

Clases y objetos

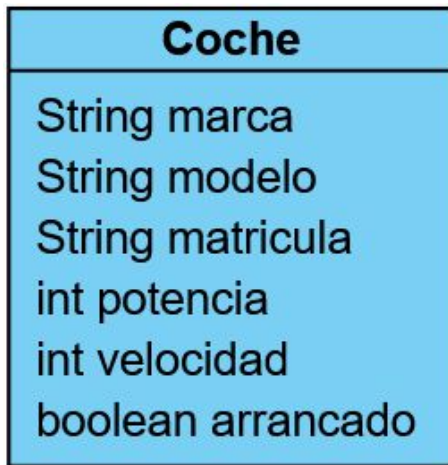
Esteban Álvarez

Concepto de clase. Problema

- Ya conocemos los **tipos de datos simples**
int, short, byte, double, long, char...
- Y el tipo de datos **String**, que es un poco más especial
- **Problema:** en nuestro programa queremos almacenar datos relativos a un coche, ¿cómo lo hacemos?
- La única **solución** posible ahora mismo es declarar y utilizar **variables**
 - `String marca, modelo, matricula;`
 - `int potencia, velocidad;`
 - `boolean arrancado;`

Concepto de clase. Solución

- Pero lo anterior **no es un coche**: son 6 variables sueltas, 3 String, 2 enteros y 1 boolean
- La mejor solución nos la dan **las clases** que permiten “empaquetar” esas 6 variables sueltas en un **objeto de tipo Coche**



Clases y objetos

- Por tanto, las clases son un **tipo de datos complejo** que:
 - están definidas en el propio lenguaje:
Scanner, String...
 - pueden ser definidas por el programador (por nosotros):
clase **Coche**, clase **Persona**, clase **Cliente**
- Y, como acabamos de ver, **agrupan** una serie de variables que definen esa clase y permiten crear **objetos** de esa clase
 - Se pueden crear varios coches diferentes
 - Todos tendrán las mismas características: `marca, modelo, matrícula, ...`
 - Pero cada uno tendrá diferentes valores para esas características

Clases y objetos

EJERCICIO

- Observa a tu alrededor e identifica algún objeto que veas
 - Da nombre a su clase
 - Identifica sus atributos
 - ¿Qué valores tienen esos atributos?

Clases y objetos

Para la mejor comprensión del tema se verán de forma conjunta las clases y los objetos.

- Las presentaciones que hablen de **clases** tendrán un fondo azul
- Las presentaciones sobre **objetos** tendrán un fondo color salmón

Concepto de clase

DEFINICIÓN DE CLASE

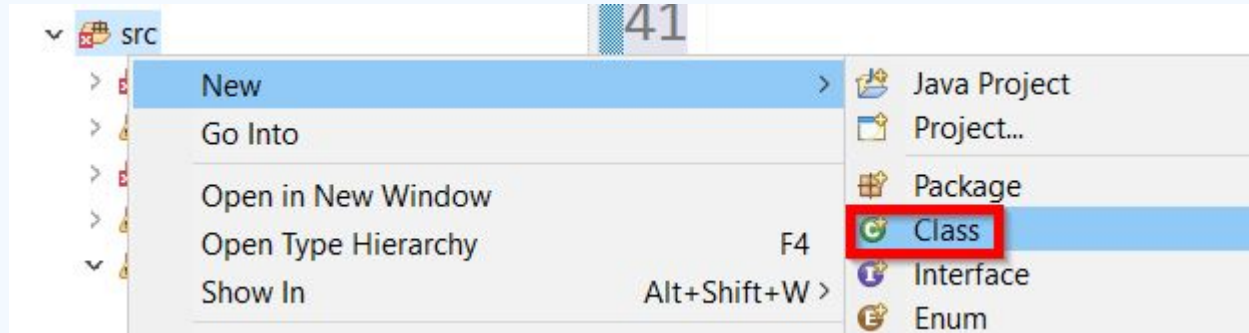
- Definición sencilla: Una clase es una **plantilla** que permite crear **objetos** según un **modelo** predefinido.
- **Modelar** es representar el mundo real en términos informáticos (programación o bases de datos por ejemplo)

Concepto de clase

- Para **modelar** una clase tenemos que
 1. Crearla y darle un nombre
 2. identificar sus propiedades: **atributos** (marca, modelo, matrícula...)
 3. identificar qué acciones o **métodos** utiliza (arrancar, frenar, acelerar...)
- Ya tenemos claro cuándo **necesitamos** crear una clase
 - El ejemplo del coche es perfecto: necesitamos almacenar información sobre uno o más coches

Creación de una clase

- Ya sabemos cómo crear una clase en Eclipse porque el Main es una clase:



Creación de una clase

- Le damos un nombre
 - con la **inicial en mayúsculas**. Coche ~~coche~~ ~~COCHE~~
 - en **singular** ~~CocheS~~ ~~COCHES~~

En Java las **clases** son el **único elemento** que se escribe con la inicial en mayúscula.

Por esta característica son **fácilmente diferenciables** del resto: variables, tipos de datos básicos, palabras reservadas, etc.

Creación de una clase

- Identifica los nombres de clases:

```
public class GrupoDepositos {  
    private Deposito deposito1;  
    private Deposito deposito2;  
    private Deposito deposito3;  
    private String idGrupo;  
    private int numeroDepositosGrupo;  
  
    public String getIdGrupo() { return idGrupo; }  
    public Deposito getDeposito1() { return deposito1; }  
    public void setDeposito1(Deposito depos) {deposito1=depos; }  
}
```

Creación de una clase

- Identifica los nombres de clases:

```
public class GrupoDepositos {  
    private Deposito depositos1;  
    private Deposito depositos2;  
    private Deposito depositos3;  
    private String idGrupo;  
    private int numeroDepositosGrupo;  
  
    public String getIdGrupo() { return idGrupo; }  
    public Deposito getDeposito1() { return depositos1; }  
    public void setDeposito1(Deposito depos) {depositos1=depos; }  
}
```

Creación de una clase

- Al contrario que hasta ahora, **no añadimos** el método main

Name:

Modifiers: ☒ public ☐ package ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

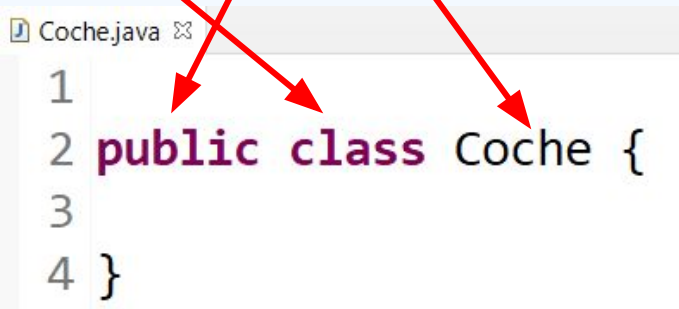
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

Creación de una clase

- Ya tenemos nuestra clase pública Coche, por ahora vacía



```
Coche.java ✕
1
2 public class Coche {
3
4 }
```

- Todo el contenido de la clase va entre las llaves { }

El nombre de la clase **debe coincidir** con el del fichero .java
(Coche=Coche.java)

Atributos

- Sabemos que necesitamos una clase para crear un paquete y “**agrupar**” variables
- Esas variables van a definir las **características** de la clase y reciben el nombre de **ATRIBUTOS** o **propiedades**
- El primer paso del modelado de clases consiste en **identificar los atributos** de la clase que estamos creando

Atributos

- Para ello tenemos que pensar en el objeto que queremos modelar y **decidir cuáles de sus propiedades** representaremos en nuestra aplicación
- Para la clase `Coche` me interesa almacenar:
 - Marca, modelo, matrícula, potencia, velocidad y estado (arrancado o no)
- Pero un coche en el mundo real tiene **otras muchas características** que no me interesan en este momento:
 - Número de bastidor, número de plazas, cilindrada, combustible, color...

El número y tipo de atributos de una clase **dependerá del problema** al que queramos dar solución a través de la programación

Atributos

EJERCICIO

- Piensa en los atributos que necesitarías para modelar en Java las siguientes clases:
 - Persona
 - Alumno
 - Animal
 - Mascota
 - Ascensor
 - Alimento

Atributos

EJERCICIO

- Piensa en los atributos que necesitarías para modelar en Java las siguientes clases:
 - Persona: nombre, apellidos, dni, fechaNacimiento...
 - Alumno: Los anteriores más curso, grupo, asignaturas..
 - Animal: raza, nombreCientífico, edad...
 - Mascota: Los anteriores más nombre, vacunado...
 - Ascensor: marca, tipo, motor, piso, maxPiso, minPiso..
 - Alimento: nombre, marca, tipo, fechaCaducidad...

Atributos

- Los atributos se representan en una clase **como variables** dentro de esa clase
 - **Sólo se declaran, no se inicializan**
- EJERCICIO: Declara los atributos de la clase `Coche`

Coche
String marca
String modelo
String matricula
int potencia
int velocidad
boolean arrancado

Atributos

- EJERCICIO: Declara los atributos de la clase Coche

```
public class Coche {  
    //1.ATRIBUTOS  
    //Se ponen al inicio de la clase  
    //Sólo se declaran, NUNCA se inicializan  
    String marca, modelo, matricula;  
    int potencia, velocidad;  
    boolean arrancado;  
}
```

Atributos

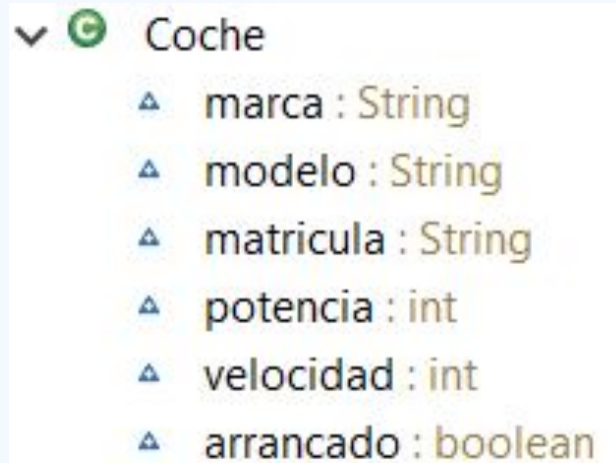
- EJERCICIO: Declara los atributos de la clase Coche

```
public class Coche {  
    String marcaCoche, modeloCoche, matriculaCoche;  
    int potenciaCoche, velocidadCoche;  
    boolean arrancadoCoche;  
}
```

Poner como sufijo el nombre de la clase **no es una buena práctica**. Ya se sobreentiende que un atributo `marca` que pertenece a la clase `Coche`, es la marca de un Coche

Atributos

- Eclipse muestra un resumen de los atributos en la parte derecha de la ventana:



Atributos

- Con esa clase `Coche` creada, ya podemos **crear objetos de tipo `Coche` en otras clases**
- Lo haremos en la clase principal

Declaración y creación de objetos

- Un objeto se **declara igual** que cualquier variable:

`TipoDeDatos nombreDelObjeto;`

- El tipo de datos es el **nombre de la clase**
- EJERCICIO: Crea la clase `PrincipalCoches` y **declara** un objeto de tipo `Coche` y nombre `coche`

Declaración y creación de objetos

- EJERCICIO: Crea la clase `PrincipalCoches` y **declara** un objeto de tipo `Coche` y nombre `coche`

```
public class PrincipalCoches {  
    public static void main(String[] args) {  
        //Declaración del objeto coche  
        //de tipo Coche  
        Coche coche;  
    }  
}
```

Observa que la inicial en mayúscula es la forma de diferenciar el objeto de la clase

Declaración y creación de objetos

- En este momento el objeto `coche` está **declarado** pero no está creado: todavía no existe como tal
- Con la declaración sólo hemos hecho una “reserva”: voy a utilizar un objeto de tipo `Coche` y nombre `coche`
- Para crearlo hay que **asignarle** **new** `NombreDeLaClase()`;
- EJERCICIO: Crea el objeto `coche`

Declaración y creación de objetos

- EJERCICIO: Crea el objeto `coche`

```
Coche coche;  
coche=new Coche();
```

- Es muy habitual **declarar y crear** un objeto en la misma línea

```
Coche coche=new Coche();
```

A la creación de objetos se le denomina **instanciación** de objetos.

`coche` es una **instancia** de la clase `Coche`

Declaración y creación de objetos

- Cuando se crea un objeto con **new** se crea **una copia** basándose en la plantilla (Clase)
- Por tanto, con **new** `Coche()` estamos creando un “paquete” que va a tener: `marca, modelo, matrícula, potencia, velocidad y arrancado`
- Al principio el **valor de los atributos** de un objeto es
 - **0** si es un tipo de datos numérico (entero o real)
 - **false** si es boolean
 - **null** si es un objeto (String)

```
Coche coche=new Coche();
```

Declaración y creación de objetos

- Depurando, podemos comprobarlo:

▼ ⓘ coche	Coche (id=27)
▶ arrancado	false
▶ marca	null
▶ matricula	null
▶ modelo	null
▶ potencia	0
▶ velocidad	0

Declaración y creación de objetos

- Si creamos un segundo coche, `coche2`:
 - Se creará también a partir de la misma **plantilla**: tendrá los mismos atributos
 - El **valor** de los atributos es el mismo (por ahora)
 - Pero **son dos objetos diferentes**

Dos personas con el mismo nombre, mismos apellidos, mismo DNI, etc. por mucho que tengan sus datos iguales, **son dos personas diferentes**

Declaración y creación de objetos

- Una vez mas, depurando vemos esa diferencia:
 - `coche` y `coche2` tienen los mismos atributos
 - incluso los mismos valores
 - pero son dos coches diferentes: tienen **distinto id**

▼ ⓘ coche	Coche (id=22)
▲ arrancado	false
▲ marca	null
▲ matricula	null
▲ modelo	null
▲ potencia	0
▲ velocidad	0
▼ ⓘ coche2	Coche (id=25)
▲ arrancado	false
▲ marca	null
▲ matricula	null
▲ modelo	null
▲ potencia	0
▲ velocidad	0

Declaración y creación de objetos

Cuando creamos un objeto, se crea **una copia** basándose en la plantilla descrita por **la clase**.

Fíjate cómo los dos coches tienen los mismos atributos

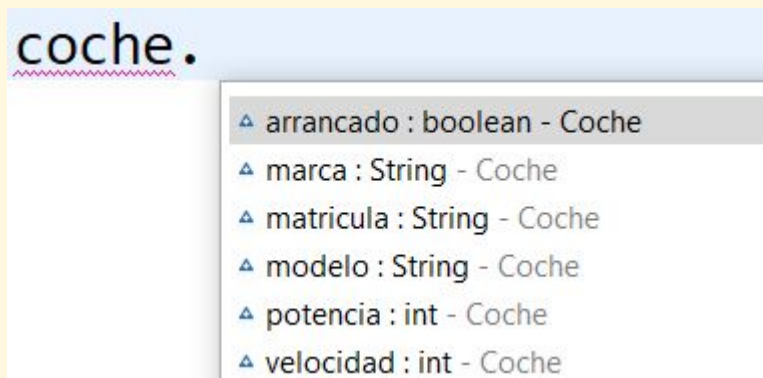
▼ ⓘ coche	Coche (id=22)
▲ arrancado	false
▲ marca	null
▲ matricula	null
▲ modelo	null
▲ potencia	0
▲ velocidad	0
▼ ⓘ coche2	Coche (id=25)
▲ arrancado	false
▲ marca	null
▲ matricula	null
▲ modelo	null
▲ potencia	0
▲ velocidad	0

Uso de objetos

- Ahora que ya tenemos creado un coche el primer uso lógico es **asignarle valores a sus atributos**
- Para acceder a los atributos de un objeto se utiliza el **operador punto .**
- Sintaxis:

`objeto.atributo`

- Por ejemplo:



Uso de objetos

- El **uso** que podemos dar al atributo de un objeto es el mismo que el de cualquier variable:

- **Asignarle** un valor

```
//Accedo a la matrícula de "coche"
```

```
//y le asigno un valor
```

```
coche.matricula="1234ABC";
```

- **Obtener** su valor

```
//Accedo a la matrícula de "coche"
```

```
//y muestro su valor (1234ABC) por pantalla
```

```
System.out.println(coche.matricula);
```

Uso de objetos

- Al asignarle una matrícula a `coche` con `coche.matricula="1234ABC"`; el objeto queda de la siguiente forma:

▼ ⓘ coche	Coche (id=22)
▶ arrancado	false
▶ marca	null
> ▶ matricula	"1234ABC" (id=25)
▶ modelo	null
▶ potencia	0
▶ velocidad	0

Uso de objetos

- EJERCICIO: Pide información al usuario para que introduzca datos para todas las propiedades de `coche` y `coche2`.
Una vez hecho lo anterior, muestra por pantalla los valores de las propiedades de los dos coches.

Uso de objetos

```
System.out.println("Marca del coche1");  
coche.marca=sc.nextLine();  
System.out.println("Matrícula del coche1");  
coche.matricula=sc.nextLine();  
...  
System.out.println("Datos de coche1. Marca: "+coche.marca+ ...);
```

Uso de objetos

- El acceso directo a los atributos de un objeto **no es una buena práctica**
- Para cambiar ese acceso hay que hacer unas **modificaciones** en la clase

Encapsulación

- Desde el punto de vista de la **seguridad**, que una clase cualquiera acceda directamente a los atributos de un objeto, no es correcto

El acceso directo a los atributos desde otra clase sería equivalente a ir al banco a retirar dinero y **tener acceso a la caja** para hacer esa operación.

Alguien se puede “confundir” y coger más dinero de la cuenta

Encapsulación

- Para proteger los atributos se utiliza la **encapsulación**: sólo la propia clase tiene acceso a los atributos

El banco pone a un trabajador de la compañía encargado de gestionar los accesos a la caja.

RESULTADO: Si el cliente quiere retirar dinero de su cuenta, es el banquero quien accede a la caja

Encapsulación

- La **encapsulación de los atributos** se hace añadiendo el modificador **private** delante de su declaración

```
public class Coche {  
    private String marca, modelo, matricula;  
    private int potencia, velocidad;  
    private boolean arrancado;  
}
```

Los atributos pueden ser: **public**, **private** o **sin modificador**. Public y sin modificador implica que se puede acceder a ellos desde otras clases

Encapsulación

- Pero ahora no podemos acceder a los atributos del objeto



Hemos protegido la caja pero no hemos contratado ningún banquero

Encapsulación

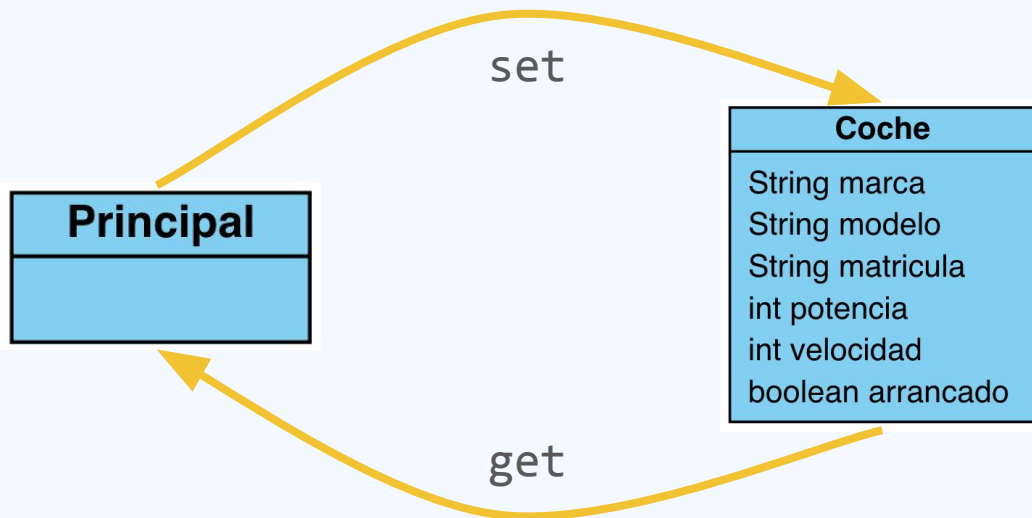
- La solución es crear **métodos** que gestionen el acceso a esos atributos
- Los métodos **definen las acciones** que puede hacer una determinada clase

Un coche tiene atributos (marca, modelo, matrícula) pero también se pueden **hacer cosas** con un coche: arrancar, acelerar, frenar, estacionar...

Esas acciones que puede hacer un coche **son los métodos** en programación

Encapsulación. Getters y Setters

- Los métodos que acceden a un atributo para leer su valor son los **getters** o métodos get
- Los métodos que acceden a un atributo para asignarle un valor son los **setters** o métodos set



Encapsulación. Getters y Setters

- Eclipse los genera automáticamente:

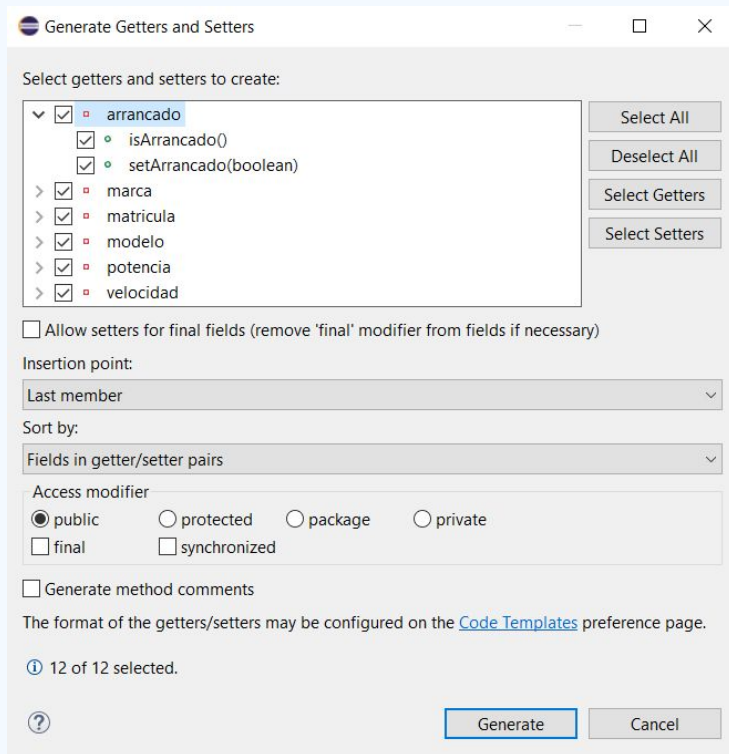
```
public class Coche {  
    //1.ATRIBUTOS  
    //Se ponen al inici  
    private String marca  
    private int potenci  
    private boolean arr
```

Show in breadcrumb	Alt+Shift+B
Quick Outline	Ctrl+O
Quick Type Hierarchy	Ctrl+T
Open With	>
Show In	Alt+Shift+W >
Cut	Ctrl+X
Copy	Ctrl+C
Copy Qualified Name	
Paste	Ctrl+V
Quick Fix	Ctrl+1
Source	Alt+Shift+S >

Sort Members...
Clean Up...
Override/Implement Methods...
Generate Getters and Setters...
Generate Delegate Methods...
Generate hashCode() and equals()...
Generate toString()...
Generate Constructor using Fields...
Generate Constructors from Superclass...
Externalize Strings...

Encapsulación. Getters y Setters

- Eclipse los genera automáticamente:



Encapsulación. Getters y Setters

- Eclipse los genera automáticamente:

```
//2.GETTERS Y SETTERS
public String getMarca() {
    return marca;
}
public void setMarca(String marca) {
    this.marca = marca;
}
public String getModelo() {
    return modelo;
}
public void setModelo(String modelo) {
    this.modelo = modelo;
}
public String getMatricula() {
    return matricula;
}
```

Encapsulación. Getters y Setters

- Explicación de un método **set**

```
public void setMarca(String marca) {  
    this.marca=marca;  
}
```

- Son **públicos** por lo que los pueden usar otras clases
- **No devuelven** datos. Se indica con la palabra reservada **void**
- Su nombre es **setAtributo**
- **Recibe un parámetro.** En este caso de tipo **String** al que va a llamar **marca**
- **Asigna** el **String marca** que recibe como **marca** del coche

Encapsulación. Getters y Setters

- Uso de un método **set**



`coche.matricula="1234ABC";`



`coche.setMatricula("1234ABC");`

Encapsulación. Getters y Setters

- Uso de un método **set**

```
coche.setMatricula("1234ABC");
```

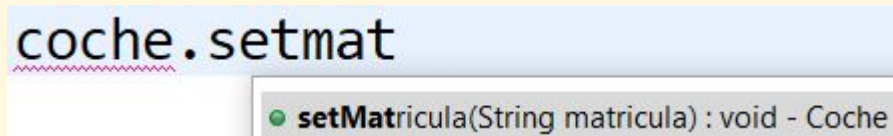
▼ ⓘ coche	Coche (id=27)
▲ arrancado	false
▲ marca	null
> ▲ matricula	"1234ABC" (id=29)
▲ modelo	null
▲ potencia	0
▲ velocidad	0

```
private String marca, modelo, matricula;  
private int potencia, velocidad;  
private boolean arrancado;  
  
public void setMatricula(String matricula) {  
    this.matricula = matricula;  
}
```

The diagram illustrates the encapsulation process. A red arrow originates from the string argument "1234ABC" in the `coche.setMatricula("1234ABC");` call. It points to the `matricula` parameter in the `setMatricula` method signature. From there, another red arrow points to the `this.matricula` assignment within the method body, and a final red arrow points to the `matricula` private attribute in the class fields, showing how the external value is passed through the setter to update the internal state.

Encapsulación. Getters y Setters

- Cuando utilizamos un método **set**, hay que fijarse en la **ayuda contextual** que ofrece Eclipse



- Método **setMatrícula**
- **Necesita** un String
- **No devuelve** datos
- Es de la **clase Coche**

Encapsulación. Getters y Setters


- Explicación de un método **get**


```
public String getMarca() {  
    return marca;  
}
```

- Son **públicos** por lo que los pueden usar otras clases
- **Devuelven** datos. Un String en este caso
- Su nombre es **getAtributo**
- **No reciben parámetros**. No necesitan información adicional para trabajar
- **Devuelven** el valor almacenado en un atributo del objeto

Encapsulación. Getters y Setters

- Uso de un método **get**

 `System.out.println(coche.matricula);`

 `System.out.println(coche.getMatricula());`

Encapsulación. Getters y Setters

- Uso de un método **get**

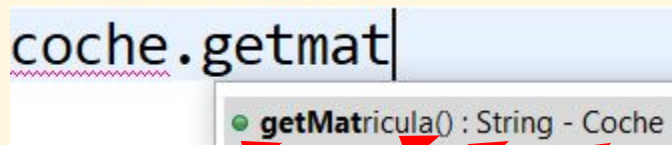
▼ coche	Coche (id=27)
▲ arrancado	false
▲ marca	null
> ▲ matricula	"1234ABC" (id=29)
▲ modelo	null
▲ potencia	0
▲ velocidad	0

```
System.out.println(coche.getMatricula());
```

```
private String marca, modelo, matricula;  
private int potencia, velocidad;  
private boolean arrancado;  
  
public String getMatricula() {  
    return matricula;  
}
```

Encapsulación. Getters y Setters


- Cuando utilizamos un método **get**, hay que fijarse en la **ayuda contextual** que ofrece Eclipse




- Método **getMatrícula**
- **No necesita** datos
- **Devuelve** un String
- Es de la **clase Coche**

Encapsulación. Getters y Setters

- Uso de un método **get**

 `System.out.println(coche.matricula);`

 `System.out.println(coche.getMatricula());`

Esto es un String con la matrícula

TRUCO: Cuando uses un get, sustituye mentalmente el código java por el **tipo de datos** que devuelve

Encapsulación. Getters y Setters

- Para los atributos booleanos, el método `get` se sustituye por `is`

```
public boolean isArrancado() {  
    return arrancado;  
}
```

Encapsulación. Getters y Setters

- Y su uso sería:

```
if(coche.isArrancado()) { //Si el coche está arancado...  
}
```

Encapsulación. Getters y Setters

EJERCICIO

- Encapsula los atributos de la clase Coche
- Genera los getters y setters para **todos** los atributos

IMPORTANTE: los atributos de una clase son **accesibles** desde los métodos de esa clase, aunque sean private

Encapsulación. Getters y Setters

EJERCICIO

- Modifica el programa principal para adaptarlo al uso de getters y setters

Encapsulación. Getters y Setters

- En el ejercicio anterior se generaron getters y setters para **todos** los atributos
- Al hacer esto, no hubiese sido necesario hacer que los atributos fuesen `private`
 - No hay ninguna diferencia porque podemos acceder a lectura y escritura de todos los atributos
- Pero ahora podemos decidir si **protegemos** un atributo:
 - contra lectura si no generamos el **método get**
 - contra escritura si no generamos el **método set**

IMPORTANTE: la verdadera encapsulación se produce protegiendo los atributos cuando **no se genera** un método get y/o set

Encapsulación. Getters y Setters

- En este ejemplo, si eliminamos (comentamos) el método setMarca, no se podrá modificar la marca de un coche una vez creado

```
// public void setMarca(String marca) {  
//     this.marca = marca;  
// }
```

`coche.setMarca("Ford");`

The method setMarca(String) is undefined for the type Coche

3 quick fixes available:

[Change to 'setMatricula\(..\)'](#)

[Create method 'setMarca\(String\)' in type 'Coche'](#)

[Add cast to 'coche'](#)



Referencia this

- En los setters aparece la palabra reservada **this** que quedó pendiente de explicación

```
public void setMarca(String marca) {  
    this.marca=marca;  
}
```

Referencia this

- **this** hace referencia al objeto actual, al objeto que usa el método en ese momento
- Y es la única forma de diferenciar el **String marca** que nos pasan del **atributo marca de la clase**
- Si no se utilizase this, no se estaría modificando el atributo: fíjate en los colores

```
public void setMarca(String marca) {  
    marca=marca;  
}
```

- A la variable `marca` que nos pasan, le estamos asignando el valor de `marca!!`

Referencia this

- **this** hace referencia al objeto actual

```
coche.setMarca("Seat");
```

```
coche2.setMarca("VW");
```

```
//this es "coche2". Por tanto, si se usa this, se está  
//modificando la marca de "coche2"  
public void setMarca(String marca) {  
    coche2.marca = marca;  
}
```

▼ ⓘ coche	Coche (id=28)
▪ arrancado	false
> ▪ marca	"Seat" (id=31)
▪ matricula	null
▪ modelo	null
▪ potencia	0
▪ velocidad	0
▼ ⓘ coche2	Coche (id=30)
▪ arrancado	false
> ▪ marca	"VW" (id=37)
▪ matricula	null
▪ modelo	null
▪ potencia	0
▪ velocidad	0

Ejercicio

- Crea la clase Alumno con los atributos **privados**:
 - Nombre de tipo String
 - Apellidos de tipo String
 - Edad de tipo entero
 - Nota1, nota2 y nota3 que son las notas de las tres asignaturas de las que se encuentra matriculado. De tipo real
 - Faltas de asistencia de tipo entero
- Genera getters y setters para los atributos

Ejercicio

Crea la clase “Principal.java” y crea dos alumnos: alumno1 y alumno2.

Después haz las siguientes tareas:

- Pedir al usuario datos para añadir toda la información al alumno1
- Pedir al usuario el nombre, los apellidos y la edad del alumno2
- Las notas del alumno2 serán las mismas que las del alumno1
- Mostrar todos los datos del alumno1 y el alumno2
- Mostrar el nombre y los apellidos del alumno más joven

Ejercicio

Crea la clase “PrincipalMenu.java”. Dentro de ella crea un objeto alumno vacío y añade un menú con las siguientes opciones:

1. Añadir datos personales. El usuario introducirá el nombre y los apellidos del alumno. No se permiten valores vacíos para el nombre o los apellidos. En caso de que alguno de los datos sea vacío, ninguno de ellos se añade al alumno
2. Añadir edad. Pedirá un entero con la edad. Validar que sea un valor positivo. Tratar la excepción
3. Añadir notas. Pide las 3 notas al usuario. Tratar la excepción
4. Mostrar datos de alumno. Muestra toda la información del alumno
5. Salir

Constructores. Constructor por defecto

- Los constructores son **métodos especiales** que permiten a la clase instanciar (crear) objetos de esa clase
- Tienen el **mismo nombre de la clase** y se utilizan con el operador **new** para crear el objeto
- Toda clase tiene un **constructor por defecto** `NombreClase()` que no es necesario programar

```
Coche coche=new Coche(); //Llamada al constructor por defecto
                          //de la clase Coche
```

Constructores. Constructor por defecto

- **No hemos codificado** el constructor `Coche()` en la clase `Coche`, sin embargo, el constructor existe y crea los objetos
- Esto ya se ha comentado pero es necesario recalcar que un constructor por defecto **inicializa el valor de los atributos** de un objeto a
 - **0** si es un tipo de datos numérico (entero o real)
 - **false** si es boolean
 - **null** si es un objeto (String)

```
Coche coche=new Coche();
```

coche	Coche (id=27)
arrancado	false
marca	null
matricula	null
modelo	null
potencia	0
velocidad	0

Constructores. Constructor por defecto

- La clase Coche **no tiene constructor** de forma explícita pero en realidad es como si tuviese el siguiente código:

```
public class Coche() {  
    public Coche() {  
        //No hay código pero se inicializan  
        //los atributos a 0, false o null  
    }  
}
```

IMPORTANTE: Los constructores se utilizan para crear objetos e **inicializar** los atributos (dar valores iniciales a los atributos cuando se crea un objeto)

Constructores. Constructor por defecto

- Aprovechando el código del constructor anterior es posible asignar a los atributos de un objeto **valores iniciales** diferentes a 0, false y null en su creación:


```
public Coche() { //Ahora todos los coches se crean con esos valores
    marca="Sin marca";
    modelo="Sin modelo";
    matricula="0000AAA";
    potencia=0;
    velocidad=0;
    arrancado=true;
}
```

▼ coche	Coche (id=28)
▣ arrancado	true
> ▣ marca	"Sin marca" (id=31)
> ▣ matricula	"0000AAA" (id=37)
> ▣ modelo	"Sin modelo" (id=38)
▣ potencia	0
▣ velocidad	0
▼ coche2	Coche (id=30)
▣ arrancado	true
> ▣ marca	"Sin marca" (id=31)
> ▣ matricula	"0000AAA" (id=37)
> ▣ modelo	"Sin modelo" (id=38)
▣ potencia	0
▣ velocidad	0

Constructores. Constructor por defecto

- **IMPORTANTE:** podemos decidir **no inicializar** alguno de los atributos del objeto
- En ese caso, el valor del atributo será el valor por defecto
 - 0, false o null dependiendo del tipo de datos del atributo

```
public Coche() {  
    marca = "Sin marca";  
    modelo = "Sin modelo";  
    matricula = "0000AAA";  
    potencia = 0;  
    velocidad = 0;  
    //arrancado = true;  
}
```



coche	Coche (id=27)
arrancado	false
> marca	"Sin marca" (id=28)
> matricula	"0000AAA" (id=34)
> modelo	"Sin modelo" (id=35)
potencia	0
velocidad	0

Constructores. Constructor por defecto

Utilizamos el constructor por defecto cuando **no nos importa** qué valores iniciales tendrán los atributos de un objeto.

Lo más probable es que luego queramos cambiarlos con los setters

Constructores. Constructor con parámetros

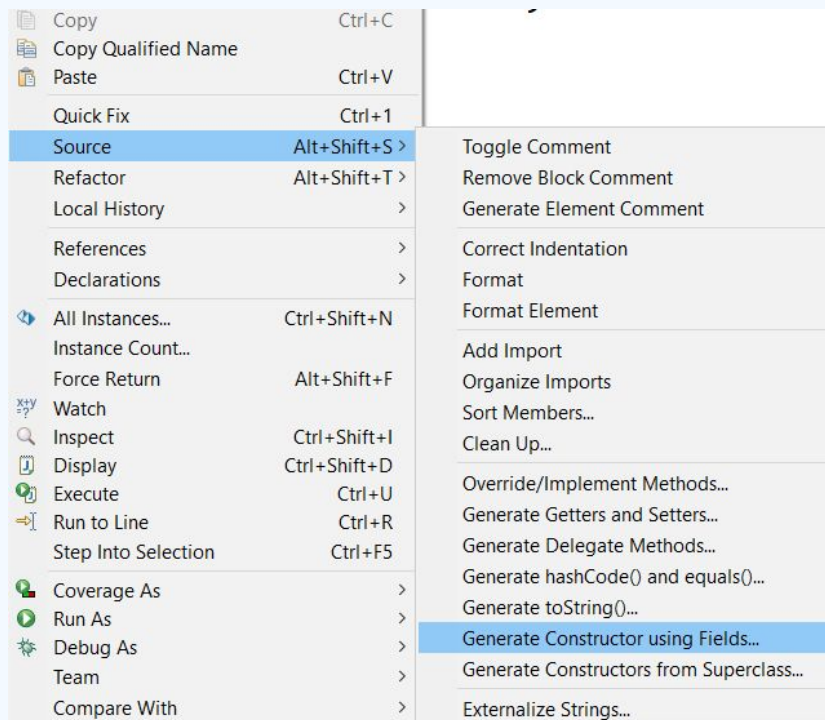
- Sin embargo, **habitualmente inicializaremos un objeto con unos valores determinados**
- Para eso tenemos los **constructores con parámetros** o constructores parametrizados

CONSTRUCTOR POR DEFECTO: Quiero crear un coche, no me importa qué valores tendrán sus atributos al inicio, luego los cambiaré (o no)

CONSTRUCTOR CON PARÁMETROS: Quiero crear un coche, con la marca “Seat”, modelo “Ibiza”, matrícula ...

Constructores. Constructor con parámetros

- De nuevo Eclipse nos da una herramienta para generar los constructores con parámetros:



Constructores. Constructor con parámetros

- Marcamos los campos que queremos inicializar con valores distintos a 0, false y null
- Recomendable marcar el check

Generate Constructor using Fields

Select super constructor to invoke:
Object()

Select fields to initialize:

<input checked="" type="checkbox"/>	marca
<input checked="" type="checkbox"/>	modelo
<input checked="" type="checkbox"/>	matricula
<input checked="" type="checkbox"/>	potencia
<input checked="" type="checkbox"/>	velocidad
<input checked="" type="checkbox"/>	arrancado

Select All
Deselect All
Up
Down

Insertion point:
After 'Coche()'

Access modifier
☒ public ☐ protected ☐ package ☐ private

☐ Generate constructor comments
☒ Omit call to default constructor super()

The format of the constructors may be configured on the [Code Templates](#) preference page.

6 of 6 selected.

Generate Cancel

Constructores. Constructor con parámetros

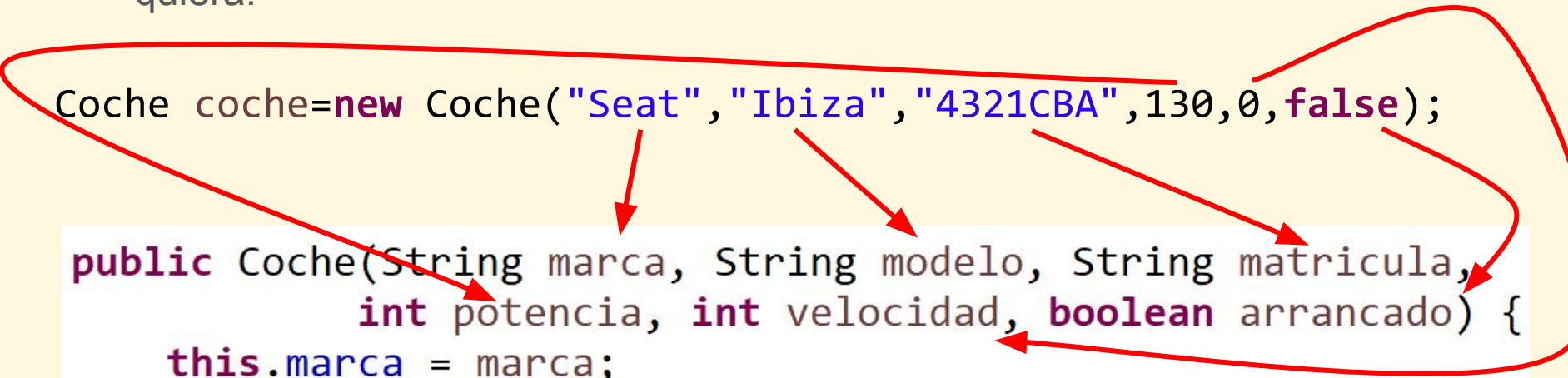
- Y se genera el código

```
public Coche(String marca, String modelo, String matricula,  
             int potencia, int velocidad, boolean arrancado) {  
    this.marca = marca;  
    this.modelo = modelo;  
    this.matricula = matricula;  
    this.potencia = potencia;  
    this.velocidad = velocidad;  
    this.arrancado = arrancado;  
}
```

Constructores. Constructor con parámetros

- Esto nos permite crear los objetos con los valores que nosotros o el usuario quiera:

```
Coche coche=new Coche("Seat","Ibiza","4321CBA",130,0,false);
```


A red oval encircles the constructor call line. Red arrows point from each argument in the call to its corresponding parameter in the constructor definition below. Specifically, an arrow points from "Seat" to "marca", "Ibiza" to "modelo", "4321CBA" to "matricula", "130" to "potencia", "0" to "velocidad", and "false" to "arrancado".

```
public Coche(String marca, String modelo, String matricula,  
              int potencia, int velocidad, boolean arrancado) {  
    this.marca = marca;  
    this.modelo = modelo;  
    this.matricula = matricula;  
    this.potencia = potencia;  
    this.velocidad = velocidad;  
}
```

Constructores. Constructor con parámetros

- Esto nos permite crear los objetos con los valores que nosotros o el usuario quiera:

Coche coche3=new Coche("VW", "Polo", "0000AAA", 95, 0, false);



The diagram consists of three red arrows pointing from the arguments in the constructor call above to the corresponding parameters in the constructor signature below. The first arrow points from "VW" to "marca", the second from "Polo" to "modelo", and the third from "0000AAA" to "matricula".

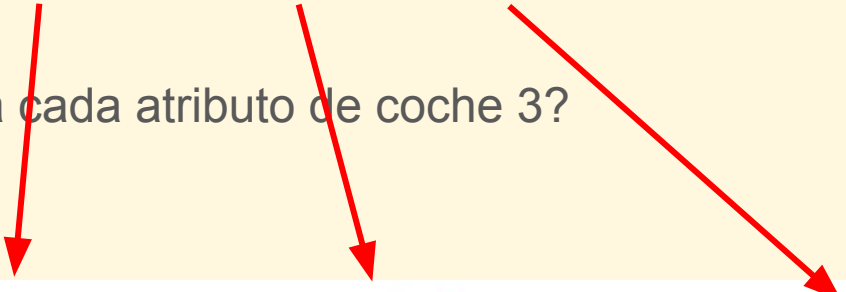
```
public Coche(String marca, String modelo, String matricula,  
             int potencia, int velocidad, boolean arrancado) {  
    this.marca = marca;  
    this.modelo = modelo;  
    this.matricula = matricula;  
    this.potencia = potencia;  
    this.velocidad = velocidad;  
}
```

Constructores. Constructor con parámetros

Es **muy importante** respetar el orden de los parámetros que pasamos al constructor

Coche coche3=new Coche("Polo", "0000AAA", "VW", 0, 95, false);

- ¿Qué valores se asignan a cada atributo de coche 3?



```
public Coche(String marca, String modelo, String matricula,  
             int potencia, int velocidad, boolean arrancado) {  
    this.marca = marca;  
    this.modelo = modelo;  
    this.matricula = matricula;  
}
```

Constructores. Constructor con parámetros

Es **muy importante** respetar el orden de los parámetros que pasamos al constructor.

Tanto, que si intercambiamos los atributos y son de tipos de datos incompatibles, se genera un error indicando que no existe el constructor

```
Coche coche3=new Coche(false,"0000AAA","VW",0,95,"Polo");
```

```
public Coche(String marca,  
              int potencia,  
              this.marca = marca;  
              this.modelo = modelo;  
              this.matricula = matricula;
```



The constructor Coche(boolean, String, String, int, int, String) is undefined

2 quick fixes available:

- [Remove arguments to match 'Coche\(\)'](#)
- [Create constructor 'Coche\(boolean, String, String, int, int, String\)'](#)

Press 'F2' for focus

Constructores. Sobrecarga

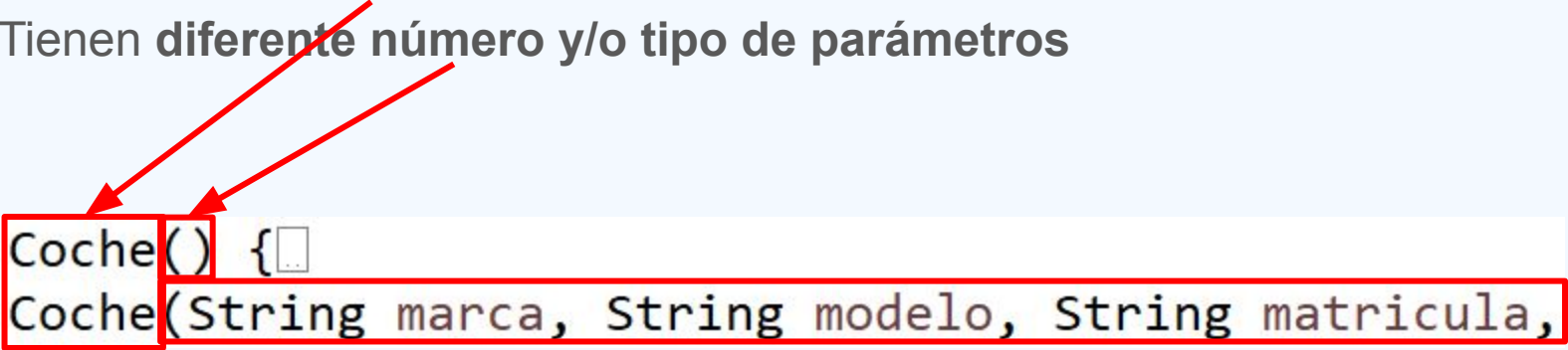
- De vuelta a la clase, vemos que ahora **hay dos constructores**

```
public Coche() {  
    marca="Sin marca";  
    modelo="Sin modelo";  
    matricula="0000AAA";  
    potencia=0;  
    velocidad=0;  
    arrancado=true;  
}  
  
public Coche(String marca, String modelo, String matricula,  
             int potencia, int velocidad, boolean arrancado) {  
    this.marca = marca;  
    this.modelo = modelo;  
    this.matricula = matricula;  
    this.potencia = potencia;  
    this.velocidad = velocidad;  
    this.arrancado = arrancado;  
}
```

- Los constructores están **sobrecargados**

Constructores. Sobrecarga

- La **sobrecarga** no es ningún problema, todo lo contrario
- Los métodos (y constructores) se sobrecargan **cuando**:
 - Tienen el **mismo nombre**
 - Tienen **diferente número y/o tipo de parámetros**



```
public Coche() {  
public Coche(String marca, String modelo, String matricula,
```

Constructores. Sobrecarga

- La **ventaja** de tener un constructor sobrecargado es que, cuando vayamos a crear un objeto podemos **decidir cómo hacerlo**
 - Con los **datos por defecto** utilizando el constructor sin parámetros
 - **Eligiendo los valores** de los atributos con el constructor con parámetros

```
Coche coche=new Coche("Seat","Ibiza","4321CBA",130,0,false);  
Coche coche2=new Coche();
```

Constructores. Sobrecarga

```
Coche coche=new Coche("Seat","Ibiza","4321CBA",130,0,false);  
Coche coche2=new Coche();
```



coche	Coche (id=28)
arrancado	false
marca	"Seat" (id=31)
matricula	"4321CBA" (id=37)
modelo	"Ibiza" (id=38)
potencia	130
velocidad	0
coche2	Coche (id=30)
arrancado	true
marca	"Sin marca" (id=39)
matricula	"0000AAA" (id=40)
modelo	"Sin modelo" (id=41)
potencia	0
velocidad	0

Constructores. Constructor con parámetros

IMPORTANTE: Si creamos un constructor con parámetros y **no creamos explícitamente** el constructor sin parámetros, éste no existe en la clase

```
Coche coche2=new Coche();|
```

The constructor Coche() is undefined

3 quick fixes available:

- + Add arguments to match 'Coche(String, String, String, int, int, boolean)'
- = Change constructor 'Coche(String, String, String, int, int, boolean)'
- 🔧 Create constructor 'Coche()'

```
//2.CONSTRUCTOR. SÓLO CON PARÁMETROS
```

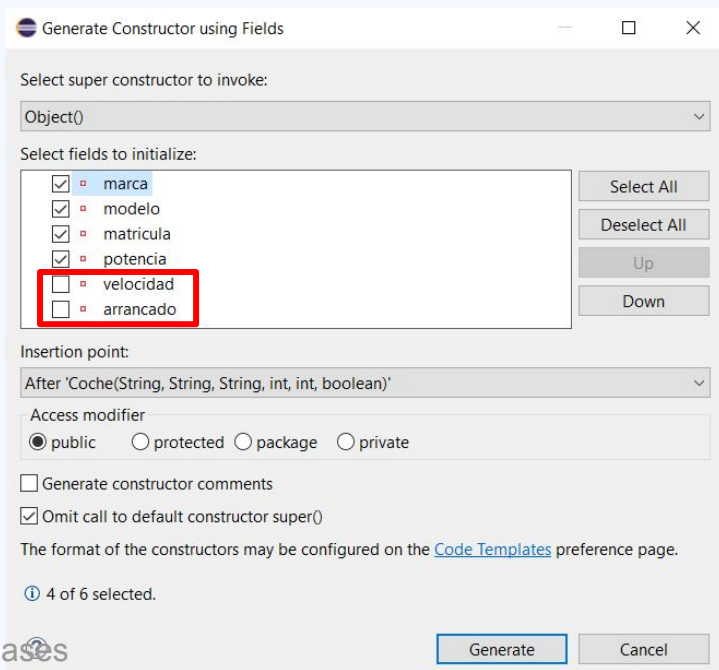
```
public Coche(String marca, String modelo, String matricula,  
              int potencia, int velocidad, boolean arrancado) {  
    this.marca = marca;  
    this.modelo = modelo;  
    this.matricula = matricula;  
    this.potencia = potencia;  
    this.velocidad = velocidad;  
    this.arrancado = arrancado;  
}
```

Constructores. Constructor con parámetros

- En el constructor con parámetros hemos decidido que **el usuario puede elegir** los valores que quiera para **todos los atributos** del coche
- Pero puede que algunos de esos **valores** deban ser **fijos**
- Por ejemplo, un coche siempre se crea sin arrancar y con velocidad 0

Constructores. Constructor con parámetros

- Por tanto **NO** queremos dar la posibilidad de que nos pasen una velocidad y un estado porque **siempre** va a ser 0 y false



```
public Coche(String marca, String modelo,
String matricula, int potencia) {
    this.marca = marca;
    this.modelo = modelo;
    this.matricula = matricula;
    this.potencia = potencia;
    //Ya sabemos que por defecto velocidad=0
    //y arrancado=false pero lo ponemos igual
    velocidad=0;
    arrancado=false;
}
```

Constructores. Constructor con parámetros

- Lo siguiente sería incorrecto: el usuario nos pasa una velocidad y un estado pero luego “pasamos de él” y asignamos 0 y false

```
public Coche(String marca, String modelo, String matricula,  
             int potencia, int velocidad, boolean arrancado)  
    this.marca = marca;  
    this.modelo = modelo;  
    this.matricula = matricula;  
    this.potencia = potencia;  
    this.velocidad = 0;  
    this.arrancado = false;  
}
```



Constructores

IMPORTANTE: No inicializar los atributos en la declaración de la clase. Utilizar siempre constructores para las inicializaciones

```
public class Coche {  
    private String marca="";  
    private String modelo="";  
    private String matricula="1111BBB";  
    private int potencia=0;  
    private int velocidad=0;  
    private boolean arrancado=false;  
}
```

¿Destructores?

- En Java **no existe un destructor** de objetos
- Sin embargo existe un mecanismo llamado **Garbage Collector** que se encarga de localizar objetos que ya no se usan y destruirlos para liberar memoria

Decidimos cuándo creamos un objeto pero no cuándo se destruye

Ejercicio

- Modifica la clase Alumno y crea un constructor con parámetros. Cuando se crea un alumno **siempre tendrá 0 faltas**
- Modifica “PrincipalMenu.java” y añade la opción “Crear alumno con datos” al menú que pedirá al usuario todos los datos necesarios para crear un Alumno y utilizará el constructor con parámetros para hacerlo

Métodos

- Los métodos definen el **comportamiento** de una clase

Un coche tiene atributos (marca, modelo, matrícula) pero también se pueden **hacer cosas** con un coche: arrancar, acelerar, frenar, estacionar...
Esas acciones que puede hacer un coche **son los métodos** en programación

- Su acción suele **afectar a los atributos**:
 - **Modificando** su/s valor/es
 - **Haciendo cálculos** en base a su/s valor/es

Métodos

- Por ejemplo ¿qué hace el **método arrancar** sobre un coche
 - en la vida real?
 - en un objeto coche?

▼ ⓘ coche	Coche (id=27)
△ arrancado	false
△ marca	null
△ matricula	null
△ modelo	null
△ potencia	0
△ velocidad	0

Métodos

- Por ejemplo ¿qué hace el **método arrancar** sobre un coche
 - en la vida real? Arranca el motor del coche
 - en un objeto coche? poner "arrancado" a true

▼ ⓘ coche	Coche (id=27)
△ arrancado	false
△ marca	null
△ matricula	null
△ modelo	null
△ potencia	0
△ velocidad	0

Métodos

- Los métodos tienen **dos partes**:
 - **Cabecera**. Es la primera línea
 - **Cuerpo**. Va entre llaves y en él ponemos las acciones del método

```
public void nombreMetodo() {  
    //Cuerpo del método  
}
```

Métodos

- Vamos a **implementar el método arrancar**
- Hemos visto que lo único que hace es `arrancado=true;`

```
public void arrancar() {  
    arrancado=true;  
}
```

- **Público** para que se pueda usar por otras clases
- **No devuelve** datos. Sólo va a cambiar el valor de `arrancado`
- Su nombre es **arrancar**. El nombre debe ser descriptivo y con notación camelCase
- **No recibe parámetros**. Para hacer `arrancado=true` no necesita ninguna información
- En el **cuerpo** ponemos el código del método

Métodos

- El uso de los métodos se hace a través del operador punto

```
Coche coche=new Coche("Seat","Ibiza","4321CBA",130,0,false);  
coche.arrancar();
```

coche	Coche (id=28)
arrancado	true
> marca	"Seat" (id=30)
> matricula	"4321CBA" (id=36)
> modelo	"Ibiza" (id=37)
potencia	130
velocidad	0

Métodos

- Implementa el método `parar()` en la clase `Coche`. Detendrá el coche

Al existir los métodos `arrancar()` y `parar()`, ya no tiene sentido el método `setArrancado()` en la clase `Coche`. **Debemos eliminarlo**

Ejercicio

- Añade el método `faltar()` a la clase `Alumno`. Cuando un alumno falta se le incrementan en 1 sus faltas de asistencia
- Modifica el “PrincipalMenu.java” y añade la opción “Registrar falta” al menú que utilizará el método anterior

Observa que ahora **sobra el método `setFaltas()`**

Métodos. Parámetros

- Algunos métodos necesitan **información adicional** para hacer su tarea
- Piensa en el método `acelerar`
 - Si le pedimos a un coche que acelere, el coche **no sabe** cuánto tiene que acelerar
 - Por tanto necesitará **un parámetro**: la `aceleración`

Métodos. Parámetros

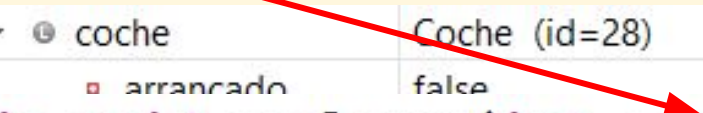
```
public void acelerar(int aceleracion) {  
    //Se incrementa la velocidad del coche  
    velocidad+=aceleracion;  
}
```

- El método va a recibir un **entero** con la **aceleración**
- En el cuerpo del método se incrementa la velocidad según la aceleración

Métodos. Parámetros

- El uso de los métodos se hace a través del operador punto

```
Coche coche=new Coche("Seat","Ibiza","4321CBA",130,0,false);  
coche.acelerar(50); //Aceleramos 50km/h
```




coche	Coche (id=28)
arrancado	false
<pre>public void acelerar(int aceleracion) { velocidad+=aceleracion; }</pre>	
potencia	130
velocidad	0

Métodos. Parámetros

IMPORTANTE: Un método no debe pedir datos por teclado al usuario.

Esa información se **debe introducir** a través de los **parámetros**

```
public void acelerar() {  
    System.out.println("Introduce la aceleración");  
     int aceleracion=sc.nextInt();  
    velocidad+=aceleracion;  
}
```

Métodos. Parámetros

IMPORTANTE: Un método no debe pedir datos por teclado al usuario.

Esa información se **debe introducir** a través de los **parámetros**



```
public void acelerar(int aceleracion) {  
    //Se incrementa la velocidad del coche  
    velocidad+=aceleracion;  
}
```

Métodos. Parámetros

Los métodos **NO HABLAN**

Métodos. Sobrecarga

- Hemos visto que el método `acelerar` necesita **información adicional** (`aceleracion`) para incrementar la velocidad del objeto coche
- Pero también podemos decidir una **aceleración por defecto** en caso de que no se indique ninguna
- En este caso `acelerar` no recibe parámetros y al haber otro método `acelerar` que sí los recibe, lo estamos **sobrecargando**

```
public void acelerar(int aceleracion) {  
    velocidad+=aceleracion;  
}  
  
public void acelerar() {  
    //Si no se indica la aceleración  
    //incrementamos la velocidad en 10  
    velocidad+=10;  
}
```

Métodos. Sobrecarga

- El uso de uno u otro método **depende de los parámetros** que se pasen

```
Coche coche=new Coche("Seat","Ibiza","4321CBA",130,0,false);  
coche.acelerar(50); //Aceleramos 50km/h  
coche.acelerar();   //Aceleramos 10km/h
```

▼ coche	Coche (id=28)
▣ arrancado	false
> ▣ marca	"Seat" (id=30)
> ▣ matricula	"4321CBA" (id=36)
> ▣ modelo	"Ibiza" (id=37)
▣ potencia	130
▣ velocidad	60

Métodos

- Implementa el método `frenar()` en la clase `Coche`. Disminuirá la velocidad del coche
- Ten en cuenta que la **velocidad mínima** de un coche es 0

Al existir los métodos `acelerar()` y `frenar()`, ya no tiene sentido el método `setVelocidad()` en la clase `Coche`.

La velocidad se cambia acelerando o frenando y no fijándola directamente con un método `set`

Ejercicio

- Sobrecarga el método `faltar()` de la clase `Alumno` para permitir registrar varias faltas de asistencia
- Modifica “PrincipalMenu.java” y añade la opción “Registrar faltas” al menú que utilizará el método anterior y pedirá al usuario el número de faltas que quiere registrar
- AMPLIACIÓN: Haz que sólo haya una entrada en el menú de opciones para registrar faltas. Dentro de esa opción, pregunta al usuario si quiere registrar una falta o más de una falta y dependiendo de cuántas faltas quiera registrar, se llama a un método o a otro

Métodos. Return

- Piensa en el siguiente problema: queremos hacer un método que muestre la matrícula de un coche y su velocidad actual
- Una posible solución sería:

```
public void matriculaYVelocidad() {  
    System.out.println("Soy un coche con matrícula "  
                        +matricula+" y voy a "+velocidad+" km/h");  
}
```

Métodos. Return

- Su uso sería:

```
Coche coche=new Coche("Seat","Ibiza","4321CBA",130,60,false);  
coche.matriculaYVelocidad();
```

- Y muestra por pantalla:


```
Soy un coche con matrícula 4321CBA y voy a 60 km/h
```

Métodos. Return

IMPORTANTE: Un método no debe mostrar datos por pantalla al usuario.

Esa información la **debe devolver** a través del **return**


```
public void matriculaYVelocidad() {  
    System.out.println("Soy un coche con matrícula "  
                        +matricula+" y voy a "+velocidad+" km/h");  
}
```



Métodos. Return

IMPORTANTE: Un método no debe mostrar datos por pantalla al usuario.

Esa información la **debe devolver** a través del **return**



```
public String matriculaYVelocidad() {  
    return "Soy un coche con matrícula "  
           +matricula+" y voy a "+velocidad+" km/h";  
}
```

Métodos. Return

Los métodos **NO HABLAN**

Métodos. Return

- Ahora su **uso cambia** en el Main

```
System.out.println(coche.matriculaYVelocidad());
```

- Como el método ya **no muestra** directamente la información de matrícula y velocidad, el **main debe hacerlo**
 - a través de un `println` en este caso
 - o a través de un html
 - o en un mensaje emergente en Android...

Métodos. Return

- Volvamos al método `acelerar` para complicarlo un poco
- Lo que tenemos hasta ahora es:

```
public void acelerar(int aceleracion) {  
    velocidad+=aceleracion;  
}
```

- Pero si lo pensamos mejor, un coche solo puede acelerar si está arrancado
- Modifica el código

Métodos. Return

```
public void acelerar(int aceleracion) {  
    if(arrancado) { //Evitar arrancado==true  
        velocidad+=aceleracion;  
    }  
}
```

- Por tanto ahora puede ser que el usuario (a través del Main) quiera acelerar un coche y **no sea posible** porque está parado
- En ese caso, el método debe indicar **si pudo o no** hacer la acción que se le pide
- Lo hará **devolviendo true/false** con el return
- Modifica el método

Métodos. Return

```
public boolean acelerar(int aceleracion) {  
    if(arrancado) {  
        velocidad+=aceleracion;  
        return true;  
    }else {  
        return false;  
    }  
}
```

- Ahora el método hace **dos cosas**: acelerar e indicar si se pudo o no acelerar

IMPORTANTE: Cuando se ejecuta una cláusula **return** se **finaliza** la ejecución del método. El resto de instrucciones no se ejecutan

Métodos. Return

- De nuevo su **uso cambia** en el Main: debemos mostrar al usuario un mensaje indicando si se pudo o no acelerar

```
if (coche.acelerar(50)) { //Evitar ==true
    //Si se pudo acelerar...
    System.out.println("Se ha acelerado el coche");
}else {
    System.out.println("No se pudo acelerar. Arranca primero el coche");
}
```

Métodos. Return

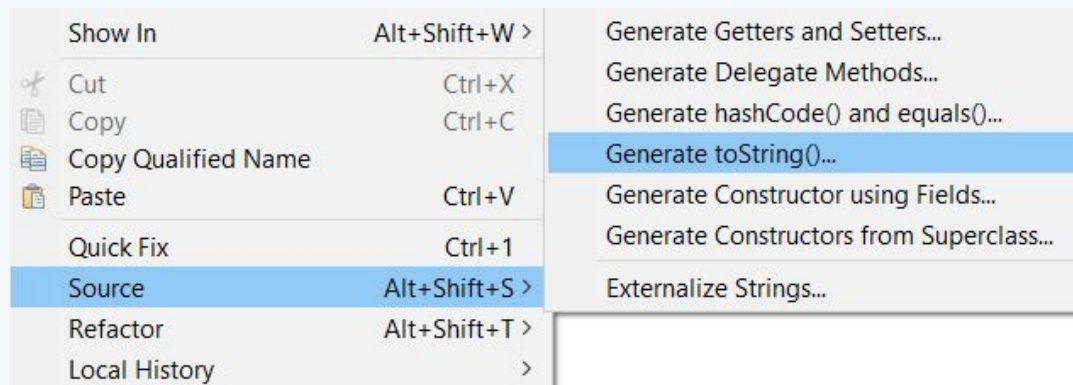
- Los **tipos de datos** que se suelen devolver en un return son:
 - `int/double` cuando hay que devolver el resultado de un cálculo
 - `String` cuando hay que devolver varios datos y esos datos se usarán para mostrarlos por pantalla
 - `boolean` cuando hay que indicar si se llevó a cabo o no una operación
 - `Objeto` cuando hay que devolver un tipo de datos complejo

Ejercicio

- Añade el método `calcularNotaMedia()` a la clase `Alumno`. Deberá calcular y devolver la nota media de las tres asignaturas del alumno
- Modifica “PrincipalMenu.java” y añade la opción “Mostrar nota media” al menú que utilizará el método anterior y mostrará la media por pantalla

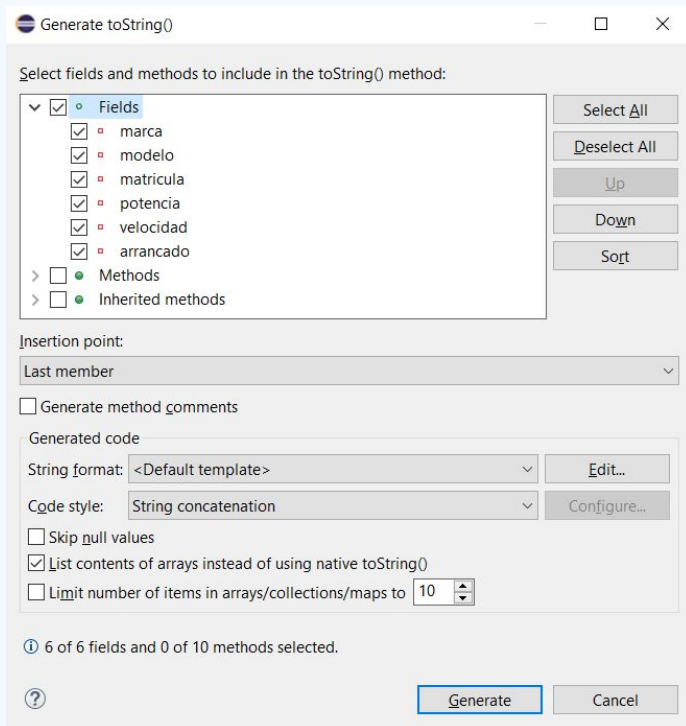
Métodos. toString

- Hasta ahora para mostrar la información que almacena un objeto utilizábamos los **getters**
- Pero podemos hacer un **método** que **devuelva toda esa información** en un String
- Esa operación está automatizada por Eclipse



Métodos. toString

- Seleccionamos los campos que queremos mostrar:



Métodos. toString

- Y se genera el código:

```
@Override
public String toString() {
    return "Coche [marca=" + marca + ", modelo=" + modelo
        + ", velocidad=" + velocidad + ", arrancado="
}
```

Métodos. toString

- Su uso es simple:

```
Coche coche=new Coche("Seat","Ibiza","4321CBA",130,0,false);  
System.out.println(coche.toString());
```

```
Coche [marca=Seat, modelo=Ibiza, matricula=4321CBA, potencia=130, velocidad=0, arrancado=false]
```

IMPORTANTE: Si se ejecutase el código `System.out.println(coche);`

También se estaría llamando al método toString de Coche

Métodos. toString

- Es recomendable **modificar** el código que se genera en el toString para mostrar la información al usuario de forma más agradable

```
public String toString() {  
    return "Objeto coche \n"  
        + "Marca: " + marca + "\n"  
        + "Modelo:" + modelo + "\n"  
        + "Matrícula: " + matricula + "\n"  
        + "Potencia: " + potencia + " CV\n"  
        + "Velocidad: " + velocidad + " Km/h\n"  
        + "Arrancado: " + arrancado + "\n";  
}
```

```
Objeto coche  
Marca: Seat  
Modelo:Ibiza  
Matrícula: 4321CBA  
Potencia: 130 CV  
Velocidad: 0 Km/h  
Arrancado: false
```

Ejercicio

- Añade el método `toString()` a la clase `Alumno` para mostrar sus datos
- Modifica en “PrincipalMenu.java” la entrada del menú “Mostrar datos de alumno” para que utilice este método

Métodos. equals

- Para **comparar** si dos objetos son iguales no podemos utilizar el operador ==

```
coche1==coche2; //Devuelve siempre false
```

- Devuelve siempre false porque **no son el mismo objeto** aunque los datos que contienen sean iguales
- Son distintos objetos porque tienen diferente **id**

▼ coche	Coche (id=22)
▲ arrancado	false
▲ marca	null
▲ matricula	null
▲ modelo	null
▲ potencia	0
▲ velocidad	0
▼ coche2	Coche (id=25)
▲ arrancado	false
▲ marca	null
▲ matricula	null
▲ modelo	null
▲ potencia	0
▲ velocidad	0

Métodos. equals

- La solución pasa por implementar el método **equals** en la clase Coche
- Con él comparamos los atributos que nos interesen de dos objetos

Métodos. equals

- Debemos decidir cuál es la **condición de igualdad**
 - Por ejemplo: dos coches son iguales si tienen la **misma matrícula**
- Cuestiones **importantes**:
 - **Devuelve** un boolean indicando si dos objetos son o no iguales
 - **Recibe** un objeto (de tipo `Object`) por parámetro. En nuestro caso recibirá otro coche, un segundo coche

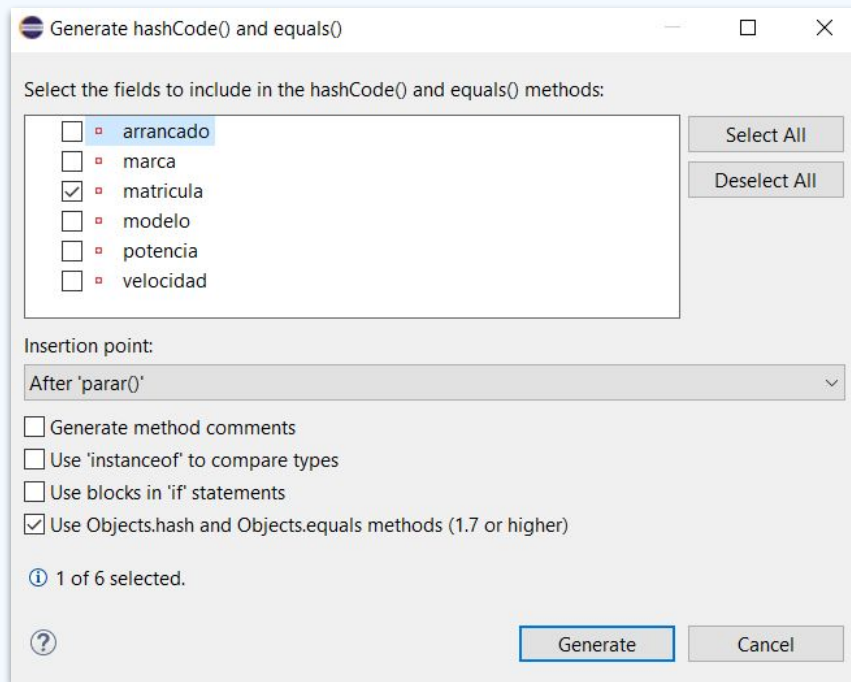
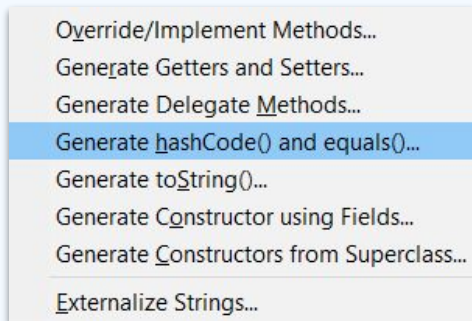
```
public boolean equals(Object obj) {  
  
}
```

Métodos. equals

```
public boolean equals(Object obj) {  
    //Si la matrícula del coche actual, es igual  
    //a la matrícula del otro coche, son iguales  
    if(this.matricula.equals(((Coche) obj).getMatricula())) {  
        return true;  
    }else {  
        return false;  
    }  
}
```

Métodos. equals

- Aunque también **Eclipse automatiza** la generación del método equals (y hashCode)
- Se selecciona el campo o campos que queremos que cumplan el criterio de igualdad



Métodos. equals

- Uso en el Main:

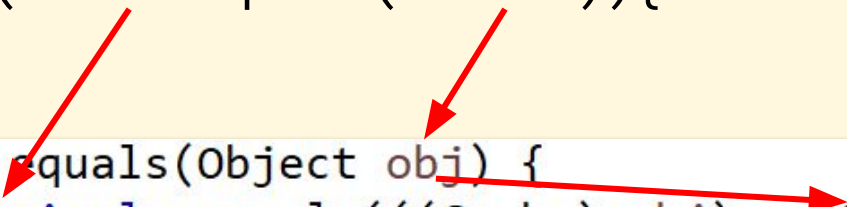
```
if(coche.equals(coche2)) {  
    System.out.println("Son iguales");  
}else {  
    System.out.println("Son distintos");  
}
```

Métodos. equals

- Uso en el Main:

```
if(coche.equals(coche2)){
```

```
public boolean equals(Object obj) {  
    if(this.matricula.equals(((Coche) obj).getMatricula())) {  
        return true;  
    }else {  
        return false;  
    }  
}
```



Ejercicio

- Añade el método `equals()` a la clase `Alumno`. Dos alumnos son iguales si tienen el mismo nombre y apellido
- En “`PrincipalMenu.java`” añade una entrada al menú que permita crear un segundo objeto de tipo `Alumno` y lo compare con otro alumno que ya exista. Deberá mostrar si los alumnos son iguales

Métodos. hashCode

- Cuando se genera el método equals es recomendable también implementar el método **hashCode**
- Lo que hace este método es **generar un ID único** para cada objeto en base al valor de alguno de sus atributos
- Más adelante veremos su utilidad

```
//Código autogenerado por Eclipse  
public int hashCode() {  
    return Objects.hash(matricula);  
}
```

Hashcode de dos coches con diferente matrícula:
2018178271
1070551353

FIN!