

TEMA 4: Utilización de objetos y Creación de clases

Sumario

.....	1
Programación orientada a objetos.....	2
Características de la POO.....	3
Propiedades y métodos de los objetos.....	5
Métodos.....	5
Atributos.....	5
Clases.....	6
Declaración e Instanciación de un objeto.....	8
Destrucción de objetos y liberación de memoria.....	9
Referencia a un objeto.....	9
Paso de parámetros a métodos.....	11
Paquetes.....	13
Import.....	13
Las clases en profundidad.....	14
Constructores.....	14
Tipos de acceso.....	17
this (palabra reservada).....	20
Constructor de Copia.....	23
static (palabra reservada).....	26
Sobrecarga de métodos.....	27
La clase Object.....	29

Programación orientada a objetos

Lo que hace la programación orientada a objetos es cambiar la forma de trabajar. No supone una revolución con los métodos anteriores, sino un cambio en la forma en la que se hacen las mismas cosas de siempre. La mayoría del trabajo se delega a los propios datos, de modo que los datos ya no son estáticos, sino que se encargan de mantenerse en forma a sí mismos. Citando al creador del lenguaje Ruby: “dejamos de tratar cada pieza de dato como una caja en la que se puede abrir su tapa y arrojar cosas en ella y empezamos a tratarlos como máquinas funcionales cerradas con unos pocos interruptores y diales bien definidos”.

En la POO (programación orientada a objetos) a menudo se trata de representar las entidades y objetos que nos encontramos en el mundo real mediante componentes de una aplicación. Es decir, debemos establecer una correspondencia directa entre el espacio del problema y el espacio de la solución. ¿Pero en la práctica esto qué quiere decir? Pues que a la hora de escribir un programa, nos fijaremos en los objetos involucrados, sus características comunes y las acciones que pueden realizar. Una vez localizados los objetos que intervienen en el problema real (espacio del problema), los tendremos que trasladar al programa informático (espacio de la solución). Con este planteamiento, la solución a un problema dado se convierte en una tarea sencilla y bien organizada.

Pongamos el ejemplo de un microondas. Consta, entre otros muchos subsistemas, de un temporizador. Su trabajo consiste en mantener el horno encendido durante un tiempo determinado por el usuario. ¿Cómo podríamos representar esto en un lenguaje de programación? En Java, por ejemplo, el temporizador podría ser, simplemente, una variable numérica de tipo float o double. El programa manipularía esa variable disminuyendo el valor una vez por segundo, y permitiría al usuario establecer el tiempo inicial del temporizador antes de poner el horno en marcha.

Con este enfoque tradicionalista, un error en cualquier parte del programa podría terminar asignando un valor falso a la variable, como un número negativo o un tiempo de varias horas. Hay un número infinito de razones inesperadas por las que podría llegar a suceder esto. Cualquiera con cierta experiencia programando sabe que se pueden perder horas o días tratando de encontrar ese error.

Pero si programamos con un lenguaje orientado a objetos, no pensaremos en el tipo de datos que mejor se ajusta a un temporizador de microondas, sino en el modo en el que un temporizador de microondas funciona en el mundo real. No parece una gran diferencia, pero lo es.

Características de la POO

- **Abstracción.** Es el proceso por el cual definimos las características más importantes de un objeto, sin preocuparnos de cómo se escribirán en el código del programa, simplemente lo definimos de forma general. En la Programación Orientada a Objetos la herramienta más importante para soportar la abstracción es la clase. Básicamente, una clase es un tipo de dato que agrupa las características comunes de un conjunto de objetos. Poder ver los objetos del mundo real que deseamos trasladar a nuestros programas, en términos abstractos, resulta de gran utilidad para un buen diseño del software, ya que nos ayuda a comprender mejor el problema y a tener una visión global de todo el conjunto. Por ejemplo, si pensamos en una clase Vehículo que agrupa las características comunes de todos ellos, a partir de dicha clase podríamos crear objetos como Coche y Camión. Entonces se dice que Vehículo es una abstracción de Coche y de Camión.
- **Modularidad.** Una vez que hemos representado el escenario del problema en nuestra aplicación, tenemos como resultado un conjunto de objetos software a utilizar. Este conjunto de objetos se crean a partir de una o varias clases. Cada clase se encuentra en un archivo diferente, por lo que la modularidad nos permite modificar las características de la clase que define un objeto, sin que esto afecte al resto de clases de la aplicación.
- **Encapsulación.** También llamada "ocultamiento de la información". La encapsulación o encapsulamiento es el mecanismo básico para ocultar la información de las partes internas de un objeto a los demás objetos de la aplicación. Con la encapsulación un objeto puede ocultar la información que contiene al mundo exterior, o bien restringir el acceso a la misma para evitar ser manipulado de forma inadecuada. Por ejemplo, pensemos en un programa con dos objetos, un objeto Persona y otro Coche. Persona se comunica con el objeto Coche para llegar a su destino, utilizando para ello las acciones que Coche tenga definidas como por ejemplo conducir. Es decir, Persona utiliza Coche pero no sabe cómo funciona internamente, sólo sabe utilizar sus métodos o acciones.
- **Jerarquía.** Mediante esta propiedad podemos definir relaciones de jerarquías entre clases y objetos. Las dos jerarquías más importantes son la jerarquía "es un" llamada generalización o especialización y la jerarquía "es parte de", llamada agregación. Conviene detallar algunos aspectos:
 - **La generalización o especialización,** también conocida como herencia, permite crear una clase nueva en términos de una clase ya existente (herencia simple) o de varias clases ya existentes (herencia múltiple). Por ejemplo, podemos crear la clase CochedeCarreras a partir de la clase Coche, y así sólo tendremos que definir las nuevas características que tenga.
 - **La agregación, también conocida como inclusión,** permite agrupar objetos relacionados entre sí dentro de una clase. Así, un Coche está formado por Motor, Ruedas, Frenos y Ventanas. Se dice que Coche es una agregación y Motor, Ruedas, Frenos y Ventanas son agregados de Coche.
- **Polimorfismo.** Esta propiedad indica la capacidad de que varias clases creadas a partir de una antecesora realicen una misma acción de forma diferente. Por ejemplo, pensemos en la clase Animal y la acción de expresarse. Nos encontramos que cada tipo de Animal puede hacerlo de manera distinta, los Perros ladran, los Gatos maullan, las Personas hablamos, etc. Dicho de otra manera, el polimorfismo indica la posibilidad de tomar un objeto (de tipo

Animal, por ejemplo), e indicarle que realice la acción de expresarse, esta acción será diferente según el tipo de mamífero del que se trate.

Propiedades y métodos de los objetos.

Las partes de un objeto son:

- **Campos, Atributos o Propiedades:** Parte del objeto que almacena los datos. También se les denomina Variables Miembro. Estos datos pueden ser de cualquier tipo primitivo (boolean, char, int, double, etc) o ser su vez ser otro objeto. Por ejemplo, un objeto de la clase Coche puede tener un objeto de la clase Ruedas.

- **Métodos o Funciones Miembro:** Parte del objeto que lleva a cabo las operaciones sobre los atributos definidos para ese objeto.

La única forma de manipular la información del objeto es a través de sus métodos. Es decir, si queremos saber el valor de algún atributo, tenemos que utilizar el método que nos muestre el valor de ese atributo. De esta forma, evitamos que métodos externos puedan alterar los datos del objeto de manera inadecuada. Se dice que los datos y los métodos están encapsulados dentro del objeto.

Métodos

Sea el ejemplo:

```
String texto = "Hola Mundo";  
int longitudTexto = texto.length();
```

Intuitivamente es fácil de comprender: a este objeto cadena se le está pidiendo que diga la longitud que tiene (y, como es una cadena de caracteres, nos responde con la cantidad total de letras de que consta). Técnicamente, lo que hemos hecho se llama invocar el método **length()** del objeto: texto

Atributos

Los atributos son los datos incluidos en un objeto. Son como las variables en los lenguajes de programación clásicos, pero están encapsuladas dentro de un objeto y, salvo que se indique lo contrario, son invisibles desde el exterior.

Los atributos de un objeto definen las características del mismo. Por ejemplo, un atributo del temporizador del microondas debería ser el número de segundos que éste debe permanecer activo

Clases

Una clase es un patrón para construir objetos. Por tanto, un objeto es una variable perteneciente a una clase determinada. Es importante distinguir entre objetos y clases: la clase es simplemente una declaración, no tiene asociado ningún objeto. Y todo objeto debe pertenecer a una clase.

Veamos un ejemplo:

```
class Persona {  
  
    private String nombre;  
    private String apellido;  
    private int edad;  
  
    public String getNombreCompleto() {  
        return nombre + " " + apellido;  
    }  
    public int getEdad() {  
        return edad;  
    }  
    public void setNombre(String txt) {  
        nombre = txt;  
    }  
    public void setApellido(String txt) {  
        apellido = txt;  
    }  
    public void setEdad(int n){  
        edad = n;  
    }  
}
```

En la anterior clase los atributos son: nombre, apellidos y edad. Son private, eso significa que no se puede acceder a ellos salvo que lo hagamos mediante un método. Así, para poder obtener la edad de un objeto persona que llamaremos: ciudadano lo haremos escribiendo:

ciudadano.getEdad()

Veamos un ejemplo completo para el objeto ciudadano:

```
{  
  
    //creamos el objeto ciudadano  
    Persona ciudadano = new Persona();  
  
    //le establecemos sus atributos  
    ciudadano.setNombre("Manuel");  
    ciudadano.setApellido("Estévez");  
    ciudadano.setEdad(20);  
  
    //Mostramos el nombre completo en pantalla  
    System.out.println(ciudadano.getNombreCompleto());  
}
```

Para crear el objeto utilizamos la instrucción new:

new Persona()

ejecutando el método getNombreCompleto() hemos obtenido el nombre y el apellido:

ciudadano.getNombreCompleto()

y ejecutando los métodos set hemos puesto valores a los atributos:

ciudadano.setNombre("Manuel");

Vemos que le hemos pasado el literal de tipo String: “Manuel” al método setNombre()
Entendamos mejor ese paso de información

Declaración e Instanciación de un objeto

En la declaración definimos el tipo del objeto:

Persona ciudadano;

En la línea anterior hemos dicho que ciudadano es de tipo Personas. De la misma forma que:

String texto;

estaríamos diciendo que texto es de tipo String

Con la sentencia anterior en la que definíamos la variable ciudadano como de tipo persona, aún no hemos creado ningún objeto. Para crearlo hay que hacer uso de la instrucción new:

ciudadano = new Persona();

Es importante entender que la variable ciudadano es únicamente una referencia, un apuntador al objeto Persona que se ha creado en memoria

Destrucción de objetos y liberación de memoria

Cuando un objeto deja de ser utilizado, es necesario liberar el espacio de memoria y otros recursos que posea para poder ser reutilizados por el programa. A esta acción se le denomina **destrucción del objeto**.


En Java la destrucción de objetos corre a cargo del **recolector de basura (garbage collector)**. Es un sistema de destrucción automática de objetos que ya no son utilizados. Lo que se hace es liberar una zona de memoria que había sido reservada previamente mediante el operador `new`.

Referencia a un objeto

Estamos acostumbrados a que cuando igualamos variables copiamos su valor:

```
{
    int a, b;
    a=7;
    b=a; //en estos momentos b vale 7 porque "copiamos" de a hacia b
    System.out.println("b:"+ b + " a:"+a);

    //modificar b no tiene efecto en a
    b=3;
    System.out.println("b:"+ b + " a:"+a);
}
```

 **Práctica 1:** Copia el código anterior y toma una captura de pantalla que muestre la salida del programa en ejecución. ¿ qué valores muestra para a y b ?

Cuando antes hicimos: **$b=a$** estábamos copiando lo que estaba en memoria RAM para la variable **a** en la parte de memoria RAM que corresponde para la variable **b**

Es por eso por lo que cuando le asignamos el valor 3 a la variable **b** no hubo efecto en la variable **a**

Pero con los objetos las cosas funcionan distinto. NO SE COPIA EL OBJETO se copia una referencia al objeto.

Veamos un ejemplo:

```
{
    //declaramos c y d como de tipo Persona
    Persona c, d;

    //creamos un objeto y guardamos su referencia en c
    c = new Persona();

    //Le establecemos sus atributos
    c.setNombre("Ana");
    c.setApellido("Hernández");
    c.setEdad(19);

    //tomamos la referencia al objeto que guarda c y se la damos a d
    d=c;

    //Modificamos la edad del objeto apuntado por la variable d:
    d.setEdad(25);

    //vemos la edad almacenada accediendo con d:
    System.out.println( "Edad accediendo con d: " + d.getEdad());

    //vemos la edad almacenada accediendo con c:
    System.out.println( "Edad accediendo con c: " + c.getEdad());
}
```

● **Práctica 2:** Copia el código anterior y toma una captura de pantalla que muestre la salida del programa en ejecución. ¿ qué valores muestra para c y d ? Modifica ahora el nombre Ana por otro mediante: `c.setNombre()` y muestra en pantalla el nombre apuntado por c y el apuntado por d

Así pues podemos observar que las variables `c` y `d` están apuntando al mismo objeto.

Esto es porque lo que se guarda en las variables es la referencia a la ubicación en memoria del objeto. Cuando hacemos: `d=c` estamos copiando esa referencia de una variable a otra. Se COPIA LA REFERENCIA NO EL OBJETO

Una forma adicional de observar esto es imprimiendo en pantalla las variables. Se podrá observar que nos dirán algo al estilo de: `Persona@6d06d69c` que nos viene a decir que la variable está referenciando a un objeto de tipo persona que está alojado en la posición de memoria: `6d06d69c`

Imprimiendo en pantalla las variables veremos que nos muestra la misma posición de memoria

● **Práctica 3:** Imprime en pantalla las variables `c` y `d` toma captura de pantalla de la salida. ¿ en qué posición de memoria está guardado el objeto? Crea un objeto nuevo mediante `new` y guarda la referencia en una variable llamada `e` Imprime `e` ¿ocupa la misma posición ?

Paso de parámetros a métodos

Los métodos pueden recibir una serie de valores denominados parámetros. En la clase Persona del ejemplo anterior, el método `setEdad()`, por ejemplo, recibía un parámetro de tipo `int` llamado `n`. En ese parámetro se indica al método cuál es la edad que debe almacenarse en el estado del objeto.

Por lo tanto, los parámetros son imprescindibles para que el objeto reciba los mensajes correctamente. Si no, ¿cómo le indicaríamos al objeto de tipo Persona cuál es la edad que tiene que almacenar?

Un método puede tener una lista larguísima de parámetros, o ninguno. Lo más habitual es que tenga entre cero y unos pocos.

En la declaración del método hay que indicar el tipo de cada parámetro. Observa este ejemplo:

```
public void setDatos(String nombre, String apellido, int edad)
```

Este hipotético método `setDatos()` podría servir para asignar valor a todos los atributos de la clase Persona del ejemplo anterior. Por supuesto, cuando llamemos al método `setDatos()` para que se ejecute, será necesario pasarle cuatro datos que coincidan en tipo con los cuatro parámetros. Esta podría ser una posible invocación:

```
p = new Persona();  
p.setDatos("Ana", "Sánchez", 45);
```

En el caso del paso de parámetros ocurre como en la igualación de variables. Si son de tipo primitivo (`int`, `double`,...) se copia el valor a la variable del método. Si es un objeto se copia la referencia.

Valores devueltos

En el ejemplo de la clase Persona se pueden observar métodos que terminan con la sentencia `return` y otros `no`. Los que sí lo hacen devuelven un resultado al código que los llamó. Ese resultado puede ser de cualquier tipo y también hay que indicarlo en la declaración del método.

El método `setDatos()` anterior no devuelve nada, por lo que, en la declaración, se usa la palabra `void` (vacío). Pero el método `getEdad()`, por ejemplo, devuelve un valor entero (la edad de la persona), y por eso se indica `int` en la declaración:

```
public int getEdad() {  
    return edad;  
}
```

Métodos estáticos

Cuando trabajábamos con cadenas de caracteres utilizando la clase `String`, veíamos las operaciones que podíamos hacer con ellas: obtener longitud, comparar dos cadenas de caracteres, cambiar a mayúsculas o minúsculas, etc. Pues bien, sin saberlo estábamos utilizando métodos estáticos definidos por Java para la clase `String`. Pero ¿qué son los métodos estáticos? Veámoslo.

Los métodos estáticos son aquellos métodos definidos para una clase que se pueden usar directamente, sin necesidad de crear un objeto de dicha clase. También se llaman métodos de clase.

Para llamar a un método estático utilizaremos:

- El nombre del método, si lo llamamos desde la misma clase en la que se encuentra definido.
- El nombre de la clase, seguido por el operador punto (`.`) más el nombre del método estático, si lo llamamos desde una clase distinta a la que se encuentra definido:

NombreClase.nombreMetodoEstatico()

Un ejemplo de lo anterior que ya hemos usado es:

Math.random()

En la Biblioteca de Clases de Java existen muchas clases que contienen métodos estáticos. Pensemos en las clases que ya hemos utilizado en unidades anteriores, como hemos comentado la clase `String` con todas las operaciones que podíamos hacer con ella y con los objetos instanciados a partir de ella. O bien la clase `Math`, para la conversión de tipos de datos. Todos ellos son métodos estáticos que la API de Java define para esas clases. Lo importante es que tengamos en cuenta que al tratarse de métodos estáticos, para utilizarlos no necesitamos crear un objeto de dichas clases.

Paquetes

Conforme nuestros programas se van haciendo más grandes, el número de clases va creciendo. Meter todas las clases en único directorio no ayuda a que estén bien organizadas, lo mejor es hacer grupos de clases, de forma que todas las clases que estén relacionadas o traten sobre un mismo tema estén en el mismo grupo.

Un paquete de clases es una agrupación de clases que consideramos que están relacionadas entre sí o tratan de un tema común.

Las clases de un mismo paquete tienen un acceso privilegiado a los atributos y métodos de otras clases de dicho paquete. Es por ello por lo que se considera que los paquetes son también, en cierto modo, unidades de encapsulación y ocultación de información.

Los paquetes se declaran utilizando la palabra clave `package` seguida del nombre del paquete. Para establecer el paquete al que pertenece una clase hay que poner una sentencia de declaración como la siguiente al principio de la clase:

```
package nombrepaquete;
```

Import

Cuando queremos utilizar una clase que está en un paquete distinto a la clase que estamos utilizando, se suele utilizar la sentencia `import`. Por ejemplo, si queremos utilizar la clase `Scanner` que está en el paquete `java.util` de la Biblioteca de Clases de Java, tendremos que utilizar esta sentencia:

```
import java.util.Scanner;
```

Se pueden importar todas las clases de un paquete, así:

```
import java.io.*;
```

Esta sentencia debe aparecer al principio de la clase, justo después de la sentencia package, si ésta existiese.

Las clases en profundidad

En el ciclo de vida de un objeto se pueden distinguir las fases de:

- Construcción del objeto.
- Manipulación y utilización del objeto accediendo a sus miembros.
- Destrucción del objeto.

Durante la fase de construcción o instanciación de un objeto es cuando se reserva espacio en memoria para sus atributos y se inicializan algunos de ellos. Un constructor es un método especial con el mismo nombre de la clase y que se encarga de realizar este proceso.

Como ya hemos visto la creación de un objeto mediante el operador **new**. Sin embargo las clases que hasta ahora has creado no tenían constructor. En esos casos se utilizan los constructores por defecto que proporciona Java al compilar la clase. Veamos el proceso de creación de un objeto mediante un constructor definido por el programador

Constructores

Los métodos constructores se encargan de llevar a cabo el proceso de creación o construcción de un objeto.

El constructor es invocado automáticamente al crear un objeto de la clase.

La estructura de los constructores es similar a la de cualquier método, con las excepciones de que **no tiene tipo de dato devuelto** (no devuelve ningún valor) y que **el nombre del método constructor debe ser obligatoriamente el nombre de la clase**

```
class NombreClase{  
    NombreClase( ... ){  
        ...  
    }  
}
```

Veamos un ejemplo con la clase Persona que teníamos antes:

```
class Persona {  
  
    private String nombre;  
    private String apellido;  
    private int edad;  
  
    Persona(String n, String a, int e ){  
        nombre=n;  
        apellido=a;  
        edad=e;  
    }  
  
    public String getNombreCompleto() {  
        return nombre + " " + apellido;  
    }  
    public int getEdad() {  
        return edad;  
    }  
    public void setNombre(String txt) {  
        nombre = txt;  
    }  
    public void setApellido(String txt) {  
        apellido = txt;  
    }  
    public void setEdad(int n){  
        edad = n;  
    }  
}
```

Se ha establecido un constructor para la clase Persona que recibe 3 parámetros que serán los valores de nombre, apellido y edad

La forma de llamar al constructor sería ahora:

```
{  
    //declaración del tipo de la variable;  
    Persona ciudadano;  
    //creamos el objeto y lo referenciamos con ciudadano  
    ciudadano = new Persona("Manuel", "Estévez", 20);  
}
```

Observar que en el momento en el que creamos el objeto mediante la orden new, estamos usando el constructor al que le estamos pasando los datos.

- **Práctica 4:** Crear la clase: `Conversor` Esta clase sirve para cambiar de euros a dolares y de dolares a euros. Tiene un constructor que recibe el tipo de cambio (utilizar el cambio actual) Pongamos por ejemplo: 0.8615 y tiene dos métodos: `euroToDolar()` y `dolarToEuro()` que reciben un `double` que representa respetivamente euros y dolares y devuelve la divisa transformada. También tiene un método llamado: `establecerTipo(double t)` que nos permitirá modificar posteriormente el tipo de cambio por si cambia con el tiempo

En la práctica anterior hemos visto que hemos establecido el tipo de cambio al principio, en el constructor. Sin embargo pudiera ocurrir que quisiéramos crear el objeto sin establecer el tipo y que tome en ese caso un valor por defecto para el tipo de cambio. Para eso podríamos disponer de un constructor adicionales

Podemos disponer de varios constructores para una misma clase que deben recibir un número diferente de parámetros y/o tipo de parámetros

- **Práctica 5:** Crear la clase: `Perro` Esta clase tiene por atributos `String raza`, `String nombre` y `int edad`. Tiene un constructor que recibe la raza el nombre y la edad. Ej.

```
Perro(String n, String r, int e){  
    nombre=n;  
    raza=r;  
    edad=e;  
}
```

y creamos otro constructor para cuando desconocemos la edad del animal:

```
Perro(String n, String r){  
    nombre=n;  
    raza=r;  
    edad=-1;  
}
```

crear un tercer Constructor que sirva para el caso en el que no sabemos ni la raza ni la edad

- **Práctica 6:** Crear un constructor para la clase `Conversor` que no reciba parámetros y especifique un tipo de cambio por defecto de 0.85 Crear dos objetos e instanciarlos

Cuando disponemos de dos o más constructores (y en general para cualquier método) con el mismo nombre, pero distintos parámetros decimos que el constructor (o método) está **sobrecargado**. La sobrecarga de constructores o métodos permite llevar a cabo una tarea de distintas maneras.

● **Práctica 7:** Define una clase Profesor con atributos: nombre (String), apellidos (String), edad (int), soltero (boolean), especialista (boolean). Define un constructor que reciba los parámetros necesarios para la inicialización y otro constructor que no reciba parámetros. Crea los métodos getter y setter para poder establecer y obtener los valores de los atributos.

Nota: Netbeans tiene una forma cómoda de crear getter() y setter():

Refactor → Encapsule fields

(o con atajo de teclado: CTRL+ALT+SHIFT+E)

● **Práctica 8:** Crear la clase Coche que tenga por atributos: String nombre, boolean encendido, boolean frenoDeManoPuesto, int posicion. Con un constructor que reciba el nombre: Coche(String n) y otro constructor que reciba el nombre y la posicion: Coche(String n, int p) Establecer los getter y los setter

Tipos de acceso

Hemos visto en el ejemplo de la clase Persona que se ha puesto la palabra public a todos los métodos. Si bien habraremos con más detalle de los modificadores de acceso cuando veamos herencia por de pronto debemos tener en cuenta como afectan **private**, **protected**, **public** y por omisión en el tipo de acceso para la propia Clase, para el Paquete y para todo otro elemento:

Modificador de Accesibilidad	Accesible para			
	Clase	Subclase	Paquete	Universal
private	Sí	No	No	No
Por omisión	Sí	No	Sí	No
protected	Sí	Sí	Sí	No
public	Sí	Sí	Sí	Sí

Lo anterior significa que cuando en el ejemplo pusimos: public estábamos permitiendo el acceso de todo el mundo a esos métodos. Sin embargo cuando pusimos private impedíamos que pudiera accederse desde fuera de la propia clase a los atributos nombre, apellido, edad

● **Práctica 9:** Tratar de acceder desde fuera de la clase Persona a los atributos privados.
Ej.

```
{  
    Persona ciudadano;  
    ciudadano = new Persona("Manuel", "Estévez", 20);  
    System.out.println( ciudadano.nombre );  
}
```

¿qué ocurre? Tomar captura de pantalla del mensaje del IDE. Modificar el acceso de los atributos a public y tratar de acceder ¿ qué ocurre ahora ?

En Java hay varios niveles de acceso a los miembros de una clase, es decir, a los atributos y métodos de una clase:

- * Público (**public**). Se puede acceder a ese miembro desde cualquier otra clase de la aplicación.
- * Privado (**private**). No se puede acceder a ese miembro desde ninguna otra clase.
- * Protegido (**protected**). No se puede acceder a ese miembro desde ninguna otra clase, excepto las que pertenezcan al mismo paquete y las subclases, que sí podrán.
- * No especificado. Si no especificas el nivel de acceso, solo podrán acceder al miembro de la clase las clases del mismo paquete, pero no las subclases.

Cada vez que se declara un miembro de clase (atributo o método), hay que indicar su nivel de acceso (private, public, protected). Como hemos visto, no especificar el nivel de acceso también es una forma de hacerlo.

El conjunto de miembros de una clase declarados como públicos y protegidos constituyen su interfaz, es decir, es la parte que muestran al resto de la aplicación. Cualquier modificación interna de la clase no debería afectar nunca a su interfaz, de manera que el funcionamiento del resto de la aplicación no se vea alterado porque tengamos que alterar una línea de código de una de las clases.

Nota: A partir de ahora siempre intentaremos que el acceso a nuestros atributos en las prácticas del tema sean de tipo private y accederemos a ellos mediante métodos

● **Práctica 10:** Crear una clase llamada Cuenta que refleje una cuenta bancaria. Tiene por atributos: String numero, String titular, double saldo Así como los métodos ingresar() y retirar() que servirán para añadir o quitar saldo

- **Práctica 11:** Crear una clase llamada Cliente que emulará los gastos de un cliente de un hotel. Como atributos tendrá como mínimo: int id, String nombre, String apellido, int habitacion, double debe, int noches. Como métodos como mínimo gastar(double) y pagar(double) que reflejan cuando el cliente consume y aumenta su deuda así como pagar parte de su cuenta. Tener en cuenta que como mínimo en el debe del cliente estará la cuantía de alquilar la habitación por el número de noches que esté. Cuando se establezca el número de la habitación también se deberá establecer el precio/noche de la habitación

Niveles de acceso a una clase

También se puede controlar el nivel de acceso a una clase, declarando como pública o como privada la totalidad de la misma.

En el primer caso (clase pública) cualquier otra clase puede usarla, por ejemplo, para instanciar objetos.

```
// Clase pública
public class MiClase {
    ...
}
```

En el segundo, solo podrán usarla otras clases de su propio paquete:

```
// Clase privada al paquete
class MiClase {
    ...
}
```

Observar que no hemos usado la palabra “private” delante de class para declarar la clase privada. Las clases declaradas como “private class” son aún más privadas. Tanto, que solo pueden acceder a ellas las clases a las que pertenecen. De momento trabajaremos con clases privadas del primer tipo.

this (palabra reservada)

La palabra **this** es una palabra reservada de Java. Esto quiere decir que no puedes usarla como identificador de variable, constante, método o clase.

this hace referencia al propio objeto que está ejecutando el código. Es decir, si tenemos una clase Coche e instanciamos cinco objetos de esa clase, tendremos en nuestro programa cinco coches, cada uno con sus atributos y sus métodos. Si estamos ubicados dentro del código de la clase si nos referimos a this nos estamos refiriendo al propio objeto donde estamos

El objeto this se puede emplear para acceder a los atributos y a los métodos del propio objeto, pero muchas veces se omite para ahorrar código. En cambio, algunas veces es necesario usarlo para evitar ambigüedades.

En el siguiente ejemplo verás como usamos this una vez sin obligación y otra vez para resolver una de esas ambigüedades.

```
class Camion {
    private String marca;
    private double potencia;
    private String matricula;

    public String getMatricula() {
        return this.matricula; // return matricula; valdria tambien
    }

    public int setMatricula(String matricula) {
        //como matricula es un parametro que recibe el método se puede confundir
        //con el atributo matricula. Aqui es necesario this para diferenciar
        this.matricula = matricula;
    }
}
```

● **Práctica 12:** Completar la clase Camion con varios constructores. Uno para cuando no se le pasan parámetros, otro para únicamente la marca, otro con todos,... Se deberá usar la palabra reservada this para cada vez que nombremos un atributo en los constructores o en los métodos.

Hay otro uso que es interesante de la palabra reservada this. Y es para llamar a un constructor desde otro constructor.

En la anterior actividad hemos creado varios constructores donde la diferencia de uno a otro es únicamente que no se define algún/algunos atributo/s Eso se presta a llamar a un único constructor pasando el valor null o similar al primer constructor.

Veamos ejemplo con una clase llamada Coche:

```
class Coche{
    private String marca;
    private String modelo;
    private double potencia;
    private int puertas;
    private String matricula;
    Coche(String marca, String modelo, double potencia, int puertas, String
matricula){
        this.marca = marca;
        this.modelo = modelo;
        this.potencia = potencia;
        this.puertas = puertas;
        this.matricula = matricula;

    }
}
```

Si quisiéramos crear un constructor que únicamente se le pase la potencia de 110 y el número de puertas de 3 sería equivalente a llamar al anterior constructor de esta forma:

```
new Coche(null, null, 110, 3, null);
```

Pues bien, mediante this podemos conseguir hacer algo similar a eso desde el constructor específico:

```
class Coche{
    private String marca;
    private String modelo;
    private double potencia;
    private int puertas;
    private String matricula;
    Coche(String marca, String modelo, double potencia, int puertas, String
matricula){
        this.marca = marca;
        this.modelo = modelo;
        this.potencia = potencia;
        this.puertas = puertas;
        this.matricula = matricula;

    }

    Coche(double potencia, int puertas){
        this(null, null, potencia, puertas, null);
    }
}
```

Hemos creado un constructor que recibe únicamente dos parámetros y que nos crea el nuevo objeto apoyándose en el anterior constructor que teníamos definido. Para ello hace uso de: **this()**

Así pues cuando hablemos de la palabra `this` como un método: **`this()`** estamos llamando a otro constructor que Java elegirá según los parámetros que se le pasen.

Es importante que tengamos en cuenta que el uso de esa instrucción se debe realizar en la primera línea del constructor

● **Práctica 13:** Crea con el código de ejemplo la clase `Coche` en el IDE y modifica el segundo constructor haciendo que la primera línea no sea la llamada a: `this()` (por ejemplo declara una variable local, etc) Toma captura de pantalla del mensaje y contesta ¿ qué mensaje muestra el IDE ?

● **Práctica 14:** Modifica la clase `Camion` de la práctica 12 de tal forma que los diferentes constructores se apoyen en uno solo haciendo uso de: `this()`

● **Práctica 15:** Crea la clase `Complejo` que sirva para utilizar números complejos. Estos números disponen de una parte real y una parte imaginaria (atributos `double real`, `double imag`) Dispondrá de tres constructores:
`Complejo(double real, double imag)`
`Complejo(double real)` → aquí se establecerá la parte imaginaria a 0
`Complejo()` → aquí se establecerán tanto la parte real como la imaginaria a 0

Constructor de Copia

Hemos visto que cuando igualamos variables que están apuntando a objetos (recordar el ejemplo con objetos de tipo Persona `d=c`) lo que hacemos realmente es copiar la referencia al objeto, no el propio objeto en sí.

Imaginemos que hayamos creado un objeto de la clase Coche con atributos: marca, modelo, potencia, numeroPuertas, matrícula Si quisiéramos reflejar dos coches fabricados de la misma tirada tendrán los dos los mismos atributos salvo matrícula sería cómodo poder hacer una copia de un coche a otro en lugar de que una referencia que es lo que nos brinda la posibilidad java cuando igualamos las variables. Siendo así dos objetos realmente diferentes en lugar de dos referencias al mismo objeto podremos manipular cada uno de forma separada sin que afecten uno a otro.

Ej

```
class Coche{
    private String marca;
    private String modelo;
    private double potencia;
    private int puertas;
    private String matricula;
    Coche(String marca, String modelo, double potencia, int puertas, String
matricula){
        this.marca = marca;
        this.modelo = modelo;
        this.potencia = potencia;
        this.puertas = puertas;
        this.matricula = matricula;
    }
}
```

Para conseguir ese efecto que hemos nombrado es para lo que se usa el constructor de copia.

El constructor de copia es un constructor que lo que recibe es un objeto de la misma clase y crea un nuevo objeto con la información que toma del objeto que se le ha pasado.

Ej.

```
Coche(Coche coche){
    this.marca = coche.marca;
    this.modelo = coche.modelo;
    this.potencia = coche.potencia;
    this.puertas = coche.puertas;
    this.matricula = coche.matricula;
}
```

Así pues la clase con los dos constructores nos quedaría:


```
class Coche{
    private String marca;
    private String modelo;
    private double potencia;
    private int puertas;
    private String matricula;
    Coche(String marca, String modelo, double potencia, int puertas, String
matricula){
        this.marca = marca;
        this.modelo = modelo;
        this.potencia = potencia;
        this.puertas = puertas;
        this.matricula = matricula;
    }

    Coche(Coche coche){
        this.marca = coche.marca;
        this.modelo = coche.modelo;
        this.potencia = coche.potencia;
        this.puertas = coche.puertas;
        this.matricula = coche.matricula;
    }
}
```

Y llamaríamos al constructor por ejemplo así:

```
{
    Coche cocheDePepe = new Coche("Peugeot", "205", 120, 4, "JTA-0427");
    Coche cocheDeAna = new Coche(cocheDePepe);
}
```

Así tenemos que cocheDePepe apunta a un objeto Coche y cocheDeAna apunta a otro objeto Coche, pero ambos tienen los mismos valores en los atributos.

 **Práctica 16:** Crear un constructor de copia para la clase Camion de la práctica 14 y un constructor de copia para la clase Complejo de la práctica 15

- **Práctica 17:** Crear una clase Factura que tenga como mínimo los atributos: double total, String detalle y los métodos agregar(String articuloConsumido, double precio) e imprimir() así como un constructor de copia

agregar() permite añadir al String detalle el nombre del artículo que se ha consumido y el precio de ese artículo a la vez que añade al total el precio.

imprimir() devuelve un String que muestra los artículos consumidos y el precio de cada uno de ellos así como el total de la factura

- **Práctica 18:** Crear una clase DNI con atributo: int dniNum y atributo String letrasPosibles

el atributo letrasPosibles tiene por valor: "TRWAGMYFPDXBNJZSQVHLCKE" que nos sirve para calcular la letra de un dni sabiendo su número. Basta con dividir el número del DNI por 23 y tomar el resto. La letra que ocupe la posición correspondiente en el String letrasPosibles es la letra apropiada

<https://www.letranif.com/formula-para-calcular-la-letra-del-dni/>

Crear los métodos: validarDNI(String dni) calcularLetra(int num)

validarDNI() verifica que un DNI tenga la letra que le corresponde (para este método se puede hacer uso de: Integer.parseInt() para extraer un número de un texto y substring() que nos sirve para extraer la subcadena que contiene el número sin incluir la letra

calcularLetra() devuelve un char con la letra correspondiente al número que se le pase.

Crear un constructor que reciba un entero que será el número del DNI, y un constructor de copia

static (palabra reservada)

Mediante la palabra reservada static podemos modificar el comportamiento de un atributo o de un método.

- Un método estático es un método que puede ser usado directamente desde la clase, sin necesidad de tener que crear una instancia para poder utilizar al método. También son conocidos como métodos de clase, frente a los métodos de objeto (es necesario un objeto para poder disponer de ellos). Los métodos estáticos no pueden manipular atributos de instancias (objetos) sino atributos estáticos (de clase) y suelen ser utilizados para realizar operaciones comunes a todos los objetos de la clase. Tampoco pueden acceder a **this**

Ej. Math.random(), Math.abs(), Integer.parseInt()

Entendiendo static en métodos y con lo que ya hemos visto ya se puede entender la declaración del main: (public static void main())

public, porque puede invocarse desde cualquier sitio (el IDE, la consola del sistema operativo...), **void** porque no devuelve valor alguno, y **static** porque puede ejecutarse sin instanciar un objeto de la clase que lo contiene. ¿Qué objeto iba a instanciar el objeto, si el programa empieza precisamente por aquí?

- Un atributo estático es un atributo común para toda la clase. Mientras que si declaramos un atributo normal tiene lugar que cada objeto de la clase poseerá su propia copia del atributo, si es un atributo static sólo hay una copia del atributo para toda la clase. Esto puede ser interesante en diferentes circunstancias. Pongamos un ejemplo:

```
class VolumenLibro{
    private static int totalVolumenes=0;
    private String estadoConservacion;
    private String propietario;
    private int id;
    VolumenLibro(String estadoConservacion, String propietario){
        totalVolumenes++;
        id = totalVolumenes;
        this.estadoConservacion = estadoConservacion;
        this.propietario = propietario;
    }
}
```

Cada vez que se crea un nuevo VolumenLibro se incrementa totalVolumenes y se puede tener un contéo del número total de volúmenes que se han creado de un libro.

Algo muy habitual es que las constantes que deba conservar una clase se declaren como static final.

static permite conservar una única copia para todos los objetos creados redundando en ahorro en memoria, final para que sea precisamente una constante.

● **Práctica 19:** Modificar la clase DNI para establecer los modificadores (final, static) que consideres necesario. Toma captura de pantalla de como queda la clase y explica por qué motivo consideras que es apropiado haber puesto los modificadores que has empleado

Sobrecarga de métodos

Es posible tener varias versiones de un mismo método (varios métodos con el mismo nombre) gracias a la sobrecarga de métodos.

El lenguaje Java soporta la característica conocida como sobrecarga de métodos. Ésta permite declarar en una misma clase varias versiones del mismo método con el mismo nombre. La forma que tendrá el compilador de distinguir entre varios métodos que tengan el mismo nombre será mediante la lista de parámetros del método: si el método tiene una lista de parámetros diferente, será considerado como un método diferente (aunque tenga el mismo nombre) y el analizador léxico no producirá un error de compilación al encontrar dos nombres de método iguales en la misma clase.

Imagínate que estás desarrollando una clase para escribir sobre un lienzo que permite utilizar diferentes tipografías en función del tipo de información que se va a escribir. Es probable que necesitemos un método diferente según se vaya a pintar un número entero (int), un número real (double) o una cadena de caracteres (String). Una primera opción podría ser definir un nombre de método diferente dependiendo de lo que se vaya a escribir en el lienzo. Por ejemplo:

Método pintarEntero (int entero).
Método pintarReal (double real).
Método pintarCadena (double String).
Método pintarEnteroCadena (int entero, String cadena).

Y así sucesivamente para todos los casos que desees contemplar...

La posibilidad que te ofrece la sobrecarga es utilizar un mismo nombre para todos esos métodos (dado que en el fondo hacen lo mismo: pintar). Pero para poder distinguir unos de otros será necesario que siempre exista alguna diferencia entre ellos en las listas de parámetros (bien en el número de parámetros, bien en el tipo de los parámetros). Volviendo al ejemplo anterior, podríamos utilizar un mismo nombre, por ejemplo pintar, para todos los métodos anteriores:

Método pintar (int entero).
Método pintar (double real).
Método pintar (double String).
Método pintar (int entero, String cadena).

En este caso el compilador no va a generar ningún error pues se cumplen las normas ya que unos métodos son perfectamente distinguibles de otros (a pesar de tener el mismo nombre) gracias a que tienen listas de parámetros diferentes.

Lo que sí habría producido un error de compilación habría sido por ejemplo incluir otro método pintar (int entero), pues es imposible distinguirlo de otro método con el mismo nombre y con la misma lista de parámetros (ya existe un método pintar con un único parámetro de tipo int).

También debes tener en cuenta que el tipo devuelto por el método no es considerado a la hora de identificar un método, así que un tipo devuelto diferente no es suficiente para distinguir un método de otro. Es decir, no podrías definir dos métodos exactamente iguales en nombre y lista de parámetros e intentar distinguirlos indicando un tipo devuelto diferente. El compilador producirá un error de duplicidad en el nombre del método y no te lo permitirá.

Sobrecarga de operadores.

Del mismo modo que hemos visto la posibilidad de sobrecargar métodos (disponer de varias versiones de un método con el mismo nombre cambiando su lista de parámetros), podría plantearse también la opción de sobrecargar operadores del lenguaje tales como +, -, *, (), <, >, etc. para darles otro significado dependiendo del tipo de objetos con los que vaya a operar.

En algunos casos puede resultar útil para ayudar a mejorar la legibilidad del código, pues esos operadores resultan muy intuitivos y pueden dar una idea rápida de cuál es su funcionamiento.

Un típico ejemplo podría ser el de la sobrecarga de operadores aritméticos como la suma (+) o el producto (*) para operar con fracciones. Si se definen objetos de una clase Fracción (que contendrá los atributos numerador y denominador) podrían sobrecargarse los operadores aritméticos (habría que redefinir el operador suma (+) para la suma, el operador asterisco (*) para el producto, etc.) para esta clase y así podrían utilizarse para sumar o multiplicar objetos de tipo Fracción mediante el algoritmo específico de suma o de producto del objeto Fracción (pues esos operadores no están preparados en el lenguaje para operar con esos objetos).

En algunos lenguajes de programación como por ejemplo C++ o C# se permite la sobrecarga, pero no es algo soportado en todos los lenguajes. En Java no soporta la sobrecarga de operadores.

En el ejemplo anterior de los objetos de tipo Fracción, habrá que declarar métodos en la clase Fracción que se encarguen de realizar esas operaciones, pero no lo podremos hacer sobrecargando los operadores del lenguaje (los símbolos de la suma, resta, producto, etc.). Por ejemplo:

```
public Fraccion sumar (Fraccion sumando)
```

```
public Fraccion multiplicar (Fraccion multiplicando)
```

Y así sucesivamente...

● **Práctica 20:** Crear una clase Reloj con atributos: int hora, int minuto, int segundo crear un constructor como mínimo con los 3 parámetros. Un constructor de copia, un método: String mostrar() que devolverá un texto con los datos del reloj en el formato: hora:minuto:segundo. Por ejemplo: 20:17:00 los metodos agregarMinutos(int), agregarHoras(int), agregarSegundos(int) que sumarán lo correspondiente al reloj. Observar que cuando supere 60minutos, 60segundos se agrega en la siguiente unidad. Sobrecargar los métodos agregarHoras(double) , agregarMinutos(double) que si tienen una parte decimal calcula el equivalente en la unidad correspondiente y lo agrega

● **Práctica 21:** continuar desarrollando la clase Complejo. Ahora tendrá un método mostrar() que devolverá una String representando el número. Y dos métodos: sumar(double) sumar(Complejo) que devolverán el complejo correspondiente a la suma
Nota: la suma de dos números complejos es sumar sus partes reales por un lado y por otro sus partes imaginarias
Ej.
Complejo c1 = new Complejo(2,0);
Complejo c2 = new Complejo(0,2);
Complejo c1mas10 = c1.sumar(10);
Complejo c1masc2 = c1.sumar(c2);

● **Práctica 22:** En esta ocasión se emula la multiplicación de números complejos. Métodos: multiplicar(double) multiplicar(Complejo)
Nota: Multiplicar un double por un complejo es multiplicar por ese número la parte real y la parte imaginaria.
Multiplicar dos números complejos $c1(m,n) * c2(o,p)$
parte real: $m*o - n*p$
parte imaginaria: $m*p + n*o$

La clase Object

La clase Object es la raíz de la jerarquía de clases de Java.

Esto quiere decir que cualquier otra clase creada en Java siempre es una subclase de Object de algún modo. Y, por lo tanto, heredará los atributos y métodos de Object. Se hablará más ampliamente de herencia más adelante, pero es un concepto simple: cuando una clase es hija de otra, hereda los atributos y métodos de su padre. Es decir, dispone de ellos sin necesidad de volver a escribirlos.

Pues bien, la clase Object, la madre de todas las clases de Java, dispone de varios métodos genéricos muy útiles y los pasa como herencia a todas sus subclases. Es decir, a cualquier otra clase, incluyendo las tuyas. Esos métodos son:

equals(): para comparar dos objetos

Si observamos en el IDE cuando ponemos a comparar dos String utilizando == el IDE nos propone que sustituyamos con equals. En general este método debe ser sobrescrito porque equals() al no tener información de como debe ser la comparación apropiada para considerar dos objetos

iguales lo que hace es mirar que compartan la misma ubicación en memoria. En el caso de String ya está realizado de forma apropiada y es la forma que debemos utilizar preferentemente a ==

Cuando se trabaja con dos literales de tipo String el compilador de Java nos hace bien la comparación con == pero en general es mejor idea utilizar equals() para String

● **Práctica 23:** Probar en el IDE las siguientes comparaciones. Obtener una salida en pantalla del valor booleano correspondiente y explicar por qué motivo la comparación nos sale true o false:

```
new String("test").equals("test")
```

```
new String("test") == "test"
```

```
new String("test") == new String("test")
```

```
"test" == "test"
```

toString(): devuelve el nombre de la clase

Cuando imprimimos objetos en pantalla se está usando internamente el método toString() Si recordamos nos muestra el nombre del tipo del objeto (la clase) y la dirección de memoria donde reside el objeto.

Este es un método que se suele sobrescribir para que muestre información más completa del objeto.

● **Práctica 24:** crear el método: String toString() para las clases que hemos creado en las actividades de este tema

● **Práctica 25:** crear el método boolean equals(Complejo) para la clase Complejo Que determinará que son iguales si las partes reales son iguales y las partes imaginarias son iguales.

● **Práctica 26:** Crear la clase CocheAlquiler como mínimo con atributos: String matricula, marca, modelo; double precio; int numDias; boolean alquilado; y la fecha de alquiler reflejada en tres valores enteros: dd, mm, aa
Se deberá reflejar que hay una cantidad mínima de días de alquiler: 2
Un constructor como mínimo que reciba matricula, marca, modelo y precio.
Un método: double alquilar(int dias, int dd, int mm, int aa) que refleja la fecha de alquiler y el número de días del alquiler. En este método se deberá controlar si el coche ya está alquilado en cuyo caso devolverá -1. -2 si se intenta alquilar por menos días del mínimo establecido y el coste del alquiler en otro caso
Un método: double devolver(int dias) que refleja que se ha devuelto el coche después de una cantidad de días. Este método devuelve -1 si el coche no está alquilado y la cantidad de sobre coste que pudiera haber incurrido si sobrepasa los días de alquiler inicialmente establecidos
Sobreescribir el método toString() que muestre los datos relevantes del alquiler

javadoc

Si redactas los comentarios de tus clases y métodos según un determinado estándar, la documentación de tu API se podrá generar de forma totalmente automática. A este formato de comentarios se le denomina javadoc, porque es el nombre de la utilidad que genera la documentación de forma automática.

Javadoc se ha extendido de tal modo que es un estándar de facto en la industria, utilizándose en la actualidad en desarrollos llevados a cabo con otros muchos lenguajes de programación, no solo Java.

Para generar APIs con Javadoc han de usarse etiquetas (tags) precedidas por el carácter "@". Estas etiquetas deben escribirse al principio de cada clase, atributo o método mediante un comentario iniciado con `/**` y acabado con `*/`. Tan sencillo como eso. Después, la aplicación javadoc las reconocerá y generará un documento HTML con la documentación de la API completa. A continuación se muestran algunos de los tags más comunes

Tag	Descripción
@author	Nombre del desarrollador.
@deprecated	Indica que el método o clase es antigua y que no se recomienda su uso porque posiblemente desaparecerá en versiones posteriores.
@param	Definición de un parámetro de un método, es requerido para todos los parámetros del método.
@return	Informa de lo que devuelve el método, no se puede usar en constructores o métodos "void".
@see	Asocia con otro método o clase.
@throws	Excepción lanzada por el método
@version	Versión del método o de la clase

Ejemplo: Escribir un algoritmo que sume todos los números naturales de n hasta m, siendo n y m números recibidos como parámetros. Devuelve la suma si todo ha ido bien o -1 en caso de error.

```
/**
 * Suma todos los números naturales entre 1 y 1000
 * @version: 1.0
 * @author: nombre
 * @param: n int número inicial de la secuencia
 * @param: m int número final de la secuencia
 * @return int la suma si todo funciona bien, -1 en caso de fallo
 */
public int sumarNumeros(int n, int m) {

    int i;
    int suma; // Variable acumulador
    if (n <= m) { // Comprobamos los límites
        suma = 0;
        for (i = n; i <= m; i++) {
            suma = suma + i;
        }
    }
    else { // Si n = m, tenemos un error
        suma = -1; // En caso de error, devolveremos -1
    }
    return suma;
}
```

Este es un ejemplo de algoritmo comentado usando el estándar javadoc. Observa el comentario al principio del método. Además, en el interior del método, aparecen comentarios adicionales que no son de javadoc. Se han escrito a la derecha de las instrucciones. A efectos de ejecución, se ignora todo lo que haya escrito entre los símbolos `/*` y `*/` o a la derecha de los símbolos `//`

● **Práctica 27:** Crear para todos los métodos creados en la clase `CocheAlquiler` documentación javadoc de los parámetros, lo que devuelve y lo que hace el método

● **Práctica 28:** Crear para todos los métodos creados en la clase `Complejo` documentación javadoc de los parámetros, lo que devuelve y lo que hace el método

● **Práctica 29:** Crear para todos los métodos creados en la clase Complejo documentación javadoc de los parámetros, lo que devuelve y lo que hace el método

● **Práctica 30:** Esta es una práctica bastante completa respecto a creación de clases. Se pretende recrear las operaciones de los vectores libres del plano. Se precisa crear una clase Punto (generarla en un fichero aparte) que tiene dos atributos: final double x, final double y que reflejan que un punto del plano es inamovible una vez creado. Precisa de un constructor de copia y del constructor: Punto(double x, double y) se deben sobrescribir los métodos: toString(), equals(Punto) el toString muestra el punto en el formato: (x,y) el método equals determina que dos puntos son iguales si coincide su valor x y su valor y por supuesto también los getter y setter

Crear la clase VectorLibre. Para ello debemos saber:

- Los vectores se definen por dos puntos. Uno inicial y uno final. Al ser un vector libre estableceremos para todos los vectores como el punto (0,0) siendo ese uno de los atributos, que lo llamaremos: origen

El otro atributo será: fin que es el otro punto que define el vector

- Dos constructores: VectorLibre(Punto desde, Punto hasta) y VectorLibre(Punto fin)

El segundo constructor lo único que precisa es tomar el punto fin dado como el atributo de nuestro nuevo vector

El primer constructor calcula las coordenadas del punto fin a crear como atributo mediante:

coordenada fin X: hasta.getX() - desde.getX()

coordenada fin Y: hasta.getY() - desde.getY();

métodos:

double modulo() → que devuelve la raíz cuadrada de x^2+y^2

VectorLibre multiplicar(double numero) → el producto por un número es multiplicar sus componentes: $V(3,8) * 2 = V(3*2,8*2)$

VectorLibre division(double numero) → La división por un número es dividir sus componentes: $V(4,8) / 2 = V(4/2,8/2)$

VectorLibre normalizacion() → Normalizar un vector es dividirlo por su módulo

VectorLibre opuesto() → Obtener el opuesto de un vector es multiplicarlo por -1

VectorLibre suma(VectorLibre vectorLibre) → la suma de vectores se hace sumando sus componentes: $V(1,2) + V(3,4) = V(1+3,2+4)$

VectorLibre resta(VectorLibre vectorLibre) → la resta es igual que la suma pero restando en lugar de sumar. Si bien este método se realizará apoyándose en otros dos métodos anteriores ya que la resta se puede obtener como la suma del opuesto

adicionalmente los métodos toString() y equals() Respecto a este último como el punto origen lo estamos estableciendo para todos en (0,0) lo único que hay que comprobar es que los puntos fin sean iguales (para ello se usará el método equals() de la clase Punto que antes se habrá creado)

Probar el funcionamiento de la clase haciendo sumas, restas, etc.

● **Práctica 31:** Basándose en la clase Coche de la práctica 8 se ampliarán las funcionalidades de la misma con los métodos:

String arrancar() → el atributo encendido se pone a true y devuelve un texto que dirá:

"OK. Coche ha arrancado" o el mensaje:

"Dañas el coche. Ya estaba encendido"

String apagar() → similar a la anterior devolviendo mensajes OK o mensaje de que el coche ya estaba apagado

String subirFrenoDeMano() → este método activa el atributo boolean frenoDeManoPuesto y sus correspondientes mensajes OK o ya estaba puesto

String bajarFrenoDeMano() → como el anterior pero al contrario

String moverIzquierda(int distancia) → modifica el atributo posicion restando la distancia que se le pasa. Si bien se tiene que tener en cuenta que si el freno de mano está puesto no se moverá y el coche se apagará. Si el coche está apagado tampoco se moverá. Tampoco podrá desplazarse a posiciones negativas. En ese caso se quedará en la posición 0 Como mensaje devuelve el OK o el motivo por el que no se completó el desplazamiento

String moverDerecha(int distancia) → lo mismo que el anterior pero en lugar de restar la distancia, se suma

Sobreescribir toString() para que informe del estado del coche: freno, posicion, etc