

JAVA II

2. PROGRAMACIÓN ORIENTADA A OBJETOS (PARTE II)

Continúa con las características de la programación orientada a objetos. Aprende para qué sirve el encapsulamiento, cuándo una clase tiene un método abstracto y cómo lo hereda a la propia clase.

1. Programación Orientada a Objetos (Parte I)

- 1.1. Características de la POO
- 1.2. Herencia
- 1.3. Sobrecarga y sobrescritura del método
- 1.4. Polimorfismo
- 1.5. Preguntas Frecuentes



2. Programación Orientada a Objetos (Parte II)

- 2.1. Encapsulamiento
- 2.2. Abstracción: Clases y métodos abstractos
- 2.3. Abstracción: Interfaz vs clase abstracta
- 2.4. Preguntas Frecuentes

3. Otros conceptos relacionados con la POO

- 3.1. Composición
- 3.2. Genéricos
- 3.3. Colecciones
- 3.4. Excepciones
- 3.5. Preguntas Frecuentes

2. PROGRAMACIÓN ORIENTADA A OBJETOS (PARTE II)


2.1. ENCAPSULAMIENTO

¿EN QUÉ CONSISTE EL ENCAPSULAMIENTO?

El encapsulamiento también llamado en Java **ocultación de datos**, nos permite limitar el acceso a los **atributos** de nuestras clases de tal forma que podamos tener un mayor control sobre ellas.

Es recomendable poner privados los atributos de nuestra clase, para que otras personas desde otras clases, no puedan acceder a nuestros datos o modificarlos como ellos quieran.

```
public class Coche {  
  
    //declaración de atributos  
    private String color;  
    private double longitud;  
    private int año;  
  
    //declaración de métodos  
    public void arrancar (){}  
    public void acelerar (){}  
    public void frenar (){}  
}
```



A diagram consisting of a vertical line and a horizontal line forming an L-shape. The horizontal line points to the text 'Atributos encapsulados'.

Atributos encapsulados

Solo es posible acceder a estos atributos desde los **métodos** definidos en la propia clase.

No olvides que... Los métodos públicos **Getters** y **Setters** son la excepción. Con ellos sí podemos acceder a los atributos privados de la clase y obtener (Get) o modificar (Set) la información (valor) de éstos.

Atributos: definen las características que tendrán los objetos que crearemos a partir de la clase.

Método: es un conjunto de instrucciones definidas dentro de una clase, que realizan una determinada tarea. A la hora de escribirlo hay que tener en cuenta el número de parámetros, el tipo de parámetros y el orden de estos.

Getter: método público que nos permite obtener información de atributos privados de una clase.

Setter: método público que nos permite modificar o establecer valores a los atributos privados de una clase.

2. PROGRAMACIÓN ORIENTADA A OBJETOS (PARTE II)

2.2. ABSTRACCIÓN: CLASES Y MÉTODOS ABSTRACTOS

¿EN QUÉ CONSISTE LA ABSTRACCIÓN?

La abstracción consiste en crear o definir algo de **forma abstracta o generalizada**, es decir, el modificador **abstract** indica que NO se provee de una implementación.

Imaginemos que, durante el diseño del modelo de clase, si algo es demasiado generalizado se puede dejar o marcar como abstracto y posteriormente las clases que extiendan de esta deben implementar dichos métodos, de tal forma que debemos extender clases abstractas y sobrescribir los métodos abstractos.

Se puede utilizar con métodos y clases.

- **Método:** es un método incompleto, sin implementación, y se termina con un punto y coma “public abstract void conducir();”. Su implementación vendrá dada por las clases que extiendan de la clase donde se ha declarado.
- **Clase:** es una clase incompleta, con lo que no se puede instanciar, es decir, no se puede crear una instancia u objeto de ella.

No olvides que... Una clase que tenga uno o más métodos abstract debe declararse como abstract a su vez.

REGLAS DE UN MÉTODO ABSTRACTO

- No debe tener implementación o cuerpo, es decir, sólo tiene la cabecera y termina con punto y coma (;).
- Debe declararse dentro de una clase abstracta.
- Debe ser sobrescrito por las subclases.

REGLAS DE UNA CLASE ABSTRACTA

- Puede tener uno, múltiples o ningún método abstracto.
- No podrá ser instanciada.
- Debería ser subclasificada para ser utilizada.

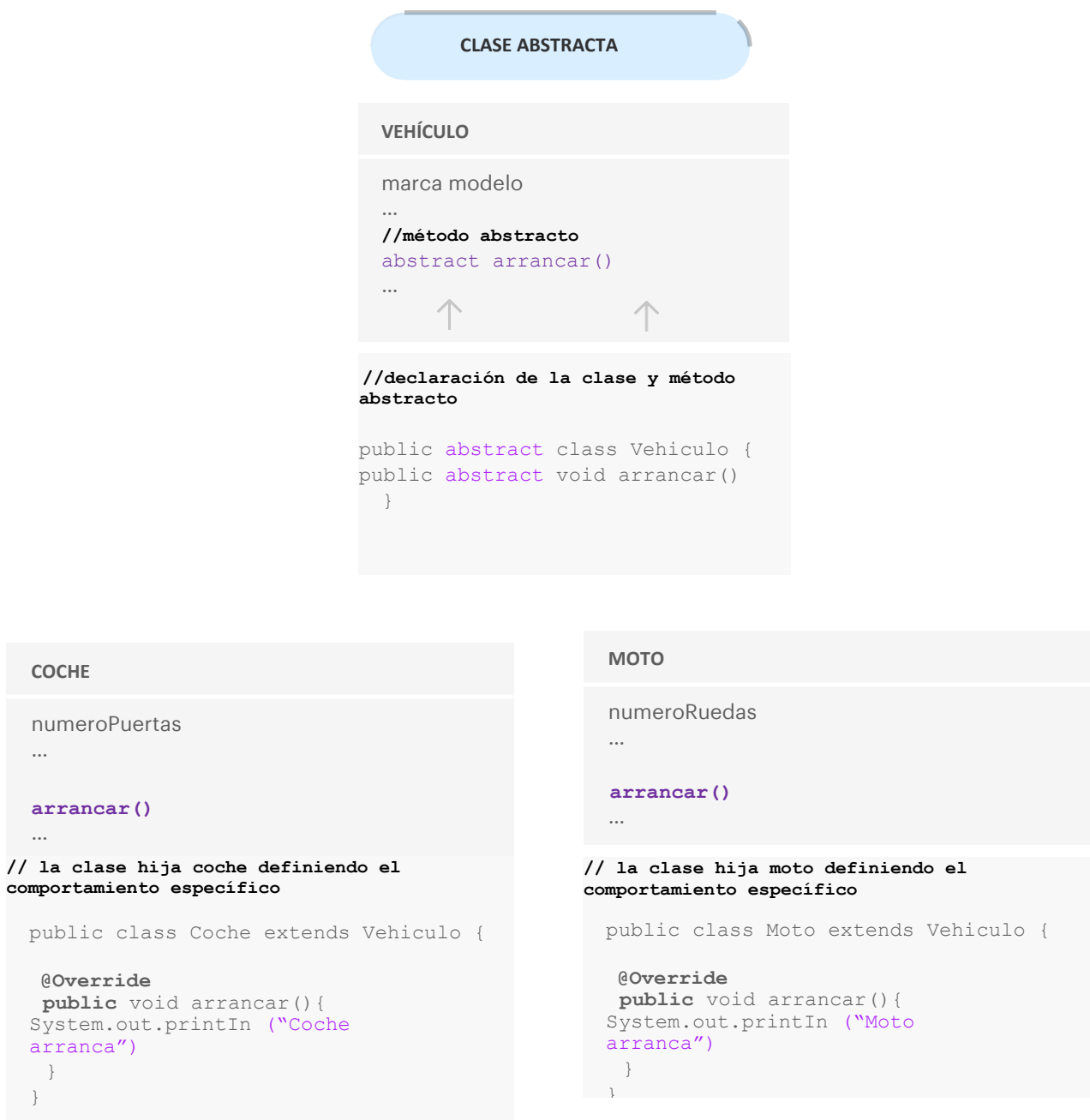
Ten en cuenta que... Cualquier subclase que se extienda desde una clase abstracta debe implementar todos los métodos abstractos de la superclase o la subclase misma debe marcarse como abstracta.

Ejemplo: Tenemos una clase FiguraGeometrica y toda figura geométrica debe tener un método para calcular su área, pero en la clase FiguraGeometrica no podemos codificar ese método, ya que ese cálculo va a ser diferente dependiendo de cada tipo de figura geométrica. Por lo tanto, definiremos el método area() abstracto en la clase FiguraGeometrica y dejaremos que cada una de las subclases (Triangulo, Circulo, Cuadrado, etc.) especifiquen el contenido de ese método sobrescribiéndolo. Al crear este método area() en la superclase, estaremos “obligando” a que todas las subclases respeten el formato especificado y tengan dicho método con la implementación que necesiten.

UN EJEMPLO DE ABSTRACCIÓN

Imaginemos que en la clase *Vehiculo* tenemos un método `arrancar()` que queremos detallar más adelante en las clases hijas (moto y coche). Esto podemos hacerlo creando la clase *Vehiculo* como una **Clase Abstracta**, es decir, como una **clase base** donde declararemos el método `arrancar()` como un **Método Abstracto, sin especificar su comportamiento o las tareas a realizar**, obligando de esta manera a que en las clases hijas (Coche y Moto) se definan estos métodos de manera específica.

Si una clase tiene un método abstracto, la clase será siempre abstracta, y la declararemos con la palabra ***abstract***. Veamos en el ilustrativo cómo sería.



2. PROGRAMACIÓN ORIENTADA A OBJETOS (PARTE II)

2.3. ¿QUÉ ES UNA INTERFAZ Y CÓMO SE IMPLEMENTA?

Como ya adelantamos en los tipos de herencia, en Java no es posible la herencia múltiple, es decir, no es posible heredar de dos clases. Pero también decíamos que sí podríamos hacerlo usando interfaces.

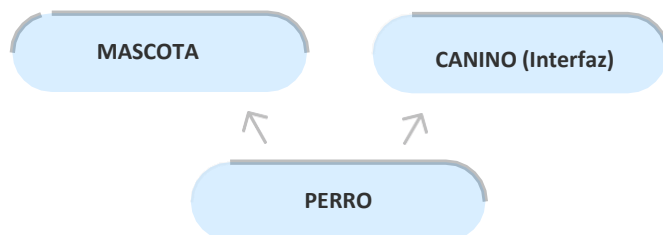
¿QUÉ ES UNA INTERFAZ?

Es una **colección de métodos abstractos y constantes** en los que se especifica **qué** se debe hacer pero no el **cómo**. Serán las clases hijas quienes definan el comportamiento al implementarlas.

Podemos ver a una interface como una especie de **plantilla para la construcción de clases**, ya que normalmente una interface se compone de un conjunto de declaraciones de cabeceras de métodos (sin implementar, de forma similar a un método abstracto) que especifican un protocolo de comportamiento para una o varias clases y, además, una interface puede emplearse también para declarar constantes que luego puedan ser utilizadas por otras clases.

A diferencia de una clase abstracta, una interfaz no puede hacer nada por si sola.

Veamos un ejemplo en el que tenemos la clase *Perro* y necesitamos que herede de dos superclases (canino y mascota). Para ello, crearemos la interfaz Canino y heredaremos sus métodos y atributos.



Creemos la interfaz Canino que incluye por ejemplo el método abstracto ladrar. A la hora de declararla, ten en cuenta que en inglés se escribe "interface".

```
public class interface Canino{  
    public abstract void ladrar();  
}
```

Para poder aplicar la interfaz en nuestra clase *Perro* basta con utilizar la palabra reservada ***implements***, seguido del nombre de la interfaz que queremos usar; y a continuación, sobrescribimos el método ladrar() con la palabra ***@Override***. Veamos como heredamos de la clase Mascota e implementamos a la vez la interfaz Canino:

```
public class Perro extends Mascota implements Canino {  
    @Override  
    public void ladrar(){  
    }  
}
```

Ten en cuenta que una clase hija puede heredar de una única clase o clase abstracta, pero

podrá usar todas las interfaces que necesite para acceder a otras clases.

REGLAS DE LAS INTERFACES

- Las **interfaces se declaran** utilizando la palabra reservada **interface**.
- Para **declarar una clase** que implemente una interface se utiliza la palabra reservada **implements** en la cabecera de declaración de la clase.
- Una **clase que implementa un interfaz** está obligada a **implementar todos los métodos que hereda** de la misma (si no lo hace o se deja alguno sin implementar, ese método tendrá que ser declarado como abstracto).
- Una interface no puede declarar ningún constructor.
- Las **interfaces** solo admiten modificadores de acceso **public** y **protected**.
- Todas las **constantes** incluidas en una interfaz se declaran implícitamente como **public final static** y es necesario inicializarlas en la misma sentencia de declaración.
- Los **métodos** declarados en un interfaz son siempre **public abstract**, de modo implícito, aunque no se especifique.
- Una clase puede implementar varias interfaces.
- Una interface puede extender de varias interfaces.
- Una interface no se puede instanciar (es abstracto).
- Una interface puede ser utilizada como tipo de dato para una variable de referencia, siempre y cuando la clase del objeto con la que se cree implemente la interface.

Ejemplo:

```
public interface Comportamientos {
    public final static int CONSTANTE = 30;

    public abstract void comer();
    public abstract void saltar();
}

public class Persona implements Comportamientos{
    @Override //Método de la interfaz Comportamientos
    public void comer() {
        System.out.println("Persona come");
    }
    @Override //Método de la interfaz Comportamientos
    public void saltar() {
        System.out.println("Persona salta" + Comportamientos.CONSTANTE + " veces.");
        /*Usamos la constante definida en Comportamientos*/
    }
    public void dormir() {
        System.out.println("Persona duerme");
    }
}

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

        p.saltar();
        p.comer();
        p.dormir();

        Comportamientos c = new Persona();
        c.saltar();
        c.comer();
    }
}
```

INTERFACES VS. CLASES ABSTRACTAS

La diferencia entre una clase abstracta y una interfaz es que **las clases no heredan de las interfaces sino que las implementan**, con lo que no hay una relación jerárquica entre ellas.

CLASES ABSTRACTAS

- Puede tener métodos abstractos y no abstractos.
- No admite herencias múltiples.
- Puede implementar varias interfaces.

INTERFACES

- Solo puede tener métodos abstractos.
- Es compatible con herencias múltiples.
- No puede hacer nada por si sola.

2. PROGRAMACIÓN ORIENTADA A OBJETOS (PARTE II)

2.4. PREGUNTAS FRECUENTES

- **¿En Java, una clase hija puede heredar de dos clases padres?**

Esto no es posible, pero podemos hacer uso de Interfaces y heredar sus métodos de forma ilimitada.

- **¿Una clase puede usar más de una interfaz?**

Si, puede usar todas las interfaces que quiera.

- **¿Cómo puedo acceder y obtener al valor de un atributo privado?**

Usando el método público getter

- **¿Cómo puedo acceder y modificar el valor de un atributo privado?**

Usando el método público setter.

**FUNDACIÓN
ACCENTURE**

**>
accenture**