FUNDACIÓN ACCENTURE

accenture

JAVA II

1. PROGRAMACIÓN ORIENTADA A OBJETOS (PARTE I)

Conoce las principales características de la Programación Orientada a Objetos, y adéntrate en 2 de ellas: la herencia y el polimorfismo.

TABLA DE CONTENIDOS

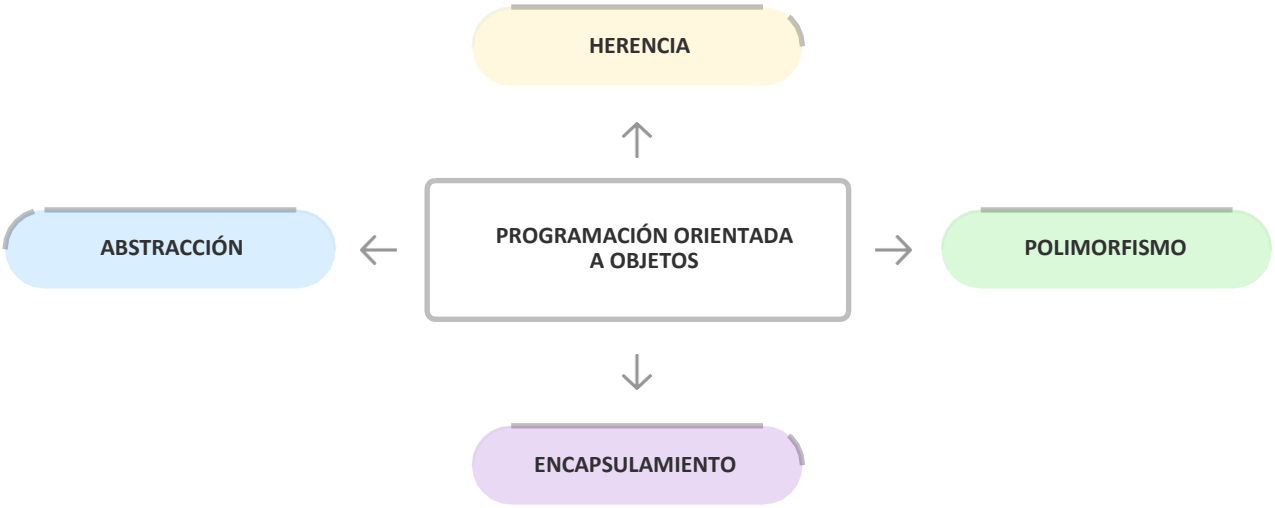
JAVA I

1.1. CARACTERÍSTICAS DE LA POO

PROGRAMACIÓN ORIENTADA A OBJETOS I

En el primer curso de Java vimos que se trata de un lenguaje de programación orientado a objetos (POO). Se llama así porque intenta **dar al código de programación el mismo comportamiento y características que tienen los objetos del mundo real**. Todo esto con el objetivo de que otro programador cuando quiere corregir un fallo o añadir nuevas funcionalidades, le resulte más sencillo.

CARACTERÍSTICAS DE LA PROGRAMACIÓN ORIENTADA A OBJETOS



- **Herencia**: las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación (imagina la forma de un árbol jerárquico) y los objetos heredan las propiedades y el comportamiento de las clases a las que pertenecen. La herencia nos permite extender una clase existente para crear una clase más especializada.

- **Polimorfismo**: permite enviar el mismo mensaje a objetos de diferentes clases, de forma que cada uno de ellos responde a ese mismo mensaje de modo distinto dependiendo de su implementación.

- **Encapsulamiento**: hace referencia a limitar el acceso a los atributos de nuestras clases o estado de nuestros objetos haciéndolos privados, de tal forma que podamos tener un mayor control sobre ellos.

- **Abstracción**: proceso de interpretación y diseño que implica reconocer y enfocarse en las características importantes del objeto, filtrando o ignorando todas las particularidades no esenciales. Esto permite definir las características diferenciales de los objetos, aquellas que lo distinguen de los demás tipos de objetos.

En este primer tema explicaremos en más detalle Herencia y Polimorfismo y en el siguiente veremos Encapsulamiento y Abstracción.

1.2. HERENCIA

¿EN QUÉ CONSISTE LA HERENCIA?

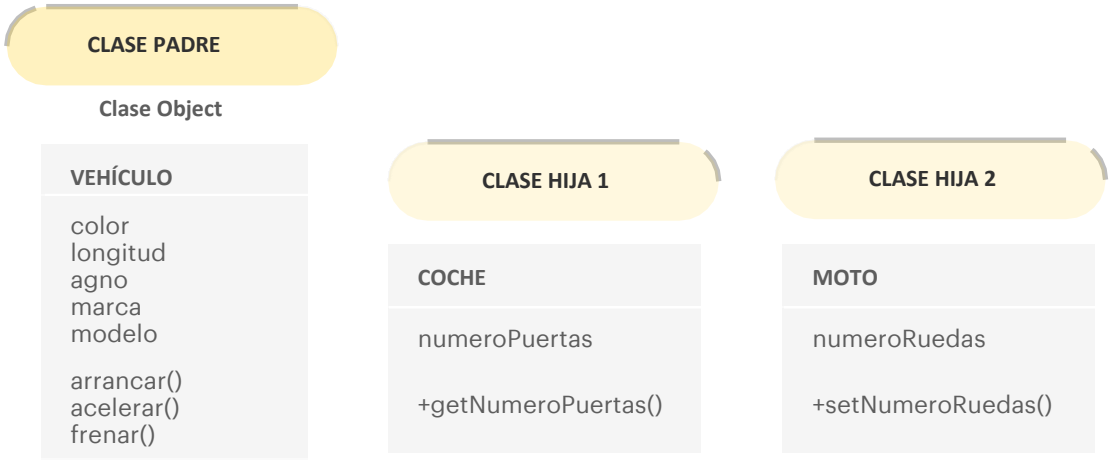
La **herencia** nos permite extender una clase existente para crear una subclase más especializada. La relación de herencia se describe como una **relación 'is-a'** (es-un) entre dos clases, por ejemplo un coche es un vehículo. En estas relaciones identificamos dos **tipos de clases**:

→ **Clase Padre** (Base/Superclase): una clase de la que se deriva otra clase. Es la clase a partir de la cual se van a crear especializaciones (en nuestro ejemplo sería la clase Vehiculo).

→ **Clase Hija** (Derivada/Subclase): una clase que se deriva de otra clase. Es la especialización (en nuestro ejemplo sería la clase Coche).

Además, cualquier jerarquía que exista tendrá en la cúspide la **clase implícita Object de la API de Java** de la que heredan todos los tipos de clases. Esto significa que la variable de referencia *Object* puede usarse para referirse a un objeto de cualquier clase.

Siguiendo con nuestro **ejemplo** del coche, nuestra clase *Coche* será una clase hija que hereda de la clase padre *Vehiculo*, que a su vez tendrá otras clases hijas, por ejemplo *Moto*.



En nuestro ejemplo, ambas clases heredarán los atributos y métodos de la clase Vehiculo (permitiendo así reutilizar el código ya existente en la clase padre), y adicionalmente podrán crear más atributos y métodos propios que los hagan más especializados.

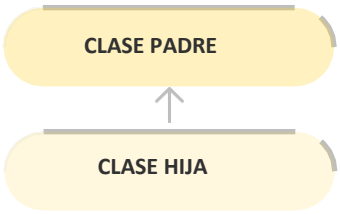
Por ejemplo sabemos que todos los coches tienen 4 ruedas, pero Motos las hay de 2 y 3 ruedas, por ello uno de sus atributos será éste para poder diferenciar los diferentes objetos que creemos a partir de la clase Moto. Y en los Coches ocurre igual con las puertas.

Ten en cuenta que... Las clases sólo pueden heredar de una única clase padre, aunque admite alguna excepción (Herencia Múltiple) que veremos en el siguiente punto.

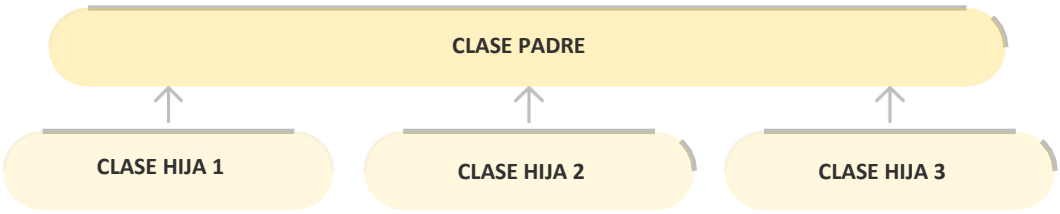
Clase Object: es la superclase implícita de la API de Java que se encuentra en lo más alto de la jerarquía de clases (tanto las que existen en la API de Java como las que creamos nosotros). De ella heredan el resto de las clases.

TIPOS DE HERENCIA

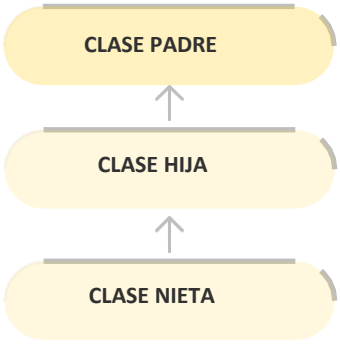
→ **Herencia única**: una clase derivada se crea a partir de una sola clase base.



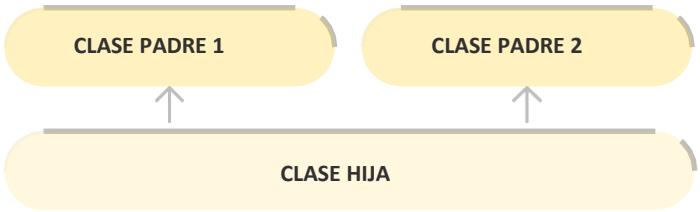
→ **Herencia jerárquica**: una clase es heredada por muchas subclases.



→ **Herencia multinivel**: una clase derivada se crea a partir de una única clase base, y esta clase derivada se convierte en la clase base para la nueva clase.



→ **Herencia múltiple**: una clase derivada se crea a partir de más de una clase base.



Ten en cuenta que... Java no admite este tipo de Herencia Múltiple con clases, es decir, no se puede extender o heredar de dos clases a la vez, pero lo podríamos lograr haciendo uso de Interfaces (lo veremos más adelante).

PROPIEDADES DE LA HERENCIA

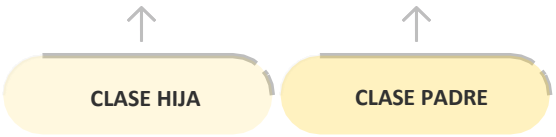
- La nueva clase puede incluir métodos y **atributos** propios para completar su función.
- Una clase puede ser a la vez subclase y superclase (ejemplo Herencia multinivel).
- Una subclase puede redefinir **métodos** heredados si no son final.
- No se heredan los miembros privados.
- No se pueden heredar **constructores**.

LA IMPLEMENTACIÓN DE LA HERENCIA EN JAVA

La implementación de la herencia se realiza utilizando la palabra reservada ***extends*** seguida del nombre de la clase padre.

Es decir, si al crear nuestra clase Coche queremos que herede los atributos y métodos de la clase padre Vehiculo tendremos que declarar el nombre de la clase hija y añadir la palabra *extends* seguida del nombre de la clase padre de la que hereda:

```
[modificador] class NomClase extends NomSuperClase {
```



Siguiendo nuestro **ejemplo** sería:

```
public class Coche extends Vehiculo
```

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.

Constructor: son un tipo de métodos especiales que inician los atributos de la clase cada vez que se crea un objeto

con valores coherentes. Sirven para establecer el estado inicial de un objeto.

PALABRA RESERVADA ***SUPER***

Habíamos visto que una de las propiedades de la herencia es que no se pueden heredar constructores de una clase padre a una clase hija. Sin embargo, utilizando la palabra reservada **super** podemos acceder a los atributos, métodos y constructores de la clase padre desde la clase hija.

Ésta es similar a la que vimos en el curso anterior **this** (que nos permitía acceder a los atributos, métodos y constructores de nuestra clase), pero en el caso de super llamamos/ accedemos a los atributos, métodos y constructores de la clase padre.

- **super.atributo**: accede a un atributo de la clase padre.
- **super.metodo():** accede a un método de la clase padre.
- **super():** llamamos al constructor vacío del padre.
- **super(objeto):** llamamos al constructor de copia de la clase padre.
- **super(param1, param2…):** llamamos al constructor de parámetros del padre, con los parámetros que necesita.

Veamos un **ejemplo** de **cómo acceder a constructores de la clase padre e invocarlos**:

CLASE PADRE

```
public class Vehiculo {

   public Vehiculo(){
      System.out.printIn("Creando vehículo");
   }
    public Vehiculo(Vehiculo veh){
        color = veh.color;
        longitud = veh.longitud;
        …
   }
}
```

La clase padre tiene un **constructor** que implementa el método imprimir "Creando vehículo" cada vez que creemos un objeto de la clase Vehiculo.

CLASE HIJA

```
public class Coche extends Vehiculo {
       public Coche(){
              super(); /*llamamos al constructor
                        vacío de la clase padre*/
              System.out.printIn ("Creando coche");
       }
       public Coche(Coche c) {
              super(c); /*llamamos al constructor
                        de copia de la clase padre*/
              numeroPuertas = c.numeroPuertas;
       }
}
```

Desde nuestra clase Coche, usando el comando **super (),** podremos acceder al constructor de la clase padre Vehículo.

Cuando creemos nuestro objeto la consola nos imprimirá la frase "Creando Vehículo" heredada y nuestra propia frase "Creando Coche"

Ten en cuenta que… Cuando se crea un objeto de una clase que hereda de otra, de forma implícita, si no se ha indicado lo contrario, se ejecuta primero el constructor por defecto (constructor vacío) de la clase padre, empezando por la más general, y va bajando en la jerarquía de herencia, pero si necesitas usar otro constructor (de copia o parámetros) deberías llamarlo de forma explícita.

1.3. SOBRECARGA Y SOBRESCRITURA DE MÉTODOS

¿QUÉ ES LA SOBRECARGA DEL MÉTODO?

La **firma** de un método es la combinación del nombre y su lista de tipos de parámetros.

La **sobrecarga** de métodos es la creación de varios métodos con el **mismo nombre**, pero con **diferente lista de tipos de parámetros**, además su implementación puede ser diferente.

Java utiliza el número y tipo de parámetros para seleccionar qué definición de método ejecutar, es decir, Java diferencia los métodos sobrecargados en base al número y tipo de parámetros o argumentos que tiene el método y no por el tipo que devuelve.

```
/* Métodos sobrecargados porque su lista de parámetros es distinta*/
int calculaSuma(int x, int y, int z){
    ...
}
int calculaSuma(double x, double y, double z){
    ...
}
/* Error: estos métodos no están sobrecargados, ya que su lista de parámetros es
igual, aunque su retorno es distinto*/
int calculaSuma(int x, int y, int z){
     ...
}
double calculaSuma(int x, int y, int z){
    ...
}
```

Nos da la posibilidad de tener dos o más tareas a ejecutar con el mismo nombre pero que tengan diferente **firma** (reciban parámetros diferentes), con lo cual, a la hora de ejecutar Java sabe que método debe ejecutar en función de la llamada a dicho método.

Para nuestro **ejemplo** de la clase Coche: Imagina que queremos crear un método que imprima alguno de los atributos de nuestro coche, por ejemplo, el **color** y el **año de fabricación** del coche. Hay que tener en cuenta que los parámetros serán diferentes en cada caso. En el caso del color, será un atributo de tipo String (texto), y para el año será un atributo de tipo int (número entero).

En ambos casos usaremos el mismo método, que hemos llamado **imprimir().** Por tanto, aunque usemos el mismo método en ambos casos, como los parámetros son diferentes, se dice que estaríamos **sobrecargando el método:**

```
/* Métodos sobrecargados */
public void imprimir(String color) {
   System.out.printIn ("El color del coche es " + color);
}
public void imprimir(int agno) {
   System.out.printIn ("El año de fabricación del coche es " + agno);
}
public static void main(String[] args) {
   imprimir("azul");
   imprimir(1980);
}
```

El método tiene el **mismo nombre** (Imprimir), pero **distinta firma**, es decir los parámetros son diferentes (color, años) y de distinto tipo de dato (String e int).

Firma: en lenguaje de programación cuando hablamos de la firma nos referimos a los parámetros del método.

¿QUÉ ES LA SOBRESCRITURA DE MÉTODOS?

La **sobrescritura de métodos** sólo puede darse en una relación de **herencia**, es decir, las subclases pueden sobrescribir los métodos que han heredado de la superclase o clase padre.

La sobrescritura es la forma por la cual una **clase hija** que hereda puede re-definir los métodos (añadir o modificar la implementación) de su clase padre, pudiendo crear nuevos métodos con el mismo nombre de su superclase pero con distinta funcionalidad.

Así como en la sobrecarga nos fijamos en la lista de parámetros, en la sobrescritura nos fijaremos en la estructura o firma del método sea igual a la de la clase padre. En este sentido, la clase hija:

- Debe tener la **misma firma** que la clase padre.
- Debe tener el **mismo nombre**, **mismo número de argumentos** y **mismo tipo de retorno** (o al menos un subtipo de este).
- No puede tener **un nivel de acceso más restrictivo** que la clase padre, es decir, si el modo de acceso en la clase padre es protected, en la clase hija no puede ser private.
- **No se pueden sobrescribir métodos static ni final**, ya que static representa métodos globales y final métodos constantes.

VENTAJAS DE LA SOBRESCRITURA DE MÉTODOS

La sobrescritura de métodos nos permite **extender o modificar la funcionalidad** de un método heredado de la clase padre sin tener que ir a la superclase (clase padre) a cambiarlo. Es decir, nos permite modificar ese método y hacerlo más específico en la subclase (clase hija). Por ejemplo, un animal hace un sonido y un perro hará el sonido “guau”.

La sobrescritura del método se implementa escribiendo ***@Override*** encima del método de la clase hija.

Ejemplo: continuando con nuestra clase Coche que heredaba los métodos de la clase padre Vehículo (arrancar, acelerar, frenar). Imaginemos que el método arrancar de la clase padre hace que se imprima la frase “Arrancar Vehiculo”, pero nosotros queremos especificar para nuestra clase el tipo de vehículo que arrancamos (coche).
Vemos como sería…

CLASE PADRE

```
Public class Vehiculo{
    public void arrancar(){
        System.out.println(“Arrancar el vehiculo");
  }
}
```

CLASE HIJA

```
public class Coche extends Vehiculo {
    @Override
    public void arrancar() {
        System.out.println(“Arrancar el coche");
    }
 }
```

Mismo método (estructura, nombre y acceso), **diferente implementación** (el mensaje es distinto).

Ten en cuenta que… Cuando sobrescribamos métodos debemos escribir la anotación **@Override** encima del método de la clase hija que queramos sobrescribir.

1.4. POLIMORFISMO

¿EN QUÉ CONSISTE EL POLIMORFISMO?

El **polimorfismo** es uno de los **principios básicos de la programación orientada a objetos** (POO). Se refiere a la capacidad de proporcionar a una variable de referencia (instancia) la habilidad de **cambiar el comportamiento** de acuerdo al objeto que esté manejando en ese momento.

Estas son algunas de las ventajas del polimorfismo:

- Podemos utilizar un objeto de la subclase cuando espere un objeto de la superclase.
- Una operación puede mostrar diferentes comportamientos en diferentes escenarios.
- El comportamiento de un objeto o instancia depende de los tipos de datos utilizados en la operación.
- Se usa ampliamente para implementar la herencia.

En el **ejemplo** hemos creado una instancia a la que hemos llamado "auto", y esta toma distintos comportamientos dependiendo de la clase que usamos para su creación.

En la clase Vehiculo tenemos un método mostrarTipoVehiculo, que está sobrescrito en cada clase hija según sus necesidades. Cuando creamos la instancia, dependiendo del tipo de la clase, esa instancia se comporta de una forma u otra al llamar a ese método.

Ejemplo:

```
public class Vehiculo {
   public void mostrarTipoVehiculo() {
      System.out.println("Esto es un vehículo");
   }
}
//Coche es una clase hija de Vehiculo
public class Coche extends Vehiculo{
   @Override //Sobrescribimos el método mostrarTipoVehiculo de la clase Vehiculo
   public void mostrarTipoVehiculo() {
      System.out.println("Esto es un coche");
   }
}
// Deportivo es una clase hija de Coche y nieta de Vehiculo
public class Deportivo extends Vehiculo {
   @Override //Sobrescribimos el método mostrarTipoVehiculo de la clase Vehiculo
   public void mostrarTipoVehiculo() {
      System.out.println("Esto es un deportivo");
   }
}
public static void main(String[] args) {
   /*La variable de referencia o instancia "auto" tomará la forma o comportamiento
     dependiendo de la clase que usamos para crear la instancia */
   Vehiculo auto = new Vehiculo();
   auto.mostrarTipoVehiculo();  //Mostrará por consola "Esto es un vehículo"

   auto = new Coche();
   auto.mostrarTipoVehiculo();  //Mostrará por consola "Esto es un coche"

   auto = new Deportivo();
   auto.mostrarTipoVehiculo();  //Mostrará por consola "Esto es un deportivo"
}
```

No olvides que... Como ya vimos en la herencia, en cada clase podemos especializar más el objeto: añadiendo nuevos atributos o métodos, haciendo de esta manera que se comporte de diferente manera en cada caso.

TIPOS DE POLIMORFISMO SEGÚN SU ENLACE

El polimorfismo se puede establecer mediante la sobrecarga o sobrescritura de métodos. El polimorfismo en Java tiene dos tipos según su **enlace**:

- **Polimorfismo en tiempo de compilación o enlace estático**. Se logra mediante la **sobrecarga** de métodos en la misma clase.
- **Polimorfismo en tiempo de ejecución o enlace dinámico**. Se logra mediante la **sobrescritura** de métodos en diferentes clases.

A continuación, veremos un ejemplo de cada uno de ellos.

ENLACE ESTÁTICO

El **enlace estático** o polimorfismo en tiempo de compilación, es la habilidad de llamar a un método específico durante el tiempo de compilación, basándose en la firma del método, ya que este tipo de polimorfismo se logra mediante la **sobrecarga** de métodos, donde varios métodos están presentes en la misma clase, pero con diferentes tipos y número de parámetros, así que las llamadas de métodos sobrecargadas se resuelven en tiempo de compilación por el compilador.

En el **ejemplo** creamos la clase Vehiculo y, dentro de ésta, **sobrecargamos** el método **arrancar()** para que actúe de forma diferente si recibe o no un parámetro.

Ejemplo:

```
public class Vehiculo {
   public void arrancar() {
      System.out.println("Arrancando el vehículo");
   }
   public void arrancar(String modelo) {
      System.out.println("Arrancando el " + modelo);
   }
   /*Aquí ya tenemos el polimorfismo en tiempo de compilación porque, al sobrecargar
     los métodos, ya sabemos que dependiendo de la firma tendrá un comportamiento
     distinto */
}
```

Enlace: es la acción de asociar un objeto con el método correspondiente en la clase receptora.

ENLACE DINÁMICO

El **enlace dinámico** o polimorfismo en tiempo de ejecución, es la habilidad de llamar a un método específico en tiempo de ejecución, basándose en la **sobrescritura** de un método en diferentes clases. El tipo de objeto en el que se invoca el método no se conoce en el momento de la compilación, por eso se decidirá en el tiempo de ejecución.

En este **ejemplo** tenemos los dos tipos:

- Enlace estático por la **sobrecarga** del método **arrancar()** en la clase Coche. En tiempo de compilación ya sabemos que dependiendo de la firma tendrá un comportamiento distinto.
- Enlace dinámico por la **sobrescritura** del método **arrancar()** de la clase Vehiculo dentro de la clase Coche. Hasta que no se ejecute no sabemos a cuál de los métodos llamará.

Ejemplo:

```
public class Vehiculo {
   public void arrancar() {
      System.out.println("Arrancando el vehículo");
   }
}

public class Coche extends Vehiculo{
   @Override //Sobrescribimos el método arrancar de Vehiculo
   public void arrancar() {
      System.out.println("Arrancando el coche");
   }
   //Sobrecargamos el método arrancar dentro de la clase Coche
   public void arrancar(String modelo) {
      System.out.println("Arrancando el " + modelo);
   }
}

public static void main(String[] args) {
   Vehiculo auto = new Vehiculo();
   auto.arrancar();  //Mostrará por consola "Arrancando el vehículo"

   auto = new Coche();
   auto.arrancar();  //Mostrará por consola "Arrancando el coche"
   auto.arrancar("Mercedes");  //Mostrará por consola "Arrancando el Mercedes"
}
```

1.5. PREGUNTAS FRECUENTES

- **¿Java permite la herencia múltiple?**

  No, aunque esta puede llevarse a cabo de forma ficticia mediante el uso de interfaces.

- **¿Se pueden heredar constructores?**

  No, aunque mediante la palabra reservada super se puede acceder a los constructores de la clase padre e invocarlo desde la clase hija.

- **¿Se puede sobrescribir un método y cambiarle el tipo de acceso?**

  Sí, siempre que este no sea más restrictivo que el de la clase padre. Si el método en la clase padre es public, no podrá ser protected o private en la clase hija, pero sí al revés.

FUNDACIÓN ACCENTURE

accenture