un plano, por lo que la clase deberá tener cuatro atributos enteros: x1, y1, x2, y2.</p> <p>En el main de la clase principal instancia 2 objetos Rectangulo en (0,0)(5,5) y (7,9)(2,3). Muestra por pantalla sus coordenadas, perímetros (suma de lados) y áreas (ancho x alto). Modifica todas las coordenadas como consideres y vuelve a imprimir coordenadas, perímetros y áreas.</p>
Articulo_1	<p>Crea un programa con una clase llamada Articulo con los siguientes atributos: nombre, precio (sin IVA), iva (siempre será 21 y por tanto no podrá ser modificado desde fuera de la clase) y cuantosQuedan (representa cuantos quedan en el almacén).</p> <p>En el main de la clase principal instancia un objeto de la clase artículo. Asígnale valores a todos sus atributos (los que quieras) y muestra por pantalla un mensaje del estilo "Pijama - Precio:10€ - IVA:21% - PVP:12,1€" (el PVP es el precio de venta al público, es decir, el precio con IVA). Luego, cambia el precio y vuelve a imprimir el mensaje.</p>

>> PARTE 2

En esta parte se crearán constructores, que son los métodos que se ejecutan cuándo se crea el objeto. Si no tenemos constructor, se llama al constructor por defecto, que da valores por defecto a todos los atributos de la clase. Es mejor crear nuestros propios constructores para controlar que se hace cuando se crea el objeto.

En esta parte vamos a modificar los programas del apartado anterior, se puede hacer una copia o modificar los creados anteriormente.

Punto_2	<p>Añade a la clase Punto un constructor que de valor inicial a las coordenadas x e y.</p> <p>Corrige el código del main para hacer uso del constructor.</p> <p>Al crear el constructor con los parámetros, ya no podemos utilizar el constructor por defecto (<code>new Punto()</code>) esto nos da la ventaja, de que siempre que creemos un objeto este tendrá valores, y no se nos olvidará asignarlos posteriormente.</p>
Persona_2	<p>Añade a la clase Persona el constructor con todos los parámetros.</p> <p>Realiza las modificaciones necesarias en el main.</p>
Rectangulo_2	<p>En nuestro software necesitamos asegurarnos de que la coordenada (x1,y1) represente la esquina inferior izquierda y la (x2,y2) la superior derecha del rectángulo. Añade a Rectangulo un constructor con los 4 parámetros. Incluye un if que compruebe los valores (*). Si son válidos guardará los parámetros en el objeto. Si no lo son mostrará un mensaje del estilo "ERROR al instanciar Rectangulo..." utilizando <code>System.err.println(...)</code>. No podremos evitar que se instancie el objeto pero al menos avisaremos por pantalla.</p> <p>Corrige el main para utilizar dicho constructor. Debería mostrar un mensaje de error.</p> <p>(*) Pista: Es suficiente con un <code>if ((condición) && (condición))</code></p>
Articulo_2	<p>Añade un constructor con 3 parámetros que asigne valores a nombre, precio y cuantosQuedan. Dicho constructor deberá mostrar un mensaje de error si alguno de los valores nombre, precio o cuantosQuedan no son válidos. ¿Qué condiciones crees que podrían determinar si son válidos o no? Razónalo e implementa el código.</p> <p>Corrige el main y prueba a crear varios artículos. Introduce algunos con valores incorrectos para comprobar si avisa del error.</p>

>> PARTE 3

Una clase bien diseñada debería incluir métodos que realicen operaciones con la información de los objetos. De ese modo la clase dispondrá de funcionalidades útiles tanto para nosotros como para otros programadores. Esto es una práctica habitual y muy recomendable.

Por ejemplo, la clase Scanner tiene métodos como `getInt()`, `getDouble()`, `getLine()`, etc. que alguien ha programado y podremos utilizar cuando los necesitemos sin tener que programarlo nosotros ni preocuparnos por cómo funcionan internamente. Lo mismo sucede con la clase Math (`random`, `min`, `max`, `abs`, etc.).

Todas estas son clases que Java incorpora por defecto (hay miles). Existen muchas otras que permiten hacer todo tipo de cosas como crear interfaces gráficas con botones, trabajar con imágenes, leer y escribir en archivos, reproducir música, enviar o recibir datos a través de la red, etc.

En esta unidad estamos aprendiendo a diseñar y programar nuestras propias clases, los fundamentos de la Programación Orientada a Objetos, necesario en cualquier proyecto software de cierta envergadura.

Recuerda que los métodos pueden ser `private` (solo pueden utilizarse desde dentro de la clase) o `public` (pueden utilizarse desde fuera, forman parte de la interfaz de la clase).

Todos los métodos deben estar correctamente comentados utilizando JavaDoc (`/** */`).

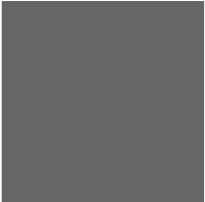
En esta parte vamos a modificar los programas del apartado anterior, se puede hacer una copia o modificar los creados anteriormente.

Punto_3	<p>Añade a la clase Punto los siguientes métodos públicos:</p> <ul style="list-style-type: none">• <code>public void imprime()</code> // Imprime por pantalla las coordenadas. Ejemplo: "(7, -5)"• <code>public void setXY(int x, int y)</code> // Modifica ambas coordenadas. Es como un setter doble.• <code>public void desplaza(int dx, int dy)</code> // Desplaza el punto la cantidad (dx,dy) indicada. Ejemplo: Si el punto (1,1) se desplaza (2,5) entonces estará en (3,6).• <code>public int distancia(Punto p)</code> // Calcula y devuelve la distancia entre el propio objeto (this) y otro objeto (Punto p) que se pasa como parámetro: distancia entre dos coordenadas. <p>Prueba a utilizar estos métodos desde el main para comprobar su funcionamiento.</p>
Persona_3	<p>Añade a la clase Persona los siguientes métodos públicos:</p> <ul style="list-style-type: none">• <code>public void imprime()</code> // Imprime la información del objeto: "DNI:... Nombre:... etc."• <code>public boolean esMayorEdad()</code> // Devuelve true si es mayor de edad (false si no).• <code>public boolean esJubilado()</code> // Devuelve true si tiene 65 años o más (false si no).• <code>public int diferenciaEdad(Persona p)</code> // Devuelve la diferencia de edad entre 'this' y p. <p>Prueba a utilizar estos métodos desde el main para comprobar su funcionamiento.</p>
Articulo_3	<p>Añade a la clase Artículo métodos públicos con las siguientes funcionalidades:</p> <ul style="list-style-type: none">• Método para imprimir la información del artículo por pantalla.• Método <code>getPVP</code> que devuelva el precio de venta al público (PVP) con iva incluido.• Método <code>getPVPDescuento</code> que devuelva el PVP con un descuento pasado como argumento.• Método <code>vender</code> que actualiza los atributos del objeto tras vender una cantidad 'x' (si es posible). Devolverá true si ha sido posible (false en caso contrario).• Método <code>almacenar</code> que actualiza los atributos del objeto tras almacenar una cantidad 'x' (si es posible). Devolverá true si ha sido posible (false en caso contrario).

>> EJERCICIOS COMPLEMENTARIOS

Emisora	<p>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 para manejar oscila entre los 80 MHz y los 108 MHz y que, al inicio, el controlador sintonice la frecuencia indicada en el constructor o 80 MHz por defecto. Si durante una operación de subida o bajada se sobrepasa uno de los dos límites, la frecuencia sintonizada debe</p>
----------------	---

	<p>pasar a ser la del extremo contrario. Escribir un pequeño programa principal para probar su funcionamiento.</p>
Bombilla	<p>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 este se apaga, todas las bombillas quedan apagadas. Cuando el interruptor general se activa, las bombillas vuelven a estar encendidas o apagadas, según estuvieran antes. Cada bombilla se enciende y se apaga individualmente, pero solo responde que está encendida si su interruptor particular está activado y además hay luz general.</p>
Compañía de móviles	<p>Necesitamos crear un programa para gestionar los cobros de las llamadas de móvil de sus usuarios. La compañía guarda la siguiente información de cada teléfono móvil para lo que se creará una clase, los atributos a almacenar serán:</p> <ul style="list-style-type: none"> • Número • Tarifa • Además necesita almacenar el consumo que tiene cada móvil (este dependerá de la tarifa). <p>Hay tres posibles tarifas:</p> <ul style="list-style-type: none"> • Elefante, el coste es a 0,30 € el minuto • Tigre, el coste es a 0,18 € el minuto • Gato, el coste es a 0,07 € el minuto <p>Las funcionalidades que tendrá la clase Movil son las siguientes:</p> <ul style="list-style-type: none"> • Llamar, de forma que cuando se realiza la llamada se ajuste el consumo total del móvil. Cada llamada tendrá una duración en segundos. • Reiniciar la factura, que consistirá es poner a cero el consumo total, suponemos que se cambia de mes y la tarifa se reinicia. • Resumen de la factura, donde se indique el número, tarifa, y consumo total. <p>El programa principal, deberá crear varios móviles y realizar llamadas de los mismos, mostrar la factura, y reiniciar tarifas. Se podrá hacer secuencialmente o mediante interacción con el usuario o menús.</p>
Venta de entradas	<p>En este caso vamos a gestionar el stock y venta de entradas (no numeradas) de un estadio de fútbol. El estadio tiene 4 zonas con las siguientes características:</p> <ol style="list-style-type: none"> 1. Fondo sur: 500 entradas a 10€ cada una 2. Fondo norte: 700 entradas a 10€ cada una 3. Preferencia alta: 300 entradas 25€ cada una 4. Preferencia baja: 250 entradas 40€ cada una <p>Hay que controlar que existen entradas antes de venderlas.</p> <p>Será necesario crear una clase Zona con las siguientes características:</p> <ul style="list-style-type: none"> • Atributos: nombre de la zona, número de entradas y precio de cada entrada. • Métodos: get y set de los atributos y double venderEntradas(int numEntradas) <ul style="list-style-type: none"> ◦ Se deberá comprobar que el número de entradas que se desea comprar es menor que las entradas disponibles si no es así se mostrará un mensaje, y no se realizará la venta, y en caso de realizarse la venta, se devolverá el precio total. <p>El programa principal deberá crear las 4 zonas, asignando el número de entradas que tiene cada zona y el precio de la entrada por zona.</p> <p>Se deberá mostrar el siguiente menú, y el programa no finalizará hasta que se seleccione la opción de salir del menú:</p>

- 
1. Mostrar número de entradas libres hay en cada zona
 2. Mostrar precio por entrada en cada zona
 3. Vender entradas
 4. Salir

En caso de seleccionar la opción 3, será necesario indicar la zona de la que se quieren comprar las entradas y solicitar cuántas entradas quiere.