Unidad 3. Java Orientado a Objetos

Contenido

1 Actividad inicial

Realizamos una pregunta sobre el mundo real de los alumnos. Pedimos a los alumnos que características físicas o de otro tipo han heredado de sus padres y abuelos, y que las vayan apuntando en el bloc de notas. Preguntamos también que comportamientos han heredado de sus padres y abuelos y que lo apunten

Herencia de padres y abuelos de los alumnos	
Características	Comportamientos
Ojos marrones (padre)	Gusto por la música (abuelo)
Pelo rizado (abuelo)	Calmado (abuelo)

Cuando se explique herencia podremos realizar una analogía con esta tabla

1.1 Introducción.

En este **capítulo vamos a explicar los diferentes patrones o acercamientos que usaremos para resolver nuestros problemas de programación en programación orientada a objetos**. Empezaremos por los patrones más sencillos e iremos avanzando hacia patrones más complicados.

Recordamos antes los pilares de la programación orientada a objeto que son cuatro:

1. **Abstracción: abstraer en orientación a objetos es representar la realidad a través de nuestras clases**. Representamos de manera precisa los detalles necesarios sobre un contexto real, omitiendo el resto.
2. **Herencia: permitimos crear clases heredando de otras**. De esta manera hay código ya escrito, lo heredamos, y lo utilizamos en las nuevas clases sin volver a escribirlo.
3. **Encapsulacion**: cuando creamos clases en nuestros programas ofrecemos para cada clase unos métodos que son servicios. **El funcionamiento interior de esa clase es irrelevante para las clases** que **usan ese servicio**. Por ejemplo, para **arrancar mi coche no necesito saber como esta compuesto el sistema eléctrico**, sólo **la llave para arrancarlo y girarla**. Ese es el servicio que me ofrecería una clase SistemaDeArranque.
4. **Polimorfismo**: tiene que ver como su etimología indica con múltiples formas, múltiples comportamientos. En **herencia vamos a tener una clase principal y múltiples subclases**. La **clase principal o el interfaz ofrece un comportamiento común que las subclases hijas** modifican **adaptándose a sus propias características.**

2 Lenguaje Java orientado a objetos

La programación orientada a objetos se basa en cómo, en el mundo real, los objetos a menudo se componen de muchos tipos de objetos más pequeños. Esta capacidad de combinar objetos, sin embargo, es sólo un aspecto muy general de la programación orientada a objetos. La programación orientada a objetos proporciona varios otros conceptos y características para que la creación y el uso de objetos sean más fáciles y flexibles, y la más importante de estas características es la de las clases.

Una *clase* es una plantilla para varios objetos con características similares. Las clases incorporan todas las características de un conjunto determinado de objetos. Cuando se escribe un programa en un lenguaje orientado a objetos, no se definen objetos reales. Las clases de objetos se definen.

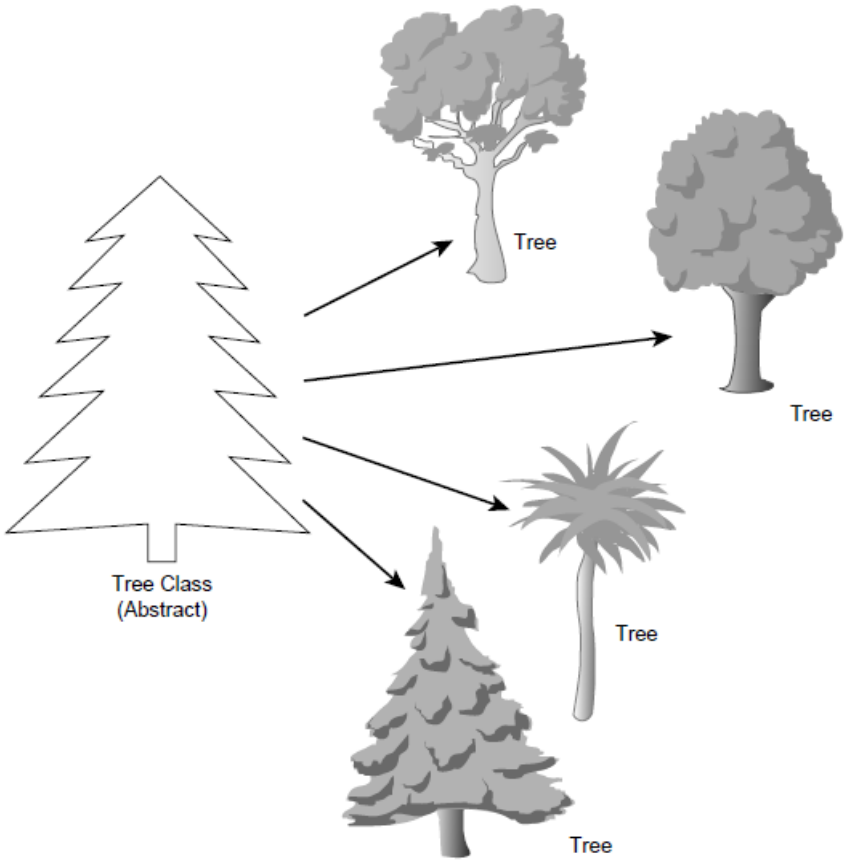
Por ejemplo, podría tener una clase **Tree** que describa las características de todos los árboles (tiene hojas y raíces, crece, crea clorofila). La clase **Tree** sirve como un modelo abstracto para el concepto de árbol: para alcanzar y agarrar, o interactuar con, o cortar un árbol, debe tener una instancia concreta de ese árbol. Por supuesto, una vez que tenga una clase de árbol, puede crear muchas instancias diferentes de ese árbol, y cada instancia de árbol diferente puede tener diferentes características (hojas cortas, altas, tupidas, gotas en otoño), mientras sigue comportándose como y siendo inmediatamente reconocible como un árbol.

Una instancia de una clase es otra palabra para un objeto real. Si las clases son una representación abstracta de un objeto, una instancia es su representación concreta. Entonces, ¿cuál es exactamente la diferencia entre una instancia y un objeto? Nada, en realidad. Objeto es el término más general, pero tanto las instancias como los objetos son la representación concreta de una clase.

De hecho, los términos instancian y objeto a menudo se usan indistintamente en el lenguaje de POO (Programación orientada a objetos). Un instancia de un árbol y un objeto de árbol son la misma cosa. En un ejemplo más cercano al tipo de cosas que podría querer hacer en la programación Java, puede crear una clase para el elemento de la interfaz de usuario denominada botón.

La clase Button define las características de un botón (su etiqueta, su tamaño, su apariencia) y cómo se comporta (¿necesita un solo clic o un doble clic para activarlo, cambia de color cuando se hace clic, qué hace cuando se activa?).

Una vez definida la clase Button, puede crear fácilmente instancias de ese botón (es decir, objetos de botón) que todos toman las características básicas del botón según lo definido por la clase, pero que pueden tener diferentes apariencias y comportamientos en función de lo que desea que haga ese botón en particular. Al crear una clase, no tiene que seguir reescribiendo el código para cada botón individual que desee usar en el programa, y puede reutilizar la clase **Button** para crear diferentes tipos de botones a medida que los necesite en este programa y en otros programas.



2.1 Pilares de la programación orientada a objetos

La programación orientada a objetos se basa en cuatro pilares, conceptos que la diferencian de otros paradigmas de programación. Estos pilares, algunos principios que daremos a conocer en unidades posteriores, reglas y convenciones para la programación orientada a objetos se originan a partir de principios de los años 90.
Las compañías de software y los ingenieros de software buscan código de alta calidad para ofrecer mejores productos de software y reducir el precio del mantenimiento y la fabricación.

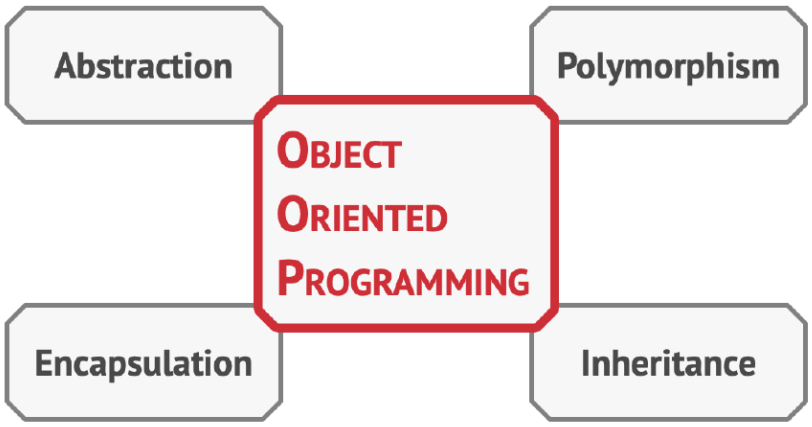
En esa búsqueda, Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides escribieron un libro ***Design Patterns: Elements of Reusable Object-Oriented Software***. Ha sido influyente en el campo de la ingeniería de software y es considerado como una fuente importante para la teoría y la práctica del diseño orientado a objetos. Se han vendido más de 500.000 ejemplares en inglés y en otros 13 idiomas.

Asimismo, en los años 90 se celebraron muchas conferencias sobre ingeniería de software, para mejorar la producción de software y sistemas de información. Seguiremos tantas de estas "reglas" que la mayoría de los desarrolladores y académicos reconocidos y admirados en Ciencias de la Computación ha fijado desde ese período.

Los conceptos de OOP nos permiten crear interacciones específicas entre objetos Java. Permiten reutilizar el código sin crear riesgos de seguridad ni hacer que un programa Java sea menos legible.

Aquí están los cuatro principios principales con más detalle.

1. abstracción
2. encapsulación
3. herencia
4. polimorfismo

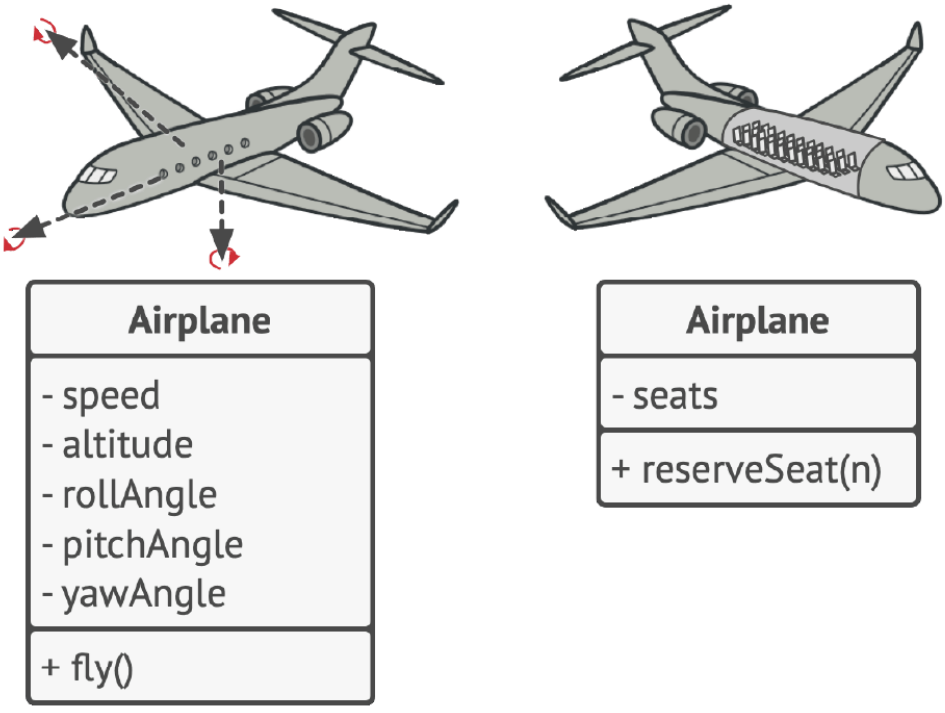


2.1.1 Abstraction

La abstracción es un modelo de un objeto o fenómeno del mundo real, limitado a un contexto específico, que representa todos los detalles relevantes para este contexto con alta precisión y omite todo lo **demás.**

La mayoría de las veces, cuando se crea un programa con POO, se da forma a los objetos del programa en función de objetos del mundo real. Sin embargo, los objetos del programa no representan los originales con una precisión del 100% (y rara vez se requiere que lo hagan). En su lugar, los objetos solo *modelan* atributos y comportamientos de objetos reales en un contexto específico, omitiendo el resto.

Por ejemplo, una clase Airplane probablemente podría existir tanto en un simulador de vuelo como en una aplicación de reserva de vuelos. Pero en el primer caso, albergaría detalles relacionados con el vuelo real, mientras que en la segunda clase solo le importaría el mapa de asientos y qué asientos están disponibles.



2.1.2 Encapsulación

Para arrancar el motor de un coche, sólo tiene que girar una tecla o pulsar un botón. No es necesario conectar cables bajo el capó, girar el cigüeñal y los cilindros, e iniciar el ciclo de potencia del motor. Estos detalles se esconden

bajo el capó del coche. Solo tienes una interfaz simple: un interruptor de arranque, un volante y algunos pedales. Esto ilustra cómo cada objeto tiene una interfaz, una parte pública de un objeto, abierta a interacciones con otros objetos.

La encapsulación es la característica de cada módulo en nuestro código para ocultar datos u operaciones que no son necesarias para revelar a otros módulos de nuestro código. Por ejemplo, suponiendo que tengo una llamada de módulo StoreEmployees que almacena datos de empleados. Otros módulos de nuestra aplicación, como windows (la interfaz gráfica) utilizarán este módulo para guardar datos. Sin embargo, esta interfaz gráfica no necesita saber si StoreEmployees guarda los datos en un archivo o una base de datos. Esos detalles de implementación están ocultos, encapsulados. Por lo tanto, el Window sólo necesita una función storeEmployeeData(Employee data) para utilizar esta funcionalidad.

La encapsulación en la programación orientada a objetos es la capacidad de un objeto para ocultar partes de su estado y comportamientos de otros objetos, exponiendo sólo una interfaz limitada al resto del programa.

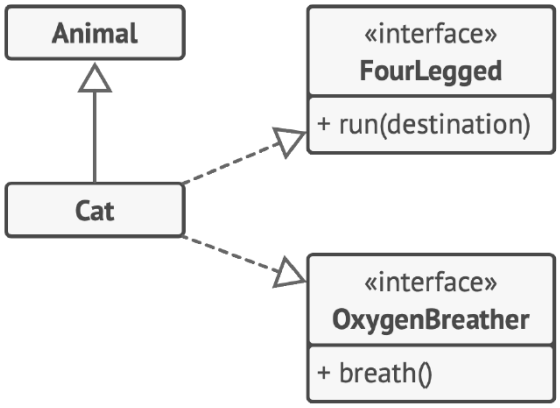
Encapsular algo significa hacerlo private y, por lo tanto, accesible solo desde dentro de los métodos de su propia clase.

Hay un modo un poco menos restrictivo llamado protected que hace que un miembro de una clase también esté disponible para las subclases. Las interfaces y las clases/métodos abstractos de la mayoría de los lenguajes de programación se basan en los conceptos de abstracción y encapsulación.

2.1.3 Herencia

La herencia es la capacidad de construir nuevas clases sobre las existentes. La principal ventaja de la herencia es la reutilización de código. Si se desea crear una clase que es ligeramente diferente de una existente, no es necesario duplicar el código. En su lugar, se extiende la clase existente y se coloca la funcionalidad adicional en una subclase resultante, que hereda los campos y métodos de la superclase.

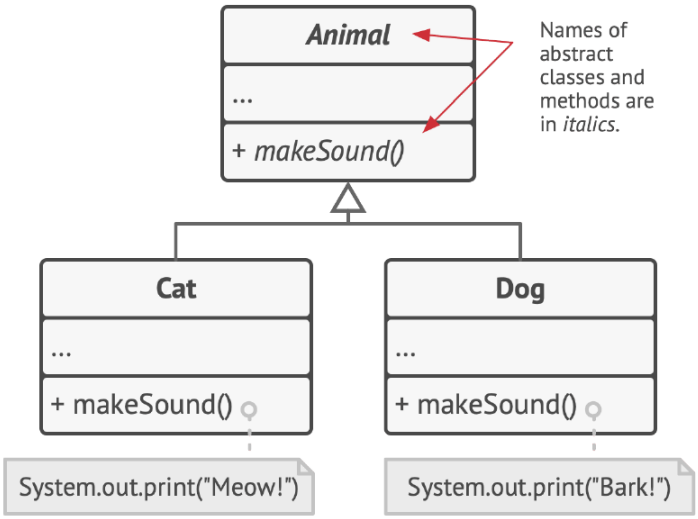
La consecuencia de utilizar la herencia es que las subclases tienen la misma interfaz que su clase primaria. No se puede ocultar un método en una subclase si se declaró en la superclase. Tú también debe implementar todos los métodos abstractos, incluso si no tienen sentido para la subclase.



2.1.4 Polimorfismo

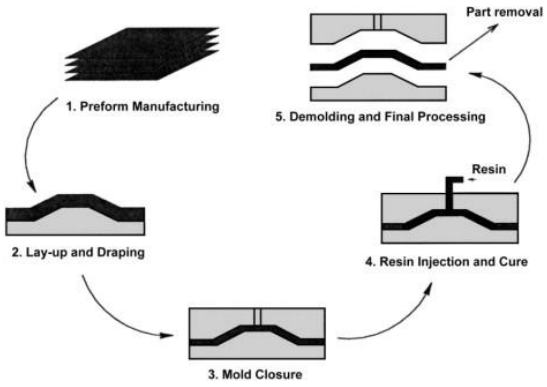
La palabra polimorfismo proviene de las palabras griegas para "muchas formas". Un método polimórfico, por ejemplo, es un método que puede tener diferentes formas, donde "forma" se puede considerar como tipo o comportamiento.

Veamos algunos ejemplos de animales. La mayoría de los animales pueden hacer sonidos. Podemos anticipar que todas las subclases necesitarán reemplazar el método makeSound base para que cada subclase pueda emitir el sonido correcto; por lo tanto, podemos declararlo *abstracto* de inmediato. Esto nos permite omitir cualquier implementación predeterminada del método en la superclase, pero forzar a todas las subclases a crear las suyas propias.



2.2 Java como lenguaje orientado a objetos.

Java es orientado a objetos en su naturaleza. Significa que los programas se componen de objetos. Sin embargo, para crear o crear objetos, necesitaría una plantilla. Una analogía para describir clases y objetos podría hacerse de la fabricación. Cuando se necesita construir cien mil pomos de puertas para automóviles, debe seguir algunos pasos. En primer lugar, se debe diseñar el trabajo de carrocería del coche. Luego construyes un molde para esa pieza. Y por último, se deber ir a producción en masa.

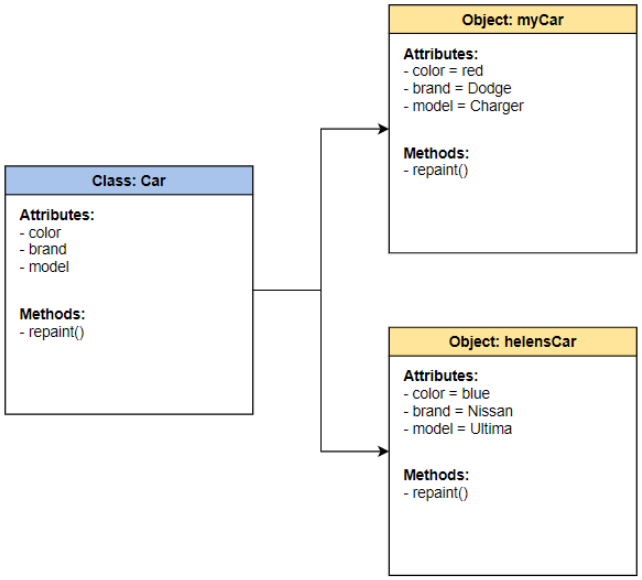


Para programar en un lenguaje orientado a objetos, se necesita su molde, su plantilla para crear un conjunto de objetos. Por ejemplo, digamos que creamos una clase, `Car,` para contener todas las propiedades que debe tener un automóvil, `color,` `marca` y `modelo.` A continuación, cree una instancia de un objeto de tipo `Car,` `myCar` para representar mi coche específico.

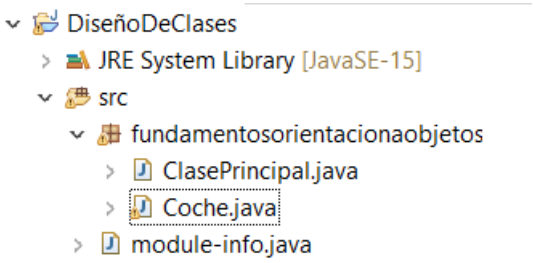
A continuación, podríamos establecer el valor de las propiedades definidas en la clase para describir mi coche, sin afectar a otros objetos o la plantilla de clase. A continuación, podemos reutilizar esta clase para representar cualquier número de coches.

Ya describimos esta acción. Es el primer pilar de la Programacion Orientada a Objetos. *La abstracción* es un modelo de un objeto o fenómeno del mundo real. Hemos abstraído el coche del mundo real, en una representación de codificación llamada clase Car. Dado que estamos representando el coche a través de software, agregamos las propiedades y los comportamientos que necesitamos controlar en nuestro programa. Suponiendo que estamos vendiendo coches nuevos, no necesitamos almacenar el cuentakilómetros, que señalan el número de kilómetros de nuestro coche.

Hemos tomado la decisión de dejar fuera de nuestra representación esta característica.



Vamos a crear este proyecto DiseñoDeClases en Eclipse. Incluiremos estas dos clases, pero primero agregaremos la clase principañ ClasePrincipal.java.




```
package oppfundamentals;

public class ClasePrincipal {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		System.out.println("Hello there. Howdy. My first java program on
the running");
```

```
	}

}
```

Cada archivo de código fuente en Java contiene menos una clase con el modificador public. Por lo tanto, para programar en Java necesitamos agregar clases a nuestro código. Para explicar el objeto y las clases lo primero que haremos es añadir uno a este proyecto. Por favor, agregue el coche de clase al proyecto y copie el código siguiente en el archivo.

```
package fundamentosorientacionaobjetos;
public class Coche {
	private String color;
	private String marca;
	private String modelo;

	public Coche (String color, String marca, String modelo) {

			this.color=color;
			this.marca=marca;
			this.modelo = modelo;

	}

	public void repintar(String color) {


		this.color=color;
	}

	@Override
	public String toString() {
		return "Car [color=" + color + ", marca=" + marca + ", modelo="
+ modelo + "]";
	}



}
```

Vamos a desglosar la clase en cada elemento diferente. Resaltado en amarillo, puede encontrar propiedades o atributos de **clase**. Están destinados a almacenar información sobre cada coche individual. Esta información es lo que llamamos el estado del objeto. Por ejemplo, suponiendo que el coche es rojo, su estado actual es ser rojo. Si repintamos el coche cambiamos su estado. Las propiedades de clase se comportan como variables que estudiamos en la Unidad 1 y en este documento. Tienen un tipo y un nombre. En este ejemplo, el tipo es String.

```
	private String color;
	private String marca;
	private String modelo;
```

En el ejemplo anterior, la clase tenemos tres propiedades: color, marca y modelo. Propiedades como se puede ver, describe el coche. Esto es menos de la mitad de un objeto. La otra mitad es lo que el objeto puede hacer, su comportamiento. El coche en nuestro programa puede ser repintado. Para dotar a los objetos de comportamiento, que son las acciones que pueden realizar, en las clases podemos definir métodos. Se puede inferir que los métodos son similares a las subrutinas. Tiene razón, son casi lo mismo. Sin embargo, estas funciones pertenecen a objetos y sólo se pueden aplicar a través de objetos.

La primera función que se encuentra normalmente en una clase es el constructor. Este es el método más singular porque permite a los programadores crear objetos como explicaremos más adelante. Es muy particular ya que no tiene un tipo de valor devuelto. El tipo de valor devuelto es implícito, un objeto de esta clase.

```
public Car (String color, String marca, String modelo) {

		this.color=color;
		this.marca=marca;
		this.modelo = modelo;

	}
```

```
public  NO DEVOLVER TIPO AQUÍ Car (String color, String marca, String modelo)
{
```

Después del constructor, los programadores suelen escribir métodos para agregar una operación o un comportamiento al objeto. Para agregar operaciones a una clase usamos métodos. Recuerde que los métodos pueden ser procedimientos o funciones. El método de repintado permite que el coche sea repintado. Cambia el comportamiento del coche. Esta es una característica intrínseca al coche. Una persona no puede ser repintada, ni el mar.

```
public void repintar (String color) {

		this.color=color;
	}
```

Por último, el método toString() devuelve una descripción del contenido del objeto en string. Java utiliza el método toString para poder imprimir el método en la salida. De lo contrario, imprimiría una dirección de memoria, como explicamos más adelante.

```
@Override
	public String toString() {
		return "Car [color=" + color + ", marca=" + marca + ", modelo="
+ modelo + "]";
	}
```

En general, hay algunos métodos que permiten crear objetos, otros nos permiten hacer acciones, cálculos o devolver datos procesados. Sin embargo, estamos hablando de objetos, pero no tenemos uno. Tenemos un plano, un molde, pero aún no hemos creado ni objetado.

2.2.1 El operador this, referencia a Objeto vs referencia a clase

Cuando se codifica dentro de una clase, tenemos a nuestra disposición una herramienta, o una clave reservada para acceder al objeto en cualquier lugar. Se denomina operador this. Esto funciona de forma similar a una variable declarada como car, Car myCar. Ir proporciona acceso a todas las propiedades y métodos del objeto. En realidad, señala el objeto que ha creado para esta clase.

```
public abstract class Car {
	protected String color;
	private String marca;
	private String modelo;

	public  Car (String color, String marca, String modelo) {

			this.color=color;
			this.marca=marca;
			this.modelo = modelo;
```

This.color hace referencia a la propiedad protected String color;. Al hacerlo, puede distinguir entre el parámetro Color de Tipo String y la propiedad protected String color; Debes tener en cuenta que el operador this solo se puede utilizar dentro del código de clase. Además, el operador this hace referencia a un objeto que se ha creado, no a la clase. No funciona a menos que se haya creado un objeto. No podemos usarlo en métodos estáticos.

Para hacer referencia a una clase, debe escribir el nombre de la clase, normalmente seguido de una constante estática o un método estático. Cambie la última línea del constructor Coche, agregando la referencia de clase al método estático. Hace que el código sea más legible y limpio.

```
Coche.incrementarNumeroDeCoches();

 public  Car (String color, String brand, String model{

                    this.color=color;
                    this.brand=brand;
                    this.model = model;


                    Coche.incrementarNumeroDeCoches();

       }
```

Coche.java

```
package fundamentosorientacionaobjetos;
public class Coche {
       private String color;
       private String marca;
       private String modelo;
       private static int numCoches=0;

       public Coche (String color, String marca, String modelo) {

                    this.color=color;
                    this.marca=marca;
                    this.modelo = modelo;
                    Coche.numCoches++;

       }

       public void repintar(String color) {


              this.color=color;
```

```
	}

	@Override
	public String toString() {
		return "Car [color=" + color + ", marca=" + marca + ", modelo="
+ modelo + "]";
	}


	public static void IncrementaNumeroDeCoches() {

		Coche.numCoches++;
	}

}
```

2.2.2 Creación de objetos

Volvamos al programa principal. Crearemos un objeto de coche. El proceso se denomina creación de instancias o creación de instancias de un objeto; Por lo tanto, un objeto es una instancia de una clase.

Primero necesitamos definir una variable de programa de esta clase. En segundo lugar, necesitamos llamar al constructor de clase usando una palabra clave reservada, new. Crear un objeto es fácil, por lo tanto, agregue las dos líneas siguientes después de la oración System.out.println.

```
package fundamentosorientacionaobjetos;

public class ClasePrincipal {



public static void main(String[] args) {
		// TODO Auto-generated method stub

		System.out.println("Hola como estais");

//Add the lines after this

Coche miTesla = new Coche("rojo","Tesla","Model S High Performace");

System.out.println(miTesla);

}

}
```

Cada vez que escribe new seguido de una llamada al constructor de clase, creamos un objeto de este coche. Además, puede crear tantos objetos como necesite. Hagámoslo, incluya el próximo coche a su programa.

```
Coche miBMWi = new Coche("azul"," BMW","i5");

System.out.println(miBMWi);
```

Para utilizar parte del comportamiento del coche en su programa, vuelva a pintar el BMW en blanco.

```
myBMWi.repintar("blanco");

      System.out.println("MYBMW despúes de repintar " + myBMWi);
```

Ejecute su programa y podría obtener un resultado similar a:

```
Car [color=rojo, marca=Tesla, modelo=Model S High Performace]
Car [color=azul, marca= BMW, modelo=i5]
MYBMW después de repintar Car [color=blanco, marca= BMW, modelo=i5]
```

Ejecuta el programa de nuevo, pero primero borre el método toString de la clase Coche, agregaremos de nuevo después de estas ejecuciones.

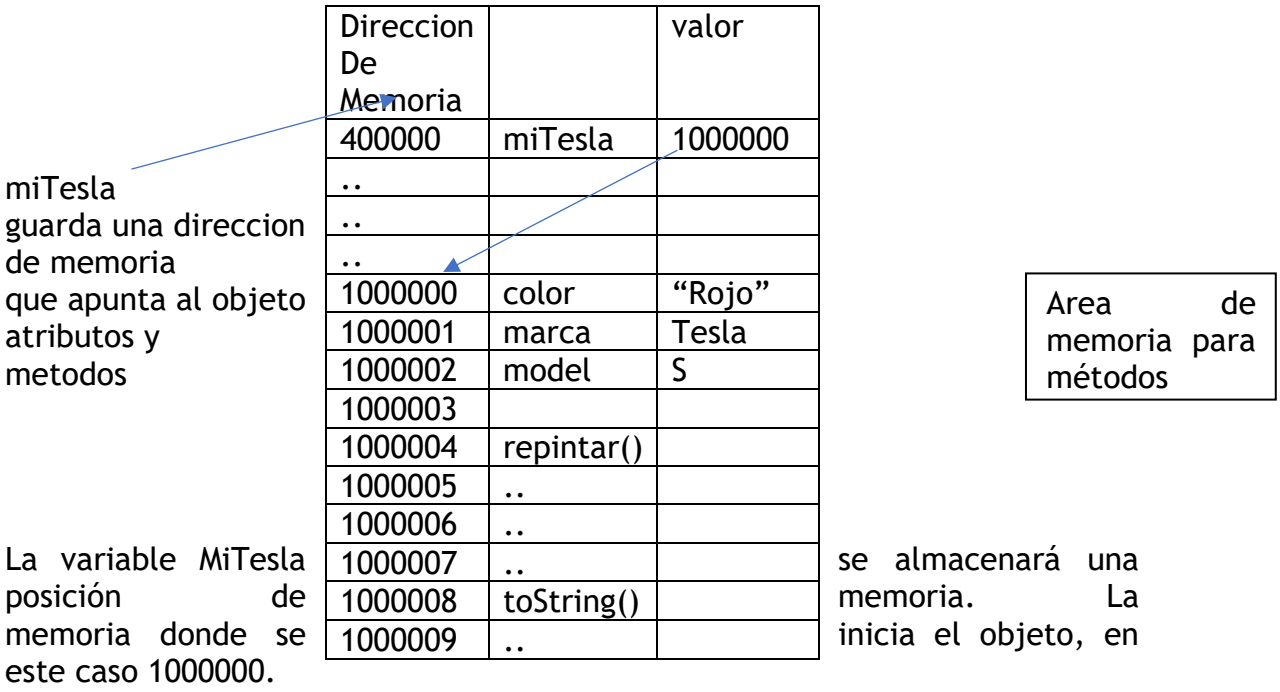
El resultado de sacar el método toString es esta ejecución:

```
fundamentosorientacionaobjetos.Coche@3830f1c0
fundamentosorientacionaobjetos.Coche @39ed3c8d
MYBMW después de repintado Coche.Coche@39ed3c8d
```

Ahora el println, imprimir el contenido real de una variable de objeto, que es una dirección de memoria en hexadecimal. Deshaga la eliminación de toString y deje la clase en su estado inicial. Este ejercicio sirve como introducción a la siguiente pregunta

Pero, ¿qué sucede cuando se crea un objeto en java?

Una variable de objeto es un puntero a una ubicación de memoria, en esta ubicación de memoria se guarda toda la información del objeto, propiedades y código.



2.2.3 Propiedades públicas frente a privadas. Encapsulación.

Para las propiedades de objeto Java ofrece métodos públicos para ser utilizados por otros objetos, sin embargo, puede ofrecer propiedades. Vamos a cambiar los modificadores de propiedad de coche que declaramos antes.

```
public class Coche {
	public String color;
	public String marca;
	public String modelo;
	public static final String TAG = "Objeto Coche";
```

y ahora agrega esta línea al final de tu programa principal.

```
	System.out.println("acceso a la propiedad color " + myBMWi.color);
```

Dado que hemos cambiado la declaración de propiedad, y debemos agregar el modificador public, ahora puede acceder al atributo color usando el punto "."

operador. Este operador proporciona acceso a cualquier declaración pública que haya agregado a las clases. Significa que a través de la variable de objeto (después de crearse) puede acceder a su estado, propiedades o sus métodos. El objeto variable es una variable que al mismo tiempo contiene una propiedad, tipo de variable.

Desde finales de los años 90, es una convención de programación orientada a objetos y un uso adecuado de lenguajes como java para no declarar propiedades como públicas siguiendo la **Encapsulación.** Encapsulación se basa en algunas medidas o evidencias de la experiencia del desarrollador:

1. El código puede ser más propenso a errores: estáticamente, otro objeto que modifique el estado de otro objeto llevará al programa al riesgo de varios errores para que sea menos predecible.

2. Pone en riesgo la integridad de los datos de su objeto, lo que resulta en que representa una amenaza para la integridad de los datos de su aplicación.

3. Hace que el código sea menos legible y mantenible.

Para concluir, está casi prohibido declarar propiedades públicas para producir código limpio.

Aplicamos el segundo Pilar de la Programación Orientada a Objetos

Aunque, a veces su objeto debe ofrecer variables constantes a otros. Esta práctica está permitida y es recomendable. Por ejemplo, la variable constante TAG se declara correctamente en caso de que otros objetos la necesiten.

```
public static final String TAG = "Objeto Coche";
```

El código limpio es un código que es fácil de leer, entender y mantener. El código limpio hace que el desarrollo de software sea predecible y aumenta la calidad de un producto resultante. A partir de ahora, no declararemos públicas las propiedades.

La idea es que se pueda acceder a los datos de los objetos a través de métodos. Los datos se pueden encapsular de tal manera que sean invisibles para el "mundo exterior".

Para obtener más detalles sobre cómo obtener software de calidad y refactorizar el código, puede revisar este enlace:

https://refactoring.guru/refactoring

Esto da lugar a dos formas de aproximarse a las clases:

1. Una clase es de tipo complejo en Java. Hemos declarado una variable `Coche miTesla`. Y este tipo se compone de otras variables, sus propiedades. Desde el punto de vista de un programador, una clase es un tipo
2. Además, una clase es una abstracción de un objeto real. Estas propiedades son las que definen y representan las características del coche. este objeto en nuestro programa. Para un diseñador de software, una clase es la realización de la realidad, que debe ser administrada por nuestro software.

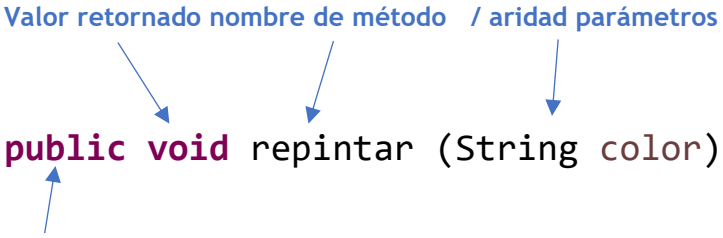
2.2.4 Métodos. Encapsulación II. Privado vs Público.

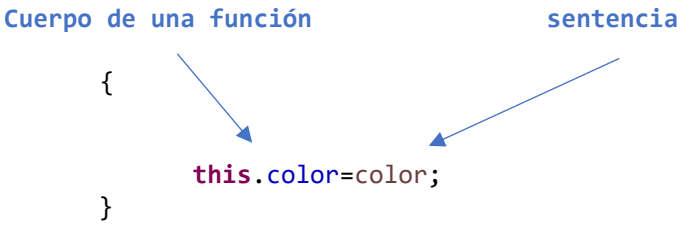
En la sección anterior se ha tratado, datos de objeto, su estado. Usando métodos, proporcionamos a los objetos operaciones, un comportamiento. Ya estudiamos que un método puede ser un procedimiento que realiza una acción pero no devuelve un resultado, o una función que devuelve resultados.

Podemos distinguir dos bloques o partes en un método:

Firma o encabezado de una función. También se conoce como la interfaz del método.
La aridad es la lista de parámetros.

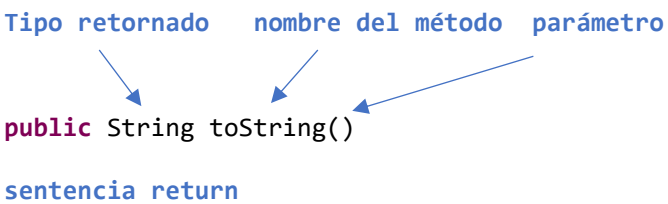
Firma o encabezado





El método repintar no devuelve valores, por lo que es un procedimiento

```
public void repintar (String color) {

		this.color=color;
	}
```

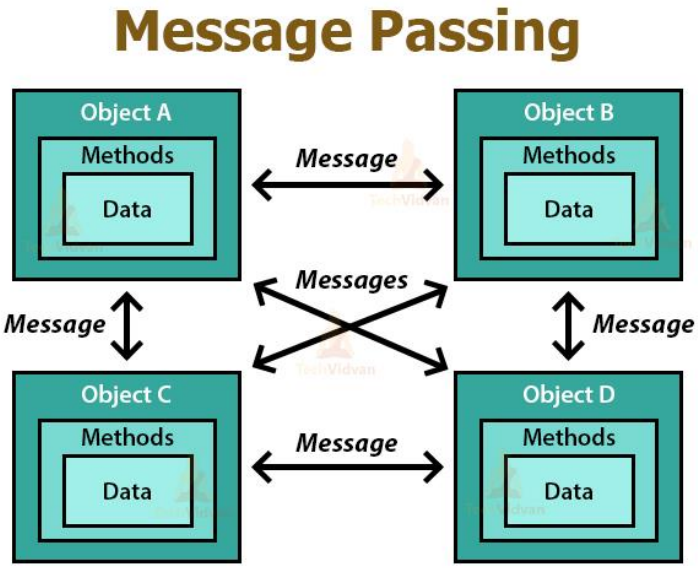


sentencia return

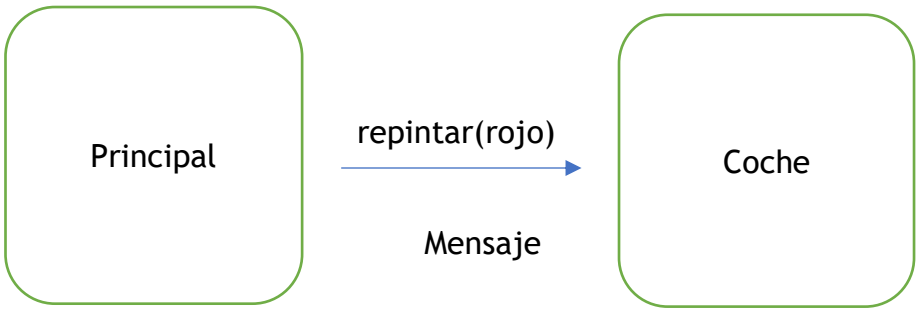
```
{
		return "Car [color=" + color + ", marca=" + marca + ", modelo="
+ modelo + "]";
	}
```

En este caso, el método toString es como una función. Por el contrario, no recibe parámetros. También podría tener métodos funcionan por igual que reciben parámetros. Pero hay otro enfoque para los métodos. Se pueden

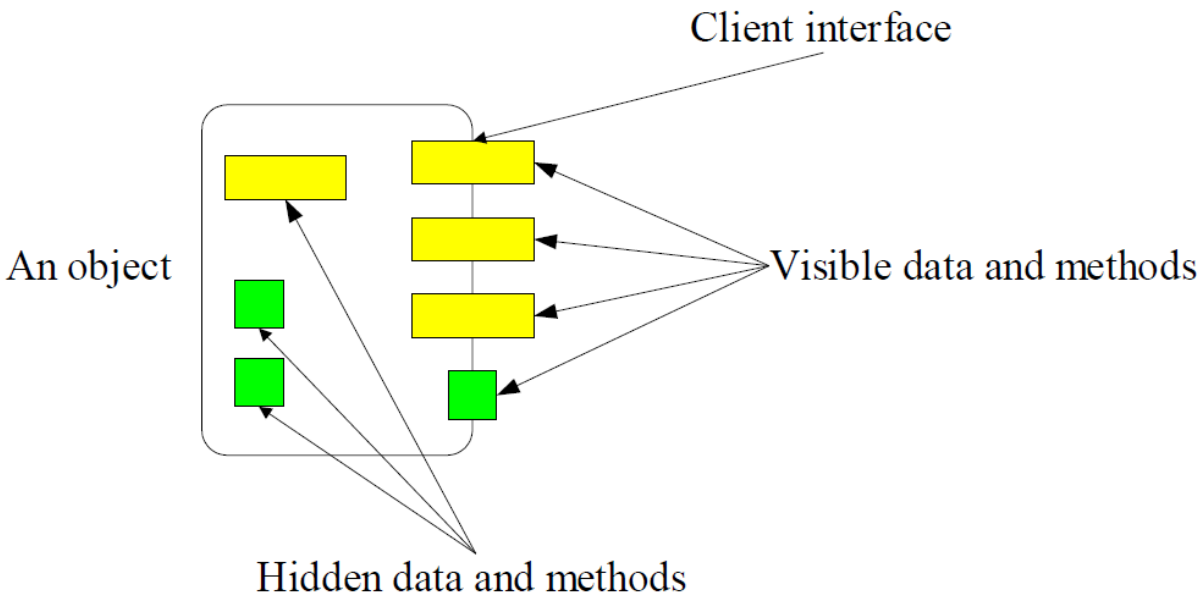
considerar como herramientas para recibir mensajes de otros objetos y enviar mensajes a otros objetos.



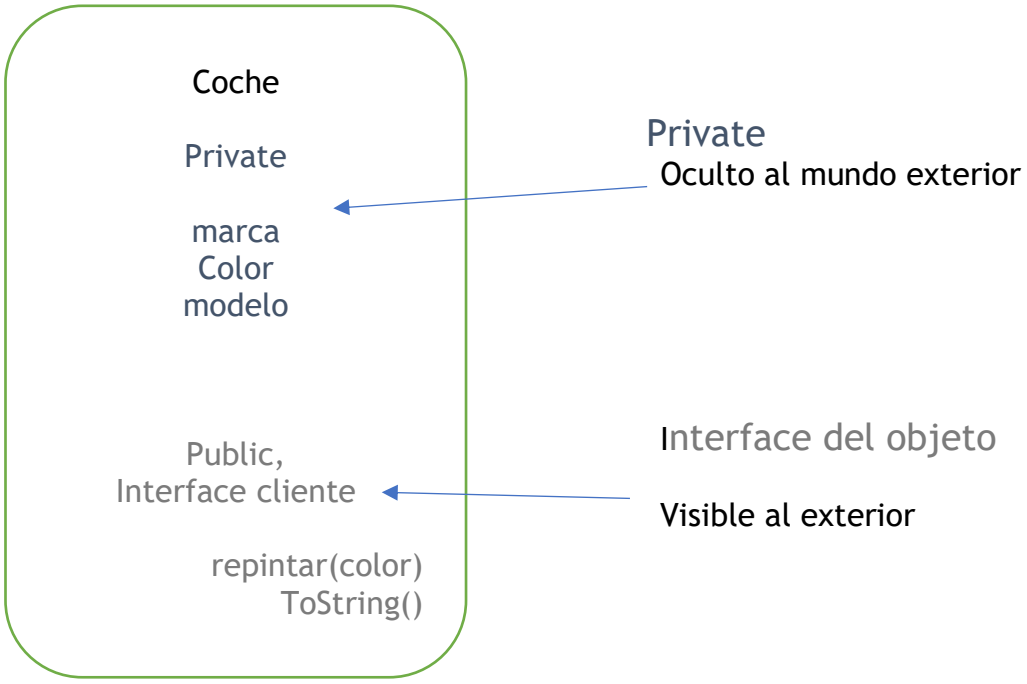
Mi clase principal enviar un mensaje al coche, para conseguir repintado en rojo.



Del mismo modo, los métodos pueden ser un medio para dar acceso a los datos públicos de los objetos. Lo que estamos tratando de lograr a través de la encapsulación es ocultar los datos y el comportamiento del objeto privado que no es necesario para otros objetos. Además, con la encapsulación estamos tratando de ofrecer una interfaz clara de la clase. Como resultado, nuestros objetos ofrecerán los métodos necesarios, "mensajes", para proporcionar su funcionalidad completa, y ocultar lo que no es conveniente mostrar, el objeto interno de trabajo.



¡De lo que el "mundo exterior" no puede ver no puede depender!
• El objeto es un "firewall" entre el objeto y el "mundo exterior".
• Los datos y métodos ocultos se pueden cambiar sin afectar al "mundo exterior".



2.2.5 Métodos de acceso.

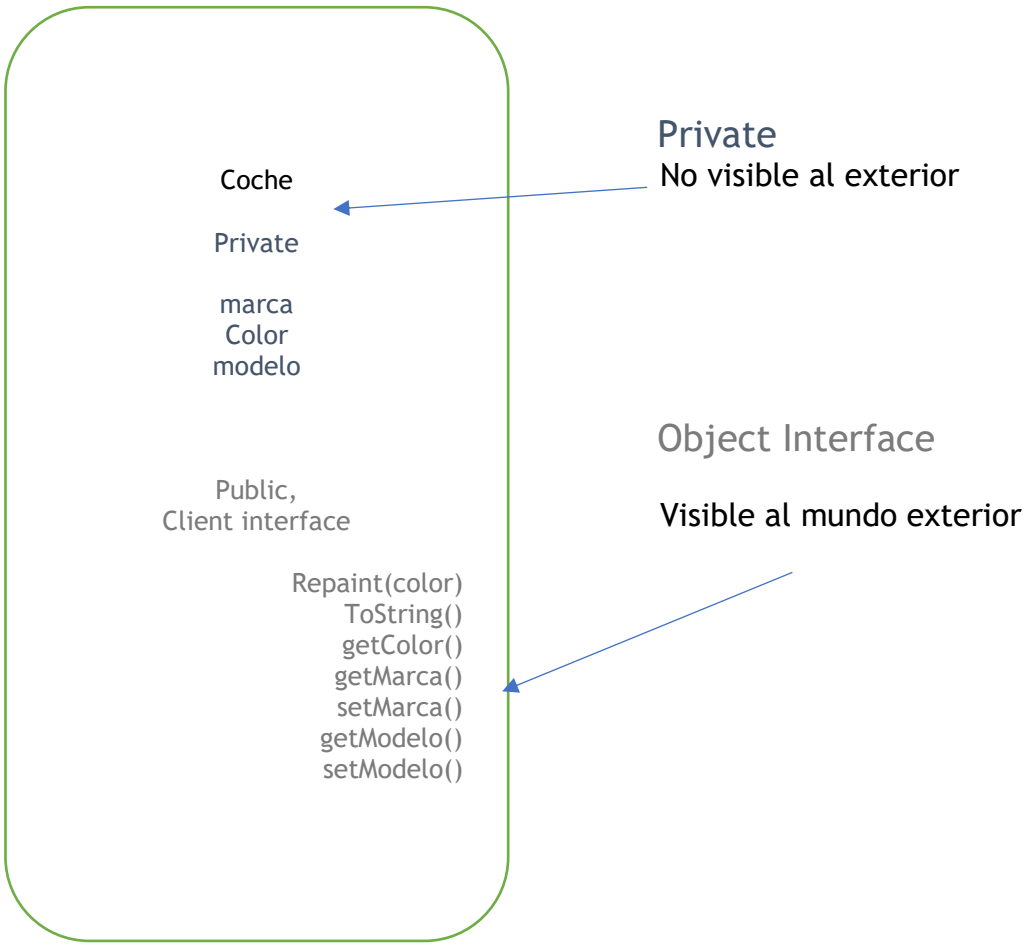
Está claro que, con el diseño actual de nuestra clase, no tenemos acceso a datos valiosos, como marca, modelo o color. Aunque debemos declarar nuestras propiedades como privadas, los programadores se comprometen a ofrecer medios para proporcionar acceso a los datos del objeto. Para hacer show debemos implementar lo que llamamos métodos de descriptor de acceso. Estos métodos proporcionan acceso y permiten realizar cambios en los datos del objeto que deben ser públicos.

Agreguemos a la clase Coche estos métodos. Se conocen como getters y setters debido a que permiten que otros objetos en nuestro programa modifiquen las propiedades de nuestro programa. Como tenemos el método repintar no ponemos el setter de Color setColor()

```
public String getColor() {
            return color;
      }
```

```
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;
}
```

Hemos ampliado la nueva interfaz de objetos



A partir de este punto, vamos a ampliar nuestra funcionalidad de clase, agregándola nuevas características.

Agregue dos nuevas propiedades y cambie el constructor de clase:

```
public class Coche {
	private String color;
	private String marca;
	private String modelo;
	private double precio;
	private double coste;



	public static final String TAG = "Objeto Coche";

	public Car (String color, String brand, String model, double precio,
double coste) {

			this.color=color;
			this.marca=marca;
			this.modelo = modelo;
			this.precio=precio;
			this.coste=coste;

	}
```

Dado que no queremos permitir que otros objetos modifiquen o accedan a estas nuevas propiedades, no agregue nuevos captadores y establecedores para estas propiedades. En su lugar, agregaremos los siguientes métodos antes del método toString. Sólo estamos interesados en ofrecer a otros objetos el beneficio obtenido por cada venta de coches.

```
public double beneficios(double iva) {

		return (precio-coste) -impuestos(iva);

	}



private double impuestos( double iva) {

		return (precio-coste)* iva;
	}
```

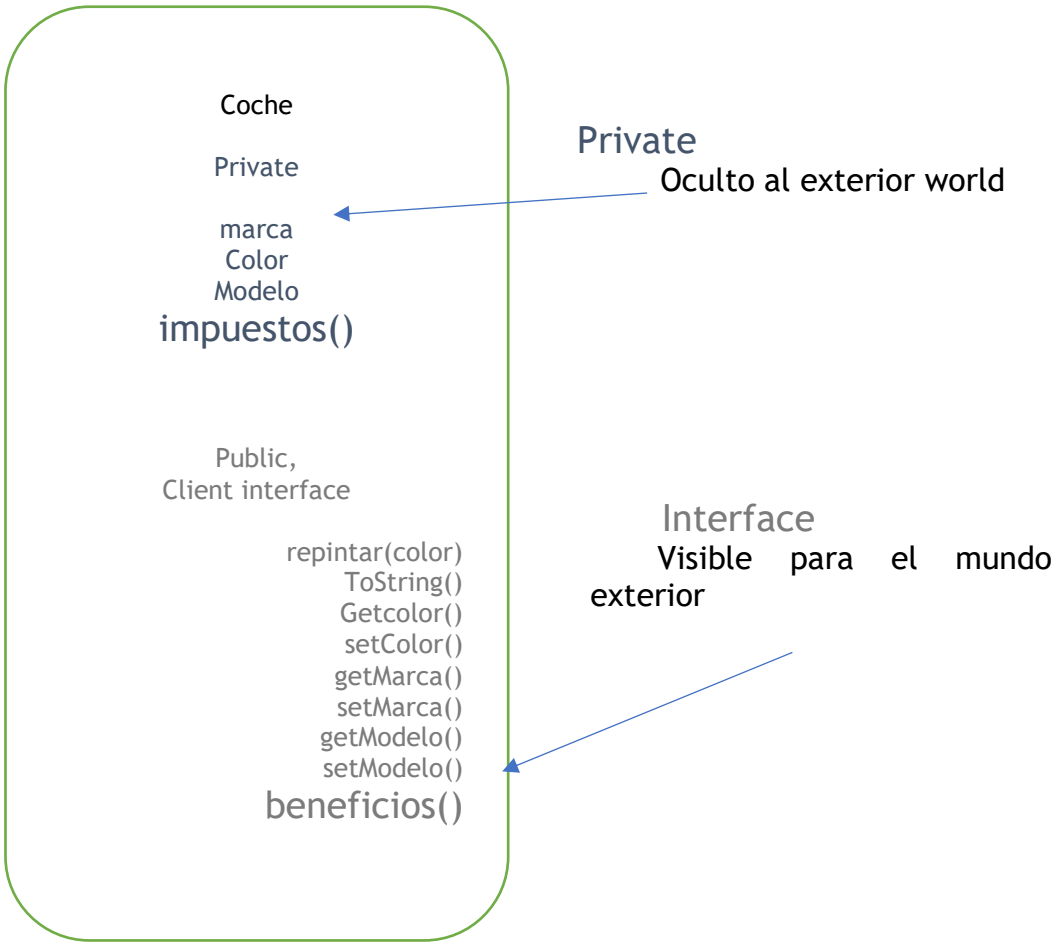
De lo contrario, no queremos explicar cómo calculamos los impuestos. Por un lado, el método de los impuestos está destinado a los cálculos internos. En

consecuencia, lo declaramos privado, siempre que esté dentro de la clase. Al mundo exterior sólo estamos ofreciendo el método de los beneficios.

Por otro lado, el método de ganancias está dirigido a ofrecer un cálculo a otros objetos, ganancia por venta. Los métodos públicos que hemos definido antes proporcionan acceso al estado del objeto. Por el contrario, el método de ganancias ofrece un algoritmo de ganancia simple, un comportamiento.

Estos métodos, cuyo objetivo es proporcionar cálculos y algoritmos internos, se etiquetan como Métodos Privados en contraste con los Métodos De Acceso, que nos dan acceso al estado de la clase o propiedades. Además, podemos añadir a la ecuación, Métodos de Comportamiento,que ofrecen una funcionalidad al mundoexterno.

La nueva definición del objeto



Para verificar las actualizaciones de nuestro programa, agregua este código al final de su programa principal para probar nuevos métodos y verificar los resultados.

```
public static void main(String[] args) {
            // TODO Auto-generated method stub


            Coche miTesla = new Coche ("rojo","Tesla","Model S High
Performace", 50000,30000);

            System.out.println(miTesla);

            Coche miBMWi = new Coche("azul"," BMW","i5",50000,40000);

            System.out.println(miBMWi);

            miBMWi.repintar("blanco");

            System.out.println("MYBMW despues de repintar " + miBMWi);

            System.out.println("color Property access via Method " +
miBMWi.getColor());


            System.out.println("beneficio de BMW despues de venta:" +
miBMWi.beneficios(0.10));
            System.out.println("beneficio  de   Tesla  después  de  venta:"  +
miTesla.beneficios (0.10));
```

La ejecución

```
Coche [color=rojo, marca=Tesla, modelo=Model S High Performace]
Coche [color=azul, marca= BMW, modelo=i5]
MYBMW despues de repintar Coche [color=blanco, marca= BMW, modelo=i5]
color Property access via Method blanco
beneficio de BMW despues de venta:9000.0
beneficio de  Tesla después de venta:18000.0
```

Aunque declaramos variables y métodos en nuestra clase, métodos y propiedades pertenecen a objetos, son parte de cada objeto. Vale la pena señalar que cada objeto tiene su propio valor para el color propertie , Tesla color = rojo, y BMW color = azul. Posteriormente, cada objeto tiene sus propios métodos. El valor devuelto del método de objeto BMW miBMWi.beneficios(0.10) es 9000, mientras que el método de objeto Tesla miTesla.beneficios(0.10)); da como

resultado 18000. Incluso si el algoritmo es el mismo, el método profit depende de las propiedades de los objetos. La clase es sólo una plantilla, los objetos copian el código (propiedades y métodos) siempre y cuando se crean. Se puede inferir que cada vez que no usamos ningún modificador o el modificador public, private o protected, creamos un plano de una propiedad o método para ser uso de objetos.

Para concluir, en una clase se puede distinguir entre estos tipos de métodos:

- Método constructor
- Private y public(Métodos)
- Accesor (Métodos) getters y setters
- Métodos u operaciones de comportamiento

2.2.6 Metodos y clases estáticas.

En las dos secciones anteriores nos hemos tratado de propiedades y métodos. En nuestras clases, definimos propiedades y métodos para que formarán parte de un objeto, una instancia de clase. Eso se traduce en la capacidad de los objetos para almacenar sus datos o mantener su estado. Además, estos mecanismos, métodos, ofrecen una interfaz, un punto de entrada, para recibir mensajes de otros objetos, o para realizar una operación.

Sin embargo, Java, como en otros lenguajes POO, proporciona el mecanismo para crear propiedades y métodos de clase. El modificador static permite a los programadores definir propiedades y métodos que pertenecen a la clase. Las instancias de objeto no las replican, pero tienen acceso a ellas y varios objetos de la misma clase comparten estos métodos y propiedades estáticos. Para demostrarlo, debemos agregar algo de código a nuestra clase

Se nos hacen algunas adiciones a la clase de coches, resaltado en amarillo. Vamos a enumerar e ilustrar los cambios:

1. Hemos introducido una propiedad estática, una propiedad de clase *numeroDeCoches* .

```
private static int numeroDeCoches =0;
```

1. Hemos incluido un método estático que devuelve el número de coches creados, que hacen uso de la propiedad estática numberOfCars. El otro incrementa el número de coches creados por uno En métodos

estáticos sólo podemos utilizar propiedades estáticas

```
public static int numeroDeCochesCreados() {

		return numeroDeCoches;
	}

private static void  incrementaElNumeroDeCoches() {

		numeroDeCoches = numeroDeCoches + 1;

	}
```

1. Hemos modificado el constructor para agregar una unidad a numeroDeCoches cada vez que se crea un automóvil. Esa actualización resulta en mantener una responsabilidad de los automóviles creados en nuestro programa

```
public Coche (String color, String marca, String modelo,double precio, double
coste) {

		this.color=color;
		this.marca=marca;
		this.modelo = modelo;
		this.precio=precio;
		this.coste= coste;


		Coche.incrementaElNumeroDeCoches();
```

```
package fundamentosorientacionaobjetos;
public class Coche {
	private String color;
	private String marca;
	private String modelo;
	private double precio;
	private double coste;
```

```
	private static int numeroDeCoches=0;

	public Coche (String color, String marca, String modelo,double precio,
double coste) {

			this.color=color;
			this.marca=marca;
			this.modelo = modelo;
			this.precio=precio;
			this.coste= coste;
			Coche.incrementaElNumeroDeCoches();

	}

	public String getColor() {
		return color;
	}

	public void setColor(String color) {
		this.color = color;
	}

	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 void repintar(String color) {


		this.color=color;
	}


public double beneficios(double iva) {

		return (precio-coste) -impuestos(iva);

	}



	private double impuestos( double iva) {

		return (precio-coste)* iva;
	}
```

```
	@Override
	public String toString() {
		return "Coche [color=" + color + ", marca=" + marca + ",
modelo=" + modelo + "]";
	}



public static int numeroDeCochesCreados() {

		return numeroDeCoches;
	}

public static void  incrementaElNumeroDeCoches() {

		numeroDeCoches = numeroDeCoches + 1;

	}


}
```

En la memoria, estas propiedades estáticas y métodos se almacenan en un área reservada para la clase Car. Sin embargo, dado que myTesla y myBMW contienen objetos que son miembros de la clase Car, tienen permiso para acceder y utilizar estos componentes estáticos.

Además, dado que `numeroDeCochesCreados()` se ha declarado público, cualquier objeto o clase de su programa puede acceder a ellos. Por el contrario, `incrementaElNumeroDeCoches ()`, declarado como privado, puede ser el uso de los miembros de la clase.

Memoria RAM

.

Memory address		value
200000	Class Coche	
200001	*numerodeCoches*	0
	numeroDeCochesCreados ()	
	incrementaElNumeroDeCoches ()	
40000	myTesla	1000000
40001	myBMW	1200004
..		
..		
..	MiTesla objeto	
1000000	color	red
1000001	brand	Tesla
1000002	model	s
1000003		
1000004	repaint()	
1000005	..	
1000006	..	
1000007	..	
1000008	toString()	
1000009	..	
…		
…	MyBMW object	
1200004	color	blue
1200005	brand	BMW
1200006	model	I5
	…….	

Ahora se puede probar estos cambios en el programa y comprobar que funcionan como estaban dirigidos. Agregue esta línea al final de la función main.

```
System.out.println("Total de coches creados: "+ Coche. numeroDeCochesCreados
());
```

Después de ejecutar el programa y tener que crear dos coches, numeroDeCochesCreados() responder correctamente.

```
Coche [color=rojo, marca=Tesla, modelo=Model S High Performace]
Coche [color=azul, marca= BMW, modelo=i5]
MYBMW despues de repintar Coche [color=blanco, marca= BMW, modelo=i5]
color Property access via Method blanco
beneficio de BMW despues de venta:9000.0
beneficio de  Tesla después de venta:18000.0
```

Si observas la declaración `coche.numeroDeCochesCreados()`, para llamar al método estático `numeroDeCochesCreados` (), primero debe escribir el nombre de la clase, Car.

Trate de llamar desde el coche principal. `numeroDeCochesCreados` (). El compilador solicitará un error. ¿Podría inferir por qué? Extraer una conjutura.

Por último, vamos a revisar su código. Hablábamos de variables constantes en una clase en secciones anteriores. ¿Por qué declaramos esta variable constante con el modificador static? ¿Crees que puedes usarlo en tu programa principal? Suponiendo que puedas, hazlo.

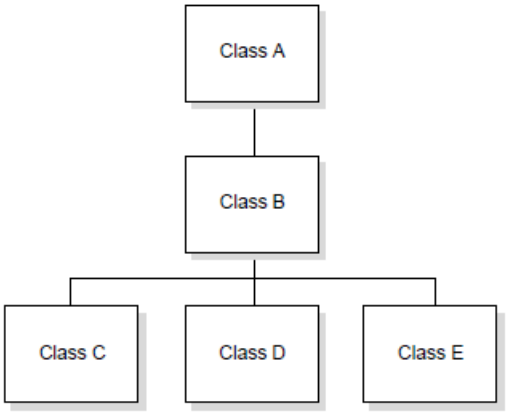
```
public static final String TAG = "Car Object";
```

2.3 Herencia en Java

La herencia es uno de los conceptos más cruciales en la programación orientada a objetos, y tiene un efecto muy directo en la forma de diseñar y escribir las clases Java. La herencia es un mecanismo eficaz que significa que cuando se escribe una clase sólo tiene que especificar cómo esa clase es diferente de alguna otra clase, mientras que también le da acceso dinámico a la información contenida en esas otras clases.

Con la herencia, todas las clases: las que escribe, las de otras bibliotecas de clases que utiliza, y los de las clases de utilidad estándar también se organizan en una jerarquía estricta.

Cada clase tiene una superclase (la clase por encima de ella en la jerarquía), y cada clase puede tener una o más subclases (clases por debajo de esa clase en la jerarquía). Se supone que las clases más abajo en la jerarquía heredan de las clases más arriba en la jerarquía.



Una jerarquía de clases. • Clase A es la superclase de B

• Clase B es una subclase de A

• La Clase B es la superclase de C, D y E

• Las clases C, D y E son subclases de B

Las subclases heredan todos los métodos y variables de sus superclases, es decir, en cualquier clase en particular, si la superclase define el comportamiento que necesita su clase, no tiene que redefinirlo o copiar ese código de alguna otra clase. La clase obtiene automáticamente ese comportamiento de su superclase, esa superclase obtiene el comportamiento de su superclase, y así sucesivamente hasta el final de la jerarquía.

La clase se convierte en una combinación de todas las características de las clases por encima de ella en la jerarquía.En la parte superior de la jerarquía de clases Java se encuentra la clase Object; todas las clases heredan de esta superclase.

Object es la clase más general de la jerarquía; define el comportamiento específico de todos los objetos de la jerarquía de clases Java. Cada clase más abajo en la jerarquía agrega más información y se adapta más a un propósito específico. De esta manera, se puede pensar en una jerarquía de clases como la definición de conceptos muy abstractos en la parte superior de la jerarquía y esas ideas cada vez más concretas cuanto más abajo en la cadena de superclases se va.

La mayoría de las veces cuando se escribe nuevas clases Java, se querrá crear una clase que tenga toda la información que tiene alguna otra clase, además de información adicional. Por ejemplo, es posible que desee una versión de button con su propia etiqueta integrada. Para obtener toda la información de Button, todo lo que tiene que hacer es definir la clase para heredar de Button.

Su clase obtendrá automáticamente todo el comportamiento definido en Button (y en las superclases de Button), por lo que todo lo que tiene que preocuparse son las cosas que hacen que su clase sea diferente de Button. Este mecanismo para definir nuevas clases como las diferencias entre ellas y sus superclases se denomina subclases.

2.4 Jerarquía de clases

Ahora intentaremos crear nuestra propia jerarquía creando nuevas clases que hereden de Coche.

El primero que añadiremos al pack es el SUV.

```
public class SUV extends Coche{

	public SUV(String color, String marca, String modelo, double precio,
double coste) {
		super(color, marca, modelo, precio, coste);

	}

}
```

Para lograr la herencia, debe utilizar la palabra reservada extends seguida del nombre de la clase, `extends Coche`.

Si repasas el Constructor, te darás cuenta de que hay algo nuevo, una llamada a super. Super es una palabra reservada para las clases. Su propósito es llamar al constructor padre. Como resultado, se ejecutará el código de constructor

primario o de superclase. Entonces este código se ejecutará:

```
	this.color=color;
		this.marca=marca;
		this.modelo = modelo;
		this.precio=precio;
		this.coste=coste;

		Coche.incrementaElNumeroDeCoches();
```

Una de las muchas ventajas de la herencia es la reutilización del código. No necesitamos volver a escribir todo el código de superclase. Como SUV hereda de Coche, todo el Código de Coche está a su disposición.

Sigamos creando un nuevo coche SUV. Pero primero, agregaremos una nueva principal, llamada AppInheritance.java.

```
package fundamentosorientacionaobjetos;
public class AppHerencia {


	public static void main(String[] args) {
		// TODO Auto-generated method stub
		SUV miTeslaSUV = new SUV("red","Tesla","Model S High
Performace", 50000,30000);



		System.out.println("Mi nuevo SUV:" + miTeslaSUV.toString());

		System.out.println("Total de coches creados: "+
Coche.numeroDeCochesCreados());


		miTeslaSUV.repintar("blanco");


		System.out.println("Mi nuevo SUV repintado:" +
miTeslaSUV.toString());


	}

}
```

El resultado en ejecución debe ser algo como:

```
Mi nuevo SUV:Coche [color=red, marca=Tesla, modelo=Model S High Performace]
Total de coches creados: 1
Mi nuevo SUV repintado:Coche [color=blanco, marca=Tesla, modelo=Model S High
```

Performace]
"Total de coches creados:1" resulta de myTeslaSUV siendo un SUV, que es un coche ya que hereda de th Car Class. En el constructor SUV, llamamos a super, el constructor de superclase, que cuenta el número de coches creados.

Presta atención ahora a este método llame: myTeslaSUV.toString(). Incluso si este método no está implementado en la clase SUV, pueded usarlo, porque hereda todo el código de superclase.

Protected vs private

Se introducirá aquí el modificador protected. La mejor opción para describir los modificadores protected es mediante el uso. Por lo tanto, debe introducir este código en el constructor de SUV

```
package fundamentosorientacionaobjetos;

public class SUV extends Coche{

	public SUV(String color, String marca, String modelo, double precio,
double coste) {
		super(color, marca, modelo, precio, coste);
		this.color= color;
	}
```

La instrucción resaltada produciría un error del compilador que resulta de la declaración de color de propiedad. Siempre que haya declarado la propiedad color como privada, las subclases heredadas no pueden acceder a ella. Parece ser incómodo que SUV ha heredado el código de clase Car, pero puede acceder directamente a las propiedades. Los modificadores de Java proporcionan niveles de seguridad sobre el acceso al código.

Sin embargo, a veces tendrá que manipular las propiedades de superclase en su subclase. No es el procedimiento más conveniente en OOP, aunque puede cambiar el modificador de propiedad de color a protected para lograrlo.

```
public class Car {
```

```
protected String color;
private String marca;
private String modelo;
private double precio;
private double coste;
```

Comprobar que el código se está compilando

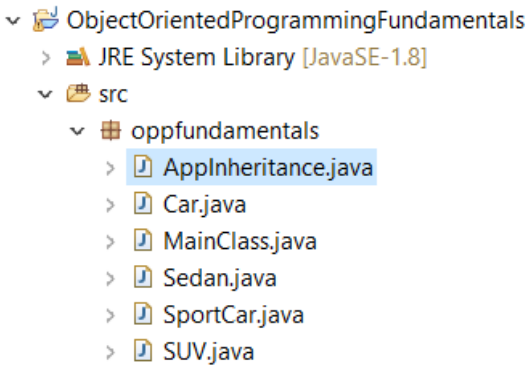
¿Por qué protegido y no público?

Los programadores usan protected en lugar de public, cuando necesitan usar una propiedad de una superclase, pero no desean que sea pública para todas las clases de su modelo de clase. El acceso desde el SUV al color de la propiedad se concede debido a los modificadores protected. Incluso una subclase de SUV, como electric SUV podría tener acceso al color propertie ya que el modificador protected otorga acceso a todas las subclases en la jerarquía, herencia directa e indirecta incluyen. No es una buena práctica, pero a veces es inevitable.

Extensión de la jerarquía de herencia

Es hora de agregar algunas clases más a la jerarquía. Por lo tanto, cree la clase CrossOver, Sedan y SportCar que heredan de Car.

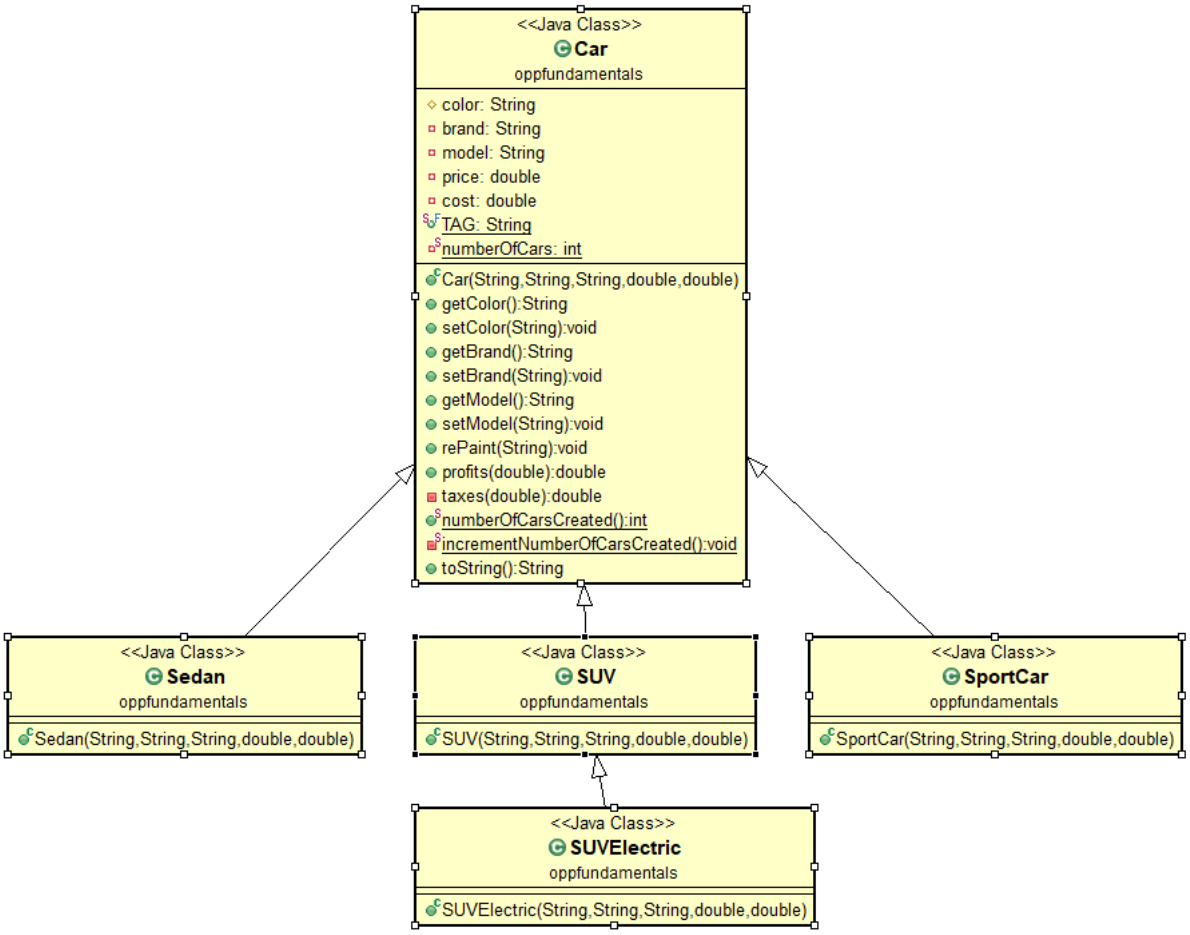
El modelo de clase de proyecto debe ser similar a la siguiente figura:



Recuerda que para hacerlos se extiende desde el coche. Finalmente agregue el SUVElectrico.java a la jerarquía. A diferencia de los otros, SUVElectric no heredará de Coche, sino que se extenderá desde SUV

```
package fundamentosorientacionaobjetos;

public class SUVElectrico extends SUV{

	public SUVElectrico(String color, String marca, String modelo, double
precio, double coste) {
		super(color, marca, modelo, precio, coste);

	}

}
```

Nuestra jerarquía de clases

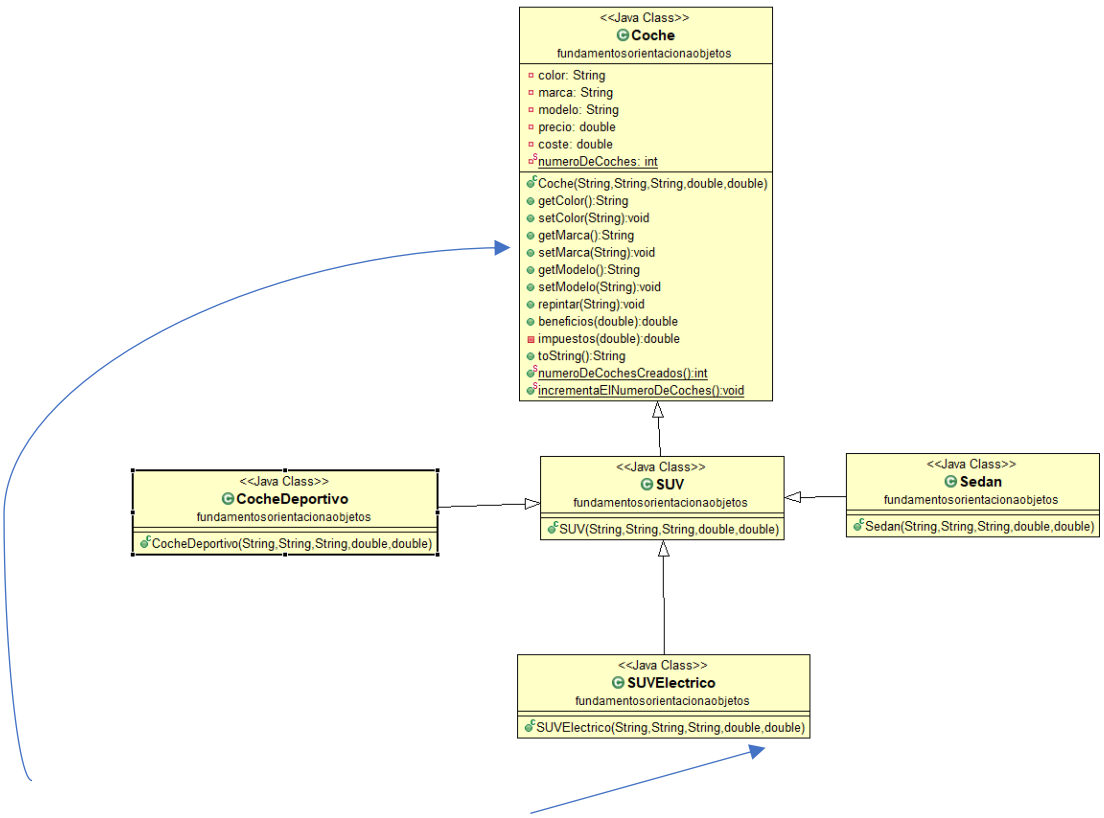


Ten en cuenta que SUVElectrico no hereda directamente de Coche. Sin embargo, a través de SUV heredará el código de la clase Coche. Por otra parte, heredará cualquier adición a SUV que no está incluido en coche. Intenta manipular el la propiedad color en la clase SUVElectrico para que pueda verificar que es plausible.

Presta atención al uso de métodos. Cuando llamamos al método beneficios en SUVElectrico

En el caso de que SUV Electrico llame al método repintar, la llamada se redirige a la definición del método en la clase Car.

```
Car miTeslaSUVElectrico = new SUV("rojo","Tesla","Model S High Performace",
50000,30000);

myTeslaSUVElectric.rePaint("green");
```




```
myTeslaSUVElectric.rePaint("verde");
```

2.4.1 Notas Cornell

Para to**das las líneas de código del ejemplo anterior que estén señaladas con el marcado**r añadir **comentarios explicando que hace cada línea**. Volver a leer los apuntes por si os surgen dudas.

3 Actividad Independiente. Jerarquía educación

Se desea crear una clase EmpleadoEducacion abstracta. Se guardará su nombre, sus apellidos, teléfono, dirección, sueldo, y porcentaje IRPF, horarioPermanencia y funciones.

Métodos abstractos: todos los métodos de acceso más dos métodos abstractos

calculoDeSueldo ()

calculoDeImpuesto ()
función() que devolverá una cadena con la función.
Deseamos también tener varios tipos de empleados:

Conserje: se desea su función será: " realizar labores de atención al publico y mantenimiento". Se desea guardar su horario y posición : planta baja o primera planta.

calculoDeSueldo -> sueldobruto -impuestos
calculoDeImpuesto -> sueldobruto* porcentaje IRPF

Profesor: su función será "Enseñar materias de su especialidad". Se desea guardar especialidad, cuerpo (primaria, secundaria o formación profesional), experiencia. Todas las propiedades tendrán método de acceso.

Métodos:

calculoDeSueldo -> sueldobruto -impuestos + bonusExperiencia
calculoDeImpuesto -> (sueldobruto + bonusExperiencia)* porcentaje IRPF
bonusDeExperiencia -> 20 * experienciaEnAños
Por cada año el bonus de experiencia es 20 euros.

Administrativos su función será "realizar labores administrativas". Se desea guardar clase 3,4 o 5. Su perfil : Contable o legislativo.

calculoDeSueldo -> sueldobruto + categoriaBonus -impuestos
calculoDeImpuesto -> (sueldobruto + bonusExperiencia)* porcentaje IRPF

categoriaBonus - > 3 300 euros, 2 200 euros, 1 100 euros.

3.1 Polimorfismo

Uno de los principales logros o características más avanzadas de la PPO es el Polimorfismo. Significa que objetos similares responden de forma similar al mismo mensaje, que se puede traducir a objetos en una jerarquía que ofrecen el mismo método, por ejemplo, el método toString que presentamos antes. Sin embargo, esta implementación del método podría diferir ligeramente de una clase a una clase.

Por ejemplo, considera un programa de dibujo que permite al usuario dibujar líneas, rectángulos, óvalos, polígonos y curvas en la pantalla. En el programa, cada objeto visible en la pantalla podría ser representado por un objeto de software en el programa. Habría cinco clases de objetos en el programa,unopara cada tipo de objeto visible que se puede dibujar. Todas las líneas pertenecerían a una clase, todos los rectángulos a otra clase, etc. Estas clases están obviamente relacionadas; todas ellas representan"objetos**dibujables".** Por el contrario, la forma en que se dibujan es diferente. El método draw() **debe diferir de una subclase a una subclase,b**aunque todos ellos sean objetos dibujables.

Dos conceptos deben ser introducidos aquí

3.2 *Sobrecarga.*

Una de las técnicas para lograr algo similar al polimorfismo en nuestras clases es aplicar sobrecarga. La idea básica de la sobrecarga es proporcionar varias implementaciones del mismo método. Lo que permite al compilador Java diferencia las versiones es la aridad del método, el tipo y el número y los parámetros que define cada versión del método. Obviamente, no podemos escribir dos versiones del mismo método con la misma aridad en la misma clase.

Los nombres de atributo de una clase pueden ser los mismos que los de otra clase, ya que se tiene acceso a ellos de forma independiente. Un atributo *x* de una clase no tiene necesariamente ninguna relación semántica con otra, ya que tienen ámbitos diferentes y no impide el uso del mismo atributo allí.

Dentro de una construcción de clase Java, los métodos pueden compartir el mismo nombre siempre que pueden distinguirse por:

1. el número de parámetros, o
2. diferentes tipos de parámetros.

Este criterio requiere que un mensaje con parámetros asociados coincida de forma única con la definición de método deseada.

Vamos a incluir esta nueva versión del método de impuestos(doble iva, doble impuestoLocal) a la clase Car. Como puede ver en el siguiente fragmento de código, el mismo método de nombre puede coexistir en la misma clase, tan pronto como muestren una aridad diferente, un tipo diferente de parámetros. En este caso, los impuestos pueden considerar sólo el vanRate, como se describe en la primera versión del método, impuestos(doble iva)

```
private double impuestos( double iva) {

            return (precio-coste)*iva;
      }
```

Sin embargo, nos gustaría que nuestro programa tomara en consideración los impuestos locales, que algunas regiones o estados aplican a las ventas de automóviles. Para ofrecer esta funcionalidad diferente, lo único que tenemos que hacer es agregar esta versión diferente del método de impuestos. Afortunadamente, el compilador de Java es capaz de distinguir sobre estas dos llamadas a métodos, debido a que el número y/o el tipo de parámetros son diferentes.

```
private double impuestos( double iva, double impuestoLocal) {

            return (price-cost)*iva + (price-cost)* impuestoLocal;


}
```

La llamada al método this.impuestos(0.20) haría referencia a la primera versión del método resaltada en azul.

La llamada al método this.impuestos(0.16, 0.5) haría referencia al resaltado en amarillo.

La sobrecarga no es un polimorfismo estricto o puro debido al hecho de que el compilador sabe en tiempo de compilación a qué método estamos haciendo referencia.

En Java, podemos sobrecargar un método tantas veces como necesitemos. Además, también podemos hacerlo para el constructor, como se muestra en el código siguiente. Reemplace la versión anterior del constructor car con estas dos nuevas versiones. Al hacerlo, estamos ofreciendo más posibilidades al programador para crear un objeto de coche.

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

            this.color=color;
            this.marca=marca;
            this.modelo = modelo;

      }


      public  Coche (String color, String marca, String modelo, double
precio, double coste) {

                  this.color=color;
                  this.marca=marca;
                  this.modelo = modelo;
                  this.precio=precio;
                  this.coste=coste;

                  Coche.IncrementaNumeroCochesCreados();

      }
```

Actividad Guiada

También se recomienda en Java, para fines de serialización, añadir una versión sin parámetros de los constructores de clase. Debe empezar a agregar esta versión del constructor a todas las clases. Por ejemplo para la clase Coche.

```
public  Coche () {

}
```

Hagámoslo con todas las clases del modelo, SUV, SUVElectric y así sucesivamente.

3.3 Actividad guiada Overriding

Reemplazar significa tener dos métodos con el mismo nombre de método y parámetros (es decir, *firma de método)*. Uno de los métodos está en la clase primaria y el otro está en la clase secundaria. Reemplazar permite a una clase secundaria proporcionar una implementación específica de un método que ya se proporciona su clase primaria.

Para seguir ilustrando el polimorfismo, estamos para explicar lo primordial. Como resultado, cambiaremos un poco nuestras subclases. En primer lugar, modificaremos el método to string en nuestras subclases.

Agregar este método a SUV

```
@Override
	public String toString() {
		return "SUV [color=" + this.getColor()+   ", marca=" +
this.getMarca() + ", modelo=" + this.getModelo() + "]";;
	}
```

Y éste a Sedán

```
	@Override
	public String toString() {
		return "Sedan [color=" + this.getColor()+   ", marca=" +
this.getMarca() + ", modelo=" + this.getModelo() + "]";
	}
```

Lo primero que se debe tener en cuenta es la etiqueta @Override. Si implementamos un método que ya está implementado en la superclase, necesitamos agregar el @Override. Estamos sobrescribiendo un nuevo código para este método, que es similar al de la superclase, ya que recibe los mismos parámetros. Además, el tipo devuelto es el mismo, String, y también de message es bastante similar. Compárelo con el original:

```
	@Override
	public String toString() {
		return "Car [color=" + color + ", marca=" + marca + ", modelo="
+ modelo + "]";
	}
```

La idea es que cada subclase puede tener un comportamiento diferente aunque todos sean similares. Las respuestas para cada clase podrían ser ligeramente diferentes. Imprimimos cada tipo de coche de manera diferente.

Estos son algunos hechos importantes acerca de la invalidación y la sobrecarga:

1. El tipo de objeto real en tiempo de ejecución, no el tipo de la variable de referencia, determina qué método reemplazado se utiliza en *tiempo de ejecución*. Por el contrario, el tipo de referencia determina qué método sobrecargado se utilizará en *tiempo de compilación*.
2. El polimorfismo se aplica a la invalidación, no a la sobrecarga.
3. La invalidación es un concepto en tiempo de ejecución, mientras que la sobrecarga es un concepto en tiempo de compilación.

Para representar completamente esta diferencia, estamos en el siguiente ejemplo. Agregar este código a su AppHerencia

```
Coche coche1 = new SUVElectrico("rojo","Tesla",
                    "Model S High Performace", 50000,30000);

        Coche coche2 = new Sedan("azul","BMW",
                    "320", 50000,30000);

        coche1.toString();

        coche2.toString();
```

En la siguiente sección, continuaremos representando la idea de polimorfismo a clases abstractas y subclases.

3.3.1 Clases abstractas

Suponiendo que la jerarquía crece y que está más especializada, no necesitaríamos crear instancias de clase Car. Por el contrario, la idea es crear objetos de subclases, como suv y coches deportivos. Como resultado, la clase Car se convierte más en una especie de plano o una plantilla para subclases. La clase abstracta todavía está implementando lo que es común a todas las

subclases, debido al hecho de que evitamos el código repetido en OPP.

Hay dos características interesantes de OPP y herencia:

1. En primer lugar un polimorfismo ya se menciona en la sección anterior.
2. Otra gran característica o ventaja de la herencia es la capacidad de una superclase para obligar a implementar métodos a subclases, con el modificador abstract.

Pongámonos a trabajar:

1. Añade el modificador abstract a tu clase Car

```
public abstract class Coche
```

Las clases abstractas son clases a partir de las cuales no podemos crear objetos. A partir de ahora, el programa en MainClass debe producir un error del compilador si intenta crear un objeto Coche.

```
public abstract class Coche {
```

```
10
11  Cannot instantiate the type Coche  new Coche ("rojo","Tesla","Model S High Performace", 50000,30000);
12
13          System.out.println(miTesla);
14
15          Coche miBMWi = new Coche("azul"," BMW","i5",50000,40000);
```

2. Cambie estas línea a:

```
            Coche miTesla = new SUV ("rojo","Tesla","Model S High
Performace", 50000,30000);

            System.out.println(miTesla);

            Coche miBMWi = new Sedan("azul"," BMW","i5",50000,40000);
```

3. Y también este método abstracto.

```
public abstract String getTipoCoche();
```

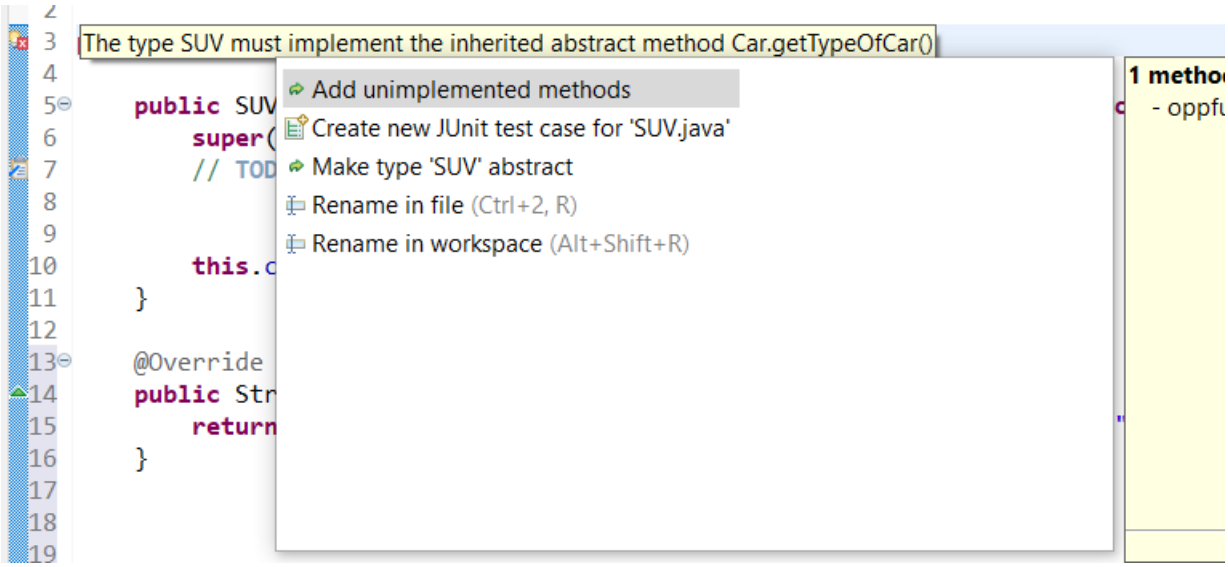
4. Los métodos abstractos son métodos sin cuerpo, sin implementación, solo ofrecemos la firma de la función. Su propósito es ser implementado en subclases que requieren polimorfismo.

Coche.java

```
package fundamentosorientacionaobjetos;
public abstract class  Coche {
	private String color;
	private String marca;
	private String modelo;
	private double precio;
	private double coste;

	private static int numeroDeCoches=0;

public abstract String getTipoCoche();



	public Coche (String color, String marca, String modelo,double precio,
double coste) {

			this.color=color;
			this.marca=marca;
			this.modelo = modelo;
			this.precio=precio;
			this.coste= coste;
			Coche.incrementaElNumeroDeCoches();

	}

	public String getColor() {
		return color;
	}

	public void setColor(String color) {
		this.color = color;
	}

	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 void repintar(String color) {


		this.color=color;
	}


public double beneficios(double iva) {

		return (precio-coste) -impuestos(iva);

	}



	private double impuestos( double iva) {

		return (precio-coste)* iva;
	}


	@Override
	public String toString() {
		return "Coche [color=" + color + ", marca=" + marca + ",
modelo=" + modelo + "]";
	}



public static int numeroDeCochesCreados() {


		return numeroDeCoches;
	}


public static void  incrementaElNumeroDeCoches() {

		numeroDeCoches = numeroDeCoches + 1;

	}



}
```

La cabeza a la clase SUV. También encontrará un error del compilador. No tengas miedo. Eclipse proporcionará una solución. Es necesario agregar los métodos no implementadas. Haga clic en Agregar métodos no implementadas, lo que resulta en la adición del nuevo método abstracto getTipoCoche() a su clase. Una vez agregado el método, debe comprobar que este método tiene un cuerpo. Debido al hecho de que no es abstracto, debe implementarlo.




```
@Override
	public String getTipoCoche() {
		// TODO Auto-generated method stub
		return null;
	}
```

Desafortunadamente, el método devuelve null. Se puede reemplazar esta declaración de devolución con "SUV";. De alguna manera, el uso de literales en la programación no es una buena práctica de codificación. Se sugiere que se agregue una variable constante en su lugar.

```
private static final String  TYPE_SUV="SUV";
```

Estas modificaciones en el programa deben ser similares al siguiente extracto de código:

```
package fundamentosorientacionaobjetos;

public class SUV extends Coche{

	private static final String  TYPE_SUV="SUV";

	public SUV(String color, String marca, String modelo, double precio,
double coste) {
		super(color, marca, modelo, precio, coste);
		// TODO Auto-generated constructor stub
	}

	@Override
```

```
	public String toString() {
		return "SUV [color=" + this.getColor() +   ", marca=" +
this.getMarca() + ", modelo=" + this.getModelo() + "]";
	}

	@Override
	public String getTipoCoche() {
		// TODO Auto-generated method stub
		return TYPE_SUV;
	}


}
```

Al final, la adición de métodos abstractos obliga a las subclases a proporcionarlos e implementarlos. Lo que es notable aquí es que una superclase lidera o impone el comportamiento de las subclases, debido a que si no implementan el método abstracto un error del compilador traería a colación.

3.4 Actividad independiente. Overriding

Siguiendo el último ejemplo, ¿podría hacer lo mismo para todas las clases?

3.5 Interfaces. Actividad guiada

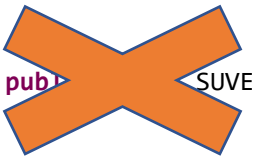
En algunos lenguajes de programación de objetos orientados, las subclases pueden heredar de más de una clase, es decir, C++, que etiquetamos como herencia múltiple. Desafortunadamente, en Java la herencia múltiple no está permitida, y eso da como resultado subclases que se extienden desde una sola superclase. No obstante, Java ofrece un mecanismo similar para simular parcialmente la herencia múltiple, interfaces.

Las interfaces son similares a las clases, pero en oposición a las clases, no ofrecen código para ser heredado por las clases. Las interfaces declaran métodos abstractos que deben ser codificados por las clases que implementan estas interfaces. Una interfaz es una colección de nombres de método, sin definiciones reales, que indican que una clase tiene un conjunto de comportamientos además de los comportamientos que la clase obtiene de sus superclases.

Podemos distinguir claramente en nuestro modelo a tipo de coches eléctricos y coches de combustibles fósiles. Entonces lo primero que deberíamos hacer es separar ambos. Por lo tanto, hacer USV eléctrico para heredar de coche, así.

```
package oppfundamentals;

public class SUVElectrico extends Car{
	private static final String  TYPE_SUV_ELECTRICO="SUVElectrico";
	public SUVElectrico(String color, String marca, String modelo, double
precio, double coste) {
		super(color, marca, modelo, precio, coste);

	}

	@Override
	public String getTypeOfCar() {
		// TODO Auto-generated method stub
		return TYPE_SUV_ELECTRICO;
	}

}
```

En otros lenguajes POO, para mejorar la definición de eléctrico podríamos haber creado una clase llamada Electric y extenderla en SUVElectric. En Java no está permitido.



```
publ        SUVElectrico extends Coche, Electrico { PROHIBIDO
```

Para lograr este modelo deseado, podemos usar interfaces. Podemos definir una Interfaz para Electric, denominada ElectricInterface, ya que nos interesa la tienda, la capacidad de la batería y el consume del coche cada 100 km.

InterfaceElectrico.java

```
package fundamentosorientacionaobjetos;

public interface InterfaceElectrico {


	public abstract double capacidadBateria ();

	public double consumoParaCienKilometros ();


}
```

Una vez declarados los métodos, podemos explicar cómo funcionan las interfaces. Observa el método `capacidadBateria` y compruebe que se ha declarado como abstracto. Por el contrario, para la segunda, no lo hicimos, aunque no es necesario. Todos los métodos declarados pero no definidos en Interfaces son abstractos, yo no necesita usar el modificador abstract. Por lo tanto, `consumoParaCienKilometros()` es igualmente abstracto. No utilizamos el modificador abstract en las interfaces porque no es necesario. Los métodos son abstractos perse,a menos que definamos su cuerpo.

Posteriormente, necesitamos implementar esta interfaz en nuestra clase SUV. Además, es obligatorio añadir esta nueva información pero solo a los coches eléctricos.

En consecuencia, necesitamos:

1. Implementación de la interfaz

```
public class SUVElectrico extends Coche implements InterfaceElectrico
```

1. Añadir dos nuevas propiedades

```
private double capacidadBateria=0;
	private double consumo=0;
```

2. Implementando los métodos que se declararon en la interfaz

```
@Override
	public double capacidadBateria () {
		// TODO Auto-generated method stub
		return capacidadBateria;
	}
```

```
	@Override
	public double consumoParaCienKilometros() {
		// TODO Auto-generated method stub
		return consumo;
	}
```

```
public class SUVElectrico extends Coche implements InterfaceElectrico{
	private static final String  TYPE_SUV_ELECTRICO="SUVElectrico";
	private double capacidadBateria=0;
	private double consumo=0;


	public SUVElectrico(String color, String marca,
			String modelo, double precio, double coste,
			double capacidadBateria, double consumo) {
		super(color, marca, modelo, precio, coste);

		this.capacidadBateria= capacidadBateria;
		this.consumo= consumo;

	}




	public String getTipoCoche() {
		// TODO Auto-generated method stub
		return TYPE_SUV_ELECTRICO;
	}



	@Override
	public double capacidadBateria() {
		// TODO Auto-generated method stub
		return capacidadBateria;
	}



	@Override
	public double consumoParaCienKilometros() {
		// TODO Auto-generated method stub
		return consumo;
	}

}
```

Se supone que cuando las clases implementan un Interface, firman un contrato con esta interfaz. Como resultado de este contrato, las clases deben implementar métodos de interfaz abstractos. Al igual que las clases abstractas, las interfaces imponen un comportamiento a las clases, borrándolas para definir

o implementar métodos. En este punto, los coches eléctricos deben proporcionar información sobre la capacidad de la batería y el consumo.

Podemos seguir detallando aún más nuestro modelo. Imagina que te gustaría gestionar la información sobre asientos de coche, para Berlines y SUV. Luego podemos crear una nueva interfaz para cumplir con su requisito, ISeats. Nos acostumbramos en java a comenzar los nombres de interfaz con I mayúscula, siguiendo una palabra o nombre compuesto que describen la función de interfaz.

```
package fundamentosorientacionaobjetos;

public interface IAsientos {


	public int numeroDeAsientos();

}
```

Comencemos a agregar cambios al SUVElectrico. Mientras que solo podemos extender desde una clase en Java, podemos implementar múltiples interfaces. marcado en amarillo, puede comprobar los cambios para este nuevo comportamiento, IAsientos.

```
package fundamentosorientacionaobjetos;


public class SUVElectrico extends Coche implements InterfaceElectrico,
IAsientos{
	private static final String  TYPE_SUV_ELECTRICO="SUVElectrico";
	private double capacidadBateria=0;
	private double consumo=0;
	private int numAsientos=0;


	public SUVElectrico(String color, String marca,
			String modelo, double precio, double coste,
			double capacidadBateria, double consumo, int numAsientos)
{
		super(color, marca, modelo, precio, coste);

		this.capacidadBateria= capacidadBateria;
		this.consumo= consumo;
		this.numAsientos= numAsientos;

	}
```

```
	public String getTipoCoche() {
		// TODO Auto-generated method stub
		return TYPE_SUV_ELECTRICO;
	}




	@Override
	public double capacidadBateria() {
		// TODO Auto-generated method stub
		return capacidadBateria;
	}




	@Override
	public double consumoParaCienKilometros() {
		// TODO Auto-generated method stub
		return consumo;
	}



	@Override
	public int numeroDeAsientos() {
		// TODO Auto-generated method stub
		return numAsientos;
	}

@Override
	public String toString() {
		return "SUV [color=" + this.getColor() +   ", brand=" +
this.getMarca() +
				", model=" + this.getModelo() + ",
capacidadBateria=" +
				capacidadBateria +  ", consumo=" +  consumo +
				", numAsientos=" + numAsientos + "]";
	}

}
```

Por último, podemos modificar nuestra clase AppHerencia para probar los cambios de modelo

```
package fundamentosorientacionaobjetos;


public class AppHerencia {
```

```
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		SUV miTeslaSUV = new SUV("red","Tesla","Model S High
Performace", 50000,30000);



		System.out.println("Mi nuevo SUV:" + miTeslaSUV.toString());

		System.out.println("Total de coches creados: "+
Coche.numeroDeCochesCreados());


		miTeslaSUV.repintar("blanco");


		System.out.println("Mi nuevo SUV repintado:" +
miTeslaSUV.toString());

		SUVElectrico coche1 = new SUVElectrico("rojo","Tesla",
				"Model S High Performace", 50000,30000,20,7,5);

		Sedan coche2 = new Sedan("azul","BMW",
				"320", 50000,30000);

		System.out.println("Mi nuevo SUV:" + coche1.toString());

		System.out.println("Mi nuevo Sedan:" + coche2.toString());

	}

}
```

El resultado de la ejecución del programa esperaba:

```
Mi nuevo SUV:SUV [color=red, marca=Tesla, modelo=Model S High Performace]
Total de coches creados: 1
Mi nuevo SUV repintado:SUV [color=blanco, marca=Tesla, modelo=Model S High
Performace]
Mi nuevo SUV:SUV [color=rojo, brand=Tesla, model=Model S High Performace,
capacidadBateria=20.0, consumo=7.0, numAsientos=5]
Mi nuevo Sedan:Sedan [color=azul, marca=BMW, modelo=320]
```

3.6 Actividad de refuerzo. Modelo de clases

1. Incluir IAsiento en la clase SUV y Sedan
2. Cree una nueva clase SedanElectrico, agregue el contrato IAsiento y InterfaceElectrico a esta clase.
3. Cree una nueva clase SedanElectrico, incluidas las interfaces mencionadas anteriormente.

4 Clases wrap para tipos primitivos. Conversión de tipos

Debido a que Java es un lenguaje de programación orientado a objetos, Java ofrece un futuro para envolver todos los tipos primitivos con clases. El propósito es poder trabajar en programas Java con clases. Estas clases wrap son similares a la clase String, ya que aunque son clases de alguna manera se comportan como tipos primitivos. Por lo tanto, una clase contenedora primitiva es una clase contenedora que encapsula, oculta o ajusta tipos de datos de los ocho tipos de datos primitivos, de modo que estos se pueden usar para crear objetos instanciados con métodos en otra clase o en otras clases. Las clases contenedoras primitivas se encuentran en la API de Java.

Las ocho clases primitivas básicas son las siguientes:

- Lang.Boolean.
- lang.Character.
- lang.Byte.
- lang.Short.
- lang.Integer.
- lang.Long.
- lang.Float.
- lang.Double.

Por ejemplo, tenemos una clase para envolver e incrustar el tipo primitivo int, llamado Integer. Un objeto de tipo Integer contiene un único atributo o propiedad cuyo tipo es int. Además, esta clase proporciona varios métodos para convertir un int en un String y un String en un int, así como otras constantes y métodos útiles cuando se trata de un int.

La clase Integer, por ejemplo, está definida de la siguiente manera:

```
Class Integer {

private int value;

public int intValue() {
 return value;
}
… mas métodos

}
```

En términos generales, estas clases primitivas similares, String incluida en la

opinión del autor, proporcionan métodos para realizar la conversión de tipos. Además, el constructor podría recibir uno de los dos parámetros de un tipo diferente. Recuerde la función de sobrecarga en Java. Permitió crear una versión diferente del mismo método.

Primitive type	Wrapper class	Constructor arguments (old versions of Java) valueOf arguments (from Java 13)
`byte`	`Byte`	`byte` or `String`
`short`	`Short`	`short` or `String`
`int`	`Integer`	`int` or `String`
`long`	`Long`	`long` or `String`
`float`	`Float`	`float`, `double` or `String`
`double`	`Double`	`double` or `String`
`char`	`Character`	`char`
`boolean`	`Boolean`	`boolean` or `String`

Se puede crear objetos de esta clase mediante el constructor, y también asignando literales, como se hizo con la clase String. Es más común crear estas clases con la instrucción asignar. De todos modos, siempre que se desee se puede realizar la conversión de tipos, puede usar el constructor del mismo modo, en versiones anteriores de Java. A partir de Java 13, los constructores para estas clases están en desuso, en otras palabras, se recomienda que no se

utilicen. Cuando Java u otros desarrolladores etiquetan en sus proyectos un método, campo o tipo como obsoleto, hay ciertos constructores, campos, tipos o métodos que ya no quieren que la gente use.

En consecuencia, para llevar a cabo la conversión de tipos se pueden utilizar algunos de los métodos proporcionados por estas clases, como valueOf. es decir, entero. valueOf("5"). Las clases de envoltura primitivas no son lo mismo que los tipos primitivos. Mientras que las variables, por ejemplo, se pueden declarar en Java como tipos de datos dobles, cortos, int, etc., las clases contenedoras primitivas crean objetos y métodos instanciados que heredan pero ocultan los tipos de datos primitivos, no como las variables a las que se asignan los valores de tipo de datos.

Pero primero intentaremos ilustrar cómo crear estos objetos wrap en las versiones modernas de java.

1. El constructor Integer está en desuso, no se recomienda su uso.

```
Integer iObj = new Integer("3");


Float flObj = new Float(dObje);
```

2. Podemos asignar literales a esta clase de variables, lo que da como resultado la creación del objeto automática.

```
Long lObje = 5L;
Double dObj =5.9;

Boolean blObj= true;
```

3. Podemos utilizar métodos de conversión.

```
Integer intVar =Integer.valueOf("6");

Byte bObj = Byte.valueOf("23");
```

4. Podemos obtener el valor primitivo original.

```
int iPrim = iObj.intValue();
```

5. Del mismo modo, puede asignar directamente estos objetos a variables de tipo primitivas. Como mencionamos anteriormente, estas clases solo tienen un atributo para almacenar el tipo primitivo. Por lo tanto, el compilador permite a los programadores trabajar con ellos como tipos primitivos.Se puede encontrar los posibles tipos de argumentos que se deben usar para valueOf en la tabla anterior

```
int iPrim = iObj;
long lPrim= lObje;

double dPrim =  dObj;
```

Finalmente, String ofrece un método para la conversión de String, valueOf, que puede recibir todos los tipos primitivos como argumentos a convertir. Este método está sobrecargado en la clase String, recibiendo diferentes parámetros. Preste atención al siguiente fragmento de código, de modo que estemos convirtiendo de tipos primitivos a String.

```
String convPrim = " int a String " + String.valueOf(iPrim) + "" +

                  " long a String " +       String.valueOf(lPrim) + "." +
                  " double a String " +     String.valueOf(dPrim) + "." +
                  " float a String " +      String.valueOf(flPrim) + "." +
                  " boolean a String " +    String.valueOf(blPrim) + "." ;

System.out.println("Conversión de tipos primitivos a String" + convPrim);
```

Asimismo, String.valueof() puede recibir todos los tipos de wrapper como argumentos a convertir, y eso resulta de la estructura de Wrappers, dado que almacenan el tipo primitivo en un atributo. Se ilustra en la siguiente secuencia de código:

```
            String convObj = " Integer a String " + String.valueOf(iObj) +
"." +

                  " Long a String " +       String.valueOf(lObje) + "." +
                  " Double a String " +     String.valueOf(dObj) + "." +
                  " Float a String " +      String.valueOf(flObj) + "." +
                  " Boolean a String " +    String.valueOf(blObj) + "."  ;


System.out.println("Conversión de clases wrapper primitivas a String " +
convObj);
```

En las siguientes unidades, veremos cómo java trata con los tipos primitivos y ofrecemos algunas clases de adaptador para usarlas en modelos Java más complejos y clases en la API de Java. Tampoco para decir, que la tendencia en la programación Java es usar clases wrapper en lugar de tipos primitivos.

5 Clases e interfaces con tipos genéricos en Java

En **esencia, el término genéricos significa tipos parametrizados.** Los **tipos parametrizados son importantes porque permiten crear clases, interfaces y métodos en los que se** especifica como parámetro el tipo de datos en los que *operan*. Una **clase, interfaz o método que funciona con un tipo de parámetro** se denomina **genérico,** como una clase o método genérico.

La principal ventaja del **código genérico es que funcionará automáticamente con el tipo de datos pasado a su parámetro de tipo.** Muchos algoritmos **son lógicamente iguales ,**independientemente del tipo de datos a los que se apliquen. Por ejemplo, un Quicksort (algoritmo de ordenación) es el mismo si está ordenando elementos de tipo Integer, String, Object o Thread. **Con los genéricos, puede definir un algoritmo una vez,** independientemente **de cualquier tipo específico** de datos, y luego aplicar ese algoritmo a una amplia variedad de tipos de datos sin ningún esfuerzo adicional.

A partir de Java SE 5, podemos crear clases cuyo tipo se indique en tiempo de compilación. Mira el siguiente ejemplo. Definimos una clase genérica que recibe un tipo T.class Generic<T>. El operador <> se conoce como operador Diamond. Nos permiten crear clases que manejan diferentes tipos.

Si nos fijamos en el ejemplo, tenemos crear a objeto de nuestra clase genérica. Uno para Doble, otro para Entero. Nos permite definir el tipo que utilizará nuestra clase genérica desde el momento de la compilación. Vamos a desglosar la siguiente clase:

Hemos declarado una clase genérica de tipo T genérico. La clase tiene dos propiedades Tipo T, operando1 y operando2.

```
public class EjemploClaseGenerica<T> {

	private T operando1;
	private T operando2;
```

Realizamos algunas operaciones a través de métodos para estos operandos genéricos, sin importar cuál sea el tipo. En el primero de ellos la clase Type se devuelve como String, mientras que en el segundo se comparan ambos operandos.

```
public String muestraTipo () {

		return operando1.getClass().toString();

	}
```

```
public boolean comparacion() {

		return  operando1.equals(operando2);

	}
```

En el tercero, comparamos operand1 con un valor pasado como parámetro.

```
public boolean comparationExternal(T op3) {


		return operand1.equals(op3);
	}
```

Para crear objetos para esta clase, necesitamos pasar el tipo como un paremeter usando el operador diamante. En el primer objeto, pasamos Double como parámetro. Por lo tanto, en la llamada al constructor necesitamos pasar valores de tipo doble como parámetro.

```
	EjemploClaseGenerica<Double> dobleGenerico = new
EjemploClaseGenerica<Double> (Double.valueOf(1),Double.valueOf(2));
```

Además, para cada método de la clase que recibe un parámetro T, ya que hemos definido la T parametrizada como Double, debemos pasar un Double como parámetro. Por ejemplo, puede echar un vistazo al método comparationExternal. Recibe un parámetro op3 tipado T.

```
public boolean comparacionExterna(T op3) {


		return operando1.equals(op3);
	}
```

Cuando llamamos a este método desde el objeto genericDouble, que T es Double en la creación del objeto, pasamos como parámetro un parámetro Double `Double param =1.0;`.

```
	System.out.println(" Operando 1 es el parámetro. La respuesta es " +
dobleGenerico.comparacionExterna(param));
```

Podríamos hacer lo mismo con los getters y setters. Intente introducir esta línea en el código y verifique si funciona.

```
        dobleGenerico.setOperando2(param);
```

EjemploClaseGenerica.java

```
package genericas;

public class EjemploClaseGenerica<T> {

        private T operando1;
        private T operando2;

        public EjemploClaseGenerica(T operando1, T operando2) {


                this.operando1=operando1;
                this.operando2=operando2;


        }

public T getOperando1() {
                return operando1;
        }




        public void setOperando1(T operando1) {
                this.operando1 = operando1;
        }




        public T getOperando2() {
                return operando2;
        }
```

```
	public void setOperando2(T operando2) {
		this.operando2 = operando2;
	}


public String muestraTipo () {

		return operando1.getClass().toString();

	}

	public boolean comparacion() {

		return  operando1.equals(operando2);

	}
	public boolean comparacionExterna(T op3) {


		return operando1.equals(op3);
	}


	@Override
	public String toString() {
		return "EjemploClaseGenerica [operando1=" + operando1 + ",
operando2=" + operando2 + "]";
	}







	public static void main(String[] args) {

		Double param =1.0;

		EjemploClaseGenerica<Double> dobleGenerico = new
EjemploClaseGenerica<Double> (Double.valueOf(1),Double.valueOf(2));


		System.out.println("El tipo de mi clase genérica es " +
dobleGenerico.muestraTipo());


		System.out.println("¿Son iguales los dos operandos? La respuesta
es " + dobleGenerico.comparacion());

	System.out.println(" Operando 1 es el parámetro. La respuesta es " +
dobleGenerico.comparacionExterna(param));

EjemploClaseGenerica<Integer> integerGenerico = new
EjemploClaseGenerica<Integer> (Integer.valueOf(1),Integer.valueOf(2));
```

```
            System.out.println("El tipo de mi clase genérica es " +
integerGenerico.muestraTipo());


            System.out.println("¿Son los dos operandos iguales? La respuesta
es " + integerGenerico.comparacion());




      }

}
```

Podemos extraer varias conclusiones de este ejemplo. La principal podría ser que las clases genéricas permiten crear clases que administran múltiples tipos, lo cual es una herramienta poderosa. Esta técnica está destinada a clases wrap y clases de colección. Las clases de colección son clases que gestionan una colección de objetos que estudiaremos en este curso.

Las clases genéricas nos permiten restringir el tipo T. Podemos hacer que el tipo T extienda una clase o implemente una Interfaz. Permítanme ilustrar la última declaración. Para ello, puede cambiar la declaración de clase en el ejemplo anterior.

```
public class EjemploClaseGenerica<T extends Number>
```

Number es la clase principal de clases numéricas contenedoras en Java, como Integer, Double, Long, etc. Siempre que T se extienda desde Número, estamos forzando a T a ser de Tipo de Number o una de sus subclases, Double, Long, etc. Este es un medio proporcionado por Java para restringir el Tipo T a un ámbito más concreto, en este caso Tipos numéricos Java. Teniendo en cuenta esta declaración, no podemos crear un objeto String con tipo de esta clase, como el nuevo EjemploClaseGenerica<String>.

Practica

Intenta sobreescribir el método equals para la clase genérica anterior. El método equals devolvería true si operando1, y el operando 2 son iguales para ambos objetos.

5.1 Interfaces genéricos

Las interfaces genéricas siguen los mismos principios de las clases genéricas. Sin embargo, son aún más útiles en Java por lo que solo declaramos métodos abstractos en Java, y podemos implementar cuando conocemos el tipo T que hemos pasado como parámetro para crear objetos de esta clase.

Abordemos el código de InterfacesGenericos como ejemplo. Hemos definido la interfaz GenericInterfaceExample<T> con genérico Tipo T

```
package genericas;

public interface GenericInterfaceExample<T> {

	T suma(T op1, T op2);
	T resta(T op1, T op2);
	T producto(T op1, T op2);
	T division(T op1, T op2);


}
```

Podemos hacer múltiples implementaciones de esta interfaz con diferentes tipos. Para ilustrar esto, considerad las siguientes dos clases:

La clase ClassDouble implementa la interfaz pasando como parámetro Double for T. Ahora, las implementaciones de métodos de la interfaz reciben Double como parámetros. Podemos hacer una implementación concreta de estos métodos basados en un tipo específico Doble. La respuesta para division(Double.valueOf(3), Double.valueOf(2)) debe ser 1.5.

```
package genericas;

public class ClassDouble implements GenericInterfaceExample<Double>{

	@Override
	public Double suma(Double op1, Double op2) {
		// TODO Auto-generated method stub
		return op1+op2;
	}

	@Override
	public Double resta(Double op1, Double op2) {
		// TODO Auto-generated method stub
		return op1-op2;
	}

	@Override
	public Double producto(Double op1, Double op2) {
		// TODO Auto-generated method stub
		return op1*op2;
	}
```

```
	@Override
	public Double division(Double op1, Double op2) {
		// TODO Auto-generated method stub
		return op1/op2;
	}

}
```

Ahora observa la siguiente implementación. Es similar pero esta vez T se escribe Long. El resultado de llamar a division(Long.valueOf(3), Long.valueOf(2)) sería 1, ya que nuestra implementación se basa en un tipo entero.

```
package genericas;

public class ClassLong implements GenericInterfaceExample<Long> {

	@Override
	public Long suma(Long op1, Long op2) {
		// TODO Auto-generated method stub
		return op1+op2;
	}

	@Override
	public Long resta(Long op1, Long op2) {
		// TODO Auto-generated method stub
		return op1-op2;
	}

	@Override
	public Long producto(Long op1, Long op2) {
		// TODO Auto-generated method stub
		return op1*op2;
	}

	@Override
	public Long division(Long op1, Long op2) {
		// TODO Auto-generated method stub
		return op1/op2;
	}


}
```

Resumiendo, los tipos genéricos llevan a nuestro código a simular comportamientos similares para tipos similares, aunque mantienen sus propias características. La división para los tipos enteros es ligeramente diferente a la división para los dobles.

6 Bibliografía y referencias web

Referencias web

Tutoriales de Java arquitectura Java

https://www.arquitecturajava.com/

Tutoriales Java geeksforgeeks

https://www.geeksforgeeks.org/

Tutoriales Java Baeldung

https://www.baeldung.com/

Bibliografía

Programación, Alfonso Jímenez Pérez, Francisco Manuel Pérez Montes, Paraninfo, 1ª edición, 2021

Entornos de Desarrollo, Maria Jesus Ramos Martín, Garceta 2ª Edición, 2020

PROGRAMACIÓN EN LENGUAJES ESTRUCTURADOS, Enrique Quero Catalinas y José López Herranz, Paraninfo, 1997.