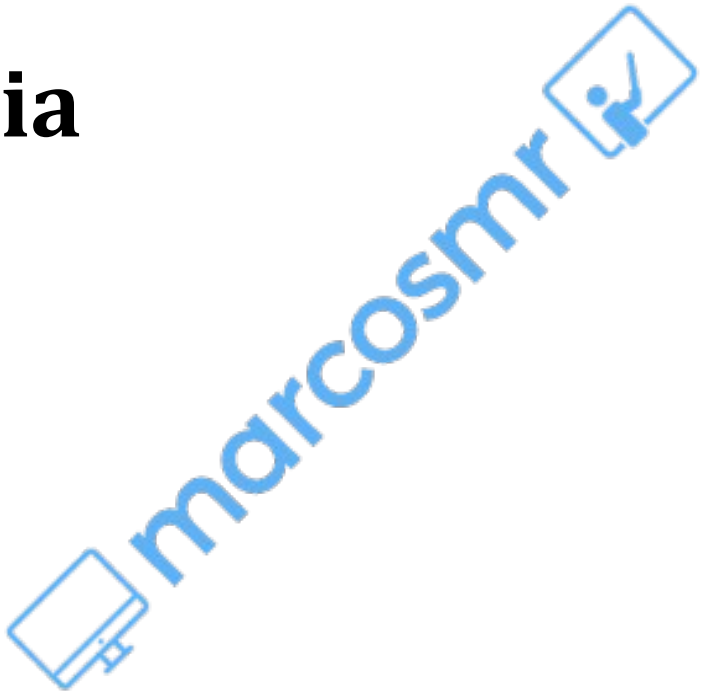
Herencia



ÍNDICE

1. Introducción

Podemos construir una clase a partir de otra mediante el mecanismo de la **herencia**. Para indicar que una clase deriva/hereda de otra se utiliza la palabra **`extends`**. Por ejemplo:

```
public class CirculoGrafico extends Circulo {
	//...
}
```

Cuando una clase deriva de otra, hereda todas sus variables y métodos. Estos métodos y atributos pueden ser redefinidos (*overridde*) en la clase derivada, que puede también definir o añadir nuevos atributos y métodos. Esto proporciona una de las ventajas principales de la Programación Orientada a Objetos (POO): la reutilización de código previamente desarrollado ya que permite a una clase más específica incorporar la estructura y comportamiento de una clase más general.

Java permite múltiples niveles de herencia, pero no permite que una clase derive de varias (**no es posible la herencia múltiple**). Se pueden crear tantas clases derivadas de una misma clase como se quiera.

Todas las clases de Java creadas por el programador tienen una superclase (clase padre). Cuando no se indica explícitamente una superclase con la palabra `extends`, la clase deriva de `java.lang.Object`, que es la clase raíz de toda la jerarquía de clases de Java.

Lo anterior implica que todas las clases tienen algunos métodos que han heredado de `Object`. No se debe confundir la herencia con la composición de clases (que es que una clase contenga un objeto de otra clase como variable miembro o atributo).

2. Ejemplo simple de herencia

Para comprender mejor la potencia y beneficios del uso de la herencia vamos a ver un ejemplo en el que vamos a trabajar con vehículos. Podemos suponer que lo que define un vehículo es: velocidad máxima, marca y modelo. De esta forma, podemos definir la siguiente clase:

```
public class Vehiculo {
	private double velocidadMaxima;
	private String marca;
	private String modelo;

	public Vehiculo(double velocidadMaxima, String marca, String modelo) {
		this.velocidadMaxima = velocidadMaxima;
		this.marca = marca;
		this.modelo = modelo;
	}

	public double getVelocidadMaxima() {
		return velocidadMaxima;
	}
	public void setVelocidadMaxima(double velocidadMaxima) {
		this.velocidadMaxima = velocidadMaxima;
	}
	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;
	}
}
```

Supongamos que queremos también modelar la clase `Coche` la cual además de velocidad máxima, marca y modelo necesita almacenar si es descapotable o no. Podemos implementar dicha clase de la siguiente forma. Como veremos más adelante, con la sentencia `super()` conseguimos llamar al constructor de la clase padre con los parámetros adecuados.

```
public class Coche extends Vehiculo {
	private boolean descapotable;

	public Coche(double velocidadMaxima, String marca, String modelo, boolean descapotable)
	{
		super(velocidadMaxima, marca, modelo);
		this.descapotable = descapotable;
	}

	public boolean isDescapotable() {
		return descapotable;
	}

	public void setDescapotable(boolean descapotable) {
		this.descapotable = descapotable;
	}
}
```

También es posible que nos interese o necesitemos tener una clase `Motocicleta` que, además de tener los atributos y comportamientos de todo `Vehiculo`, tenga un atributo de cilindrada. Podemos definir dicha clase de la siguiente forma:

```
public class Motocicleta extends Vehiculo {
	private int cilindrada;

	public Motocicleta(double velocidadMaxima, String marca, String modelo, int cilindrada)
	{
		super(velocidadMaxima, marca, modelo);
		this.cilindrada = cilindrada;
	}

	public int getCilindrada() {
		return cilindrada;
	}

	public void setCilindrada(int cilindrada) {
		this.cilindrada = cilindrada;
	}
}
```

3. Clase Object

Como acabamos de ver, la clase `Object` es la raíz de toda la jerarquía de clases de Java. Todas las clases de Java derivan de `Object`. La clase `Object` tiene métodos interesantes para cualquier objeto que son heredados por cualquier clase. Entre ellos, vamos a poner el foco en los siguientes:

- **`equals()`**. Indica si dos objetos son o no iguales. Devuelve true si son iguales, tanto si son referencias al mismo objeto como si son objetos distintos con iguales valores de las variables miembro o atributos.
- **`toString()`**. Devuelve una cadena de texto (String) que contiene una representación del objeto como cadena de caracteres. Muy útil para imprimirlo o exportarlo.
- **`getClass()`**. Devuelve un objeto de la clase Class, al cual se le pueden aplicar métodos para determinar el nombre de la clase, su superclase, las interfaces implementadas, etcétera. Se puede crear un objeto de la misma clase que otro sin saber de qué clase es.

4. Constructores en clases derivadas

Sabemos que un constructor de una clase puede llamar por medio de la palabra **`this`** a otro constructor definido previamente en la misma clase. En este contexto, la palabra **`this`** únicamente puede aparecer en la primera sentencia de un constructor.

De forma análoga el constructor de una clase derivada puede llamar al constructor de su clase padre por medio de la palabra **`super`**`()`, seguida entre paréntesis de los argumentos apropiados para uno de los constructores de la superclase. De esta forma, un constructor

solamente tiene que inicializar directamente las variables no heredadas. Hemos visto dos ejemplos en el constructor de la clase Coche y de la clase Motocicleta.

La llamada al constructor de la clase padre debe ser la primera sentencia del constructor, excepto si se llama a otro constructor de la misma clase con `this()`. Si el programador no la incluye, Java incluye automáticamente una llamada al constructor por defecto de la superclase, `super()`.

Esta llamada en cadena a los constructores de las superclases llega hasta el origen de la jerarquía de clases, esto es, al constructor de `Object`.

Como ya sabemos, si el programador no prepara un constructor por defecto, el compilador crea uno, inicializando las variables de los tipos primitivos a sus valores por defecto y los Strings y demás referencias a objetos a `null`. Antes, incluirá una llamada al constructor de la superclase → `super(...)`

5. Modificadores de acceso

Conviene refrescar los conceptos sobre los modificadores de acceso. Tanto `public` como `private` ya los tenemos muy interiorizados. Sin embargo, en este momento, es importante que asentemos el conocimiento del modificador de acceso `protected`.

En diferentes lenguajes, Java entre ellos, se usa un nivel de acceso intermedio (entre público y privado) que se denomina como "acceso protegido" y se expresa con la palabra clave `protected`, que significa que las subclases sí pueden tener acceso al campo o método decorado con dicho modificador de acceso. Este modificador de acceso suele usarse cuando se trabaja con herencia.

Desde un objeto de una subclase podremos acceder o invocar un campo o método declarado como `protected`, pero no podemos acceder o invocar a campos o métodos privados de una superclase.

Java admite una variante más en cuanto a modificadores de acceso: la omisión del mismo (no declarar ninguno de los modificadores *public*, *private* o *protected*). El comportamiento, en dicho caso, es que se permite el acceso a las clases que pertenezcan al mismo paquete.

En la siguiente tabla puedes comparar los efectos de usar uno u otro tipo de declaración en cuanto a visibilidad de los campos o métodos:

MODIFICADOR	CLASE	PAQUETE	SUBCLASE	CUALQUIERA
`public`	✔	✔	✔	✔
`protected`	✔	✔	✔	X
Sin especificar	✔	✔	X	X
`private`	✔	X	X	X

6. Redefinición de métodos

6.1 Concepto

Una clase hija puede redefinir (volver a definir) cualquiera de los métodos heredados de su superclase o clase padre siempre y cuando no estén decorados con la palabra reservada `final`. El nuevo método sustituye al heredado para todos los efectos en la clase que lo ha redefinido.

Supongamos que ampliamos nuestra clase `Vehiculo` con un método que nos informe del tipo de vehículo que es:

```
public class Vehiculo {
	// ...
	public String getInformacion() {
		return "Soy un vehículo de la marca " + getMarca();
	}
	// ...
}
```

Dicho método lo podemos redefinir en la clase hija `Coche` de la siguiente forma:

```
public class Coche extends Vehiculo{
	// …
	@Override
	public String getInformacion() {
		return "Soy un coche con una velocidad máxima de " + getVelocidadMaxima() + " km/h";
	}
	// ...
}
```

En este caso, cuando desde un objeto de tipo `Coche` llamemos al método *getInformacion()*, este será el método que se ejecutará (el que muestra la velocidad máxima).

6.2 Acceso a los métodos de la superclase

Los métodos de la superclase que han sido redefinidos pueden ser todavía accedidos por medio de la palabra `super` desde los métodos de la clase derivada, aunque con este sistema únicamente se puede subir un nivel en la jerarquía de clases (acceder a los métodos del padre).

Imaginemos que, en la clase `Motocicleta`, deseamos redefinir el método *getInformacion()* pero también queremos que se ejecute el método *getInformacion()* de la clase padre (la de `Vehiculo`). Podemos hacerlo de la siguiente forma:

```java
public class Motocicleta extends Vehiculo {
	// …
	@Override
	public String getInformacion() {
	  return super.getInformacion() + ". En concreto, una motocicleta de cilindrada " +
	         getCilindrada() + " cc";
	}
	// ...
}
```

Si ejecutamos el siguiente código, la salida por pantalla es la que se muestra a continuación:

```java
	Vehiculo v = new Vehiculo(30, "Xiaomi", "Scooter 4 Pro");
	Coche c = new Coche(180, "Dacia", "Duster", false);
	Motocicleta m = new Motocicleta(168, "Suzuki", "GSX", 1000);

	System.out.println(v.getInformacion());
	System.out.println(c.getInformacion());
	System.out.println(m.getInformacion());
```

```
Soy un vehículo de la marca Xiaomi
Soy un coche con una velocidad máxima de 180.0 km/h
Soy un vehículo de la marca Suzuki. En concreto, una motocicleta de cilindrada 1000 cc
```

6.3 Otras cuestiones de interés

Los métodos redefinidos pueden ampliar los derechos de acceso de la superclase (por ejemplo ser *public*, en vez de *protected* o *private*), pero nunca restringirlos.

Otra consideración importante es que los métodos de clase o `static` no pueden ser redefinidos en las clases derivadas.

7. Conversión de tipos

En muchas ocasiones, hay que transformar una variable de un tipo a otro, por ejemplo de `int` a `double`, o de `float` a `long`. Con los objetos, instancias de una determinada clase, también podemos realizar operaciones de conversión siempre que dichas clases estén relacionadas mediante herencia.

7.1 Implícito

El siguiente código permite convertir un `Coche` en `Vehiculo`, es decir, de una clase hija a una clase padre.

```
Vehiculo vehiculo = new Coche(100, "Opel", "Kadett", false);
```

En este momento, *vehiculo* es un `Coche` pero más adelante se le puede asignar una `Motocicleta`. Por ejemplo:

```
vehiculo = new Motocicleta(168, "Suzuki", "GSX", 1000);
```

Para entenderlo mejor vamos a ver un ejemplo en el que reutilizamos el objeto vehiculo y la información que saca por pantalla:

```
Vehiculo vehiculo = new Coche(100, "Opel", "Kadett", false);
System.out.println(vehiculo.getInformacion());

vehiculo = new Motocicleta(168, "Suzuki", "GSX", 1000);
System.out.println(vehiculo.getInformacion());
```

```
Soy un coche con una velocidad máxima de 100.0 km/h
Soy un vehículo de la marca Suzuki. En concreto, una motocicleta de cilindrada 1000 cc
```

7.2 Explícito

En otras ocasiones, queremos que una clase hija (subclase) contenga a una clase padre (superclase). Lo anterior puede parecer que tiene sentido, pero si nos paramos a pensar nos damos cuenta que una clase hija tendrá comportamiento y atributos que no tendrá su clase padre. Por ejemplo, un `Vehiculo` no tiene cilindrada (únicamente la tiene una `Motocicleta`).

De esta manera, si intentamos implementar el siguiente código, el compilador nos dará un error ya que, para el compilador, un `Vehiculo` no es considerado (o no tiene necesariamente que ser) una `Motocicleta`:

```
Motocicleta moto = new Vehiculo(168, "Suzuki", "GSX");
```

Podemos resolver lo anterior haciendo un *cast* o *casting* explícito (haciendo uso de los paréntesis). Sin embargo, si el vehículo no es una `Motocicleta`, nos dará un error en tiempo de ejecución.

Para verlo mejor, vamos a implementar el siguiente código en el que creamos un `Coche` y una `Motocicleta` pero las almacenamos como `Vehiculo`.

```
Vehiculo vehiculoMoto = new Motocicleta(168, "Suzuki", "GSX", 1000);
Vehiculo vehiculoCoche = new Coche(180, "Dacia", "Duster", false);

Motocicleta moto = (Motocicleta) vehiculoMoto;
System.out.println(moto.getInformacion());

moto = (Motocicleta) vehiculoCoche; // ¡EXCEPCIÓN!
```

Cuando hacemos el *casting* de ***vehiculoMoto*** a `Motocicleta`, no hay problema porque, en origen, se trata de una `Motocicleta` y, por tanto, se puede llevar a cabo la conversión de tipos. El problema es al convertir ***vehiculoCoche*** ya que no es una `Motocicleta` sino un `Coche` y se produce una excepción en tiempo de ejecución.

8. Polimorfismo

Lo anterior tiene que ver con una de las características de la Programación Orientada a Objetos (POO) conocida como **polimorfismo**. El polimorfismo se puede dar a nivel de método o de clase.

8.1 A nivel de método

El polimorfismo a nivel de método, nos permite que dos métodos tengan el mismo nombre y hagan funciones distintas, aunque similares, en objetos distintos. Por ejemplo, el método *getInformacion()* para un `Coche` o para una `Motocicleta`.

También puede que tengamos, en la misma clase o en clases heredadas, un método con el mismo nombre pero con diferente número y/o tipo de parámetros. En función de lo anterior, el método hará cosas diferentes (aunque se supone que relacionadas).

Por ejemplo, cuando trabajamos con colecciones sabemos que tenemos el método `add()` que nos permite agregar un nuevo elemento a la colección. Dicho método puede recibir un parámetro con el elemento a añadir a una lista (en ese caso se añadirá al final de la lista) o podemos pasarle también el índice en el que añadir el nuevo elemento.

8.2 A nivel de clase

El polimorfismo a nivel de clase nos permite crear una clase con la referencia a su superclase (clase padre) pero cuyo contenido es el de la subclase (clase hija), como hemos visto anteriormente en este apartado del tema.

Si tenemos el siguiente código, el método *getInformacion()* que se ejecutará es el definido en la clase `Coche`.

```
Vehiculo vehiculo = new Coche(100, "Opel", "Kadett", false);
vehiculo.getInformacion();
```

Lo anterior nos permite trabajar con `Vehiculos` pero que se ejecute el comportamiento o lógica adecuada en función del tipo de objeto que sea realmente (en nuestro ejemplo el `Coche`).

Resumiendo, vamos a tener tres formas distintas de instanciar un objeto y almacenar su referencia:

1) `Hijo varH = new Hijo()` **`Coche varH = new Coche()`**

   Podemos acceder a los métodos y atributos del hijo y del padre. Si hay métodos sobrescritos, accederemos a los del hijo.

2) `Padre varP = new Padre()` **`Vehiculo varP = new Vehiculo()`**

   Podemos acceder a los métodos y atributos del padre. Es imposible acceder a los del hijo (precisamente porque no hay hijo como tal).

3) `Padre varH = new Hijo()` **`Vehiculo varH = new Coche()`**

   Conocido como *upcasting* o conversión implícita**.** Podemos acceder a los métodos y atributos del padre. Si un método del padre está sobrescrito, utilizará el del hijo. Únicamente podemos acceder a los del hijo si hacemos un *downcasting explícito*, es decir, un *casting* con los paréntesis. Recuerda que hacer una conversión de tipos incorrecta produce una excepción **ClassCastException** en tiempo de ejecución.

9. Comprobación de tipos

Para saber si un objeto es de un tipo u otro podemos utilizar la palabra reservada **instanceof**. A continuación se muestra un ejemplo de uso.

```
	Vehiculo vehiculo = new Motocicleta(168, "Suzuki", "GSX", 1000);
	//Vehiculo vehiculo = new Coche(180, "Dacia", "Duster", false);

	if(vehiculo instanceof Motocicleta)
	{
		System.out.println("El vehículo es una motocicleta");
	}
	else
	{
		System.out.println("El vehículo NO es una motocicleta. Su clase es " +
vehiculo.getClass().getSimpleName());
	}
```

La salida por pantalla será:

```
El vehículo es una motocicleta
```

Sin embargo, si comentamos la primera línea de código y descomentamos la segunda la salida por pantalla será la siguiente:

```
El vehículo NO es una motocicleta. Su clase es Coche
```

Esta palabra reservada se utiliza mucho antes de realizar un *casting* explícito de clases ya que nos permite evitar la excepción que hemos comentado anteriormente. En el caso que el *vehiculo* fuese un coche, no se produciría la conversión.

```
	if(vehiculo instanceof Motocicleta)
	{
		Motocicleta moto = (Motocicleta) vehiculo;
	}
```

10. Modificar final

Un aspecto importante que tenemos que conocer también es cómo aplica la palabra reservada **`final`** al trabajar con herencia.

10.1 En clases

Nos permite impedir que una clase sea heredada por otra. Si intentamos crear otra clase heredada el compilador producirá un error. Por ejemplo:

```
public final class Vehiculo { ... }
```

En este momento, en nuestras clases `Coche` y `Motocicleta` se producirá un error.

10.2 En métodos

Nos sirve para impedir que un método, implementado en una clase padre, sea modificado (sobrescrito) por las clases que heredan (por las clases hijas).

Si una clase hija define un método con el mismo nombre y parámetros que el de la clase padre (es decir, si define un método que sobrescriba al del padre) el compilador producirá un error.

Esto es muy común encontrarlo cuando trabajamos con clases abstractas (las veremos más adelante) o interfaces, donde se obliga a implementar los métodos de éstas si son heredadas o implementadas respectivamente.

En nuestra clase `Vehiculo` podemos realizar lo siguiente y eso impedirá que las clases `Coche` y `Motocicleta` puedan sobrescribir el método *getInformacion()*.

```
public final String getInformacion() {
	return "Soy un vehículo de la marca " + getMarca();
}
```

11. Paso de superclases y subclases por parámetro

En ocasiones nos interesará pasar una subclase o colección de subclases a un método y que el método reciba la superclase (clase padre) o una colección de la superclase. En función de si pasamos un único objeto o una colección de ellos, debemos implementar distintas firmas en el método.

11.1 Paso de un único objeto

Si únicamente vamos a recibir por parámetro un único objeto podemos declarar directamente la clase padre.

```
private static void mostrarVehiculo(Vehiculo vehiculo) {
	// ...
}
```

A este método podemos pasarle tanto un `Coche` o una `Motocicleta`. Obviamente, también podemos pasarle un `Vehiculo`. El siguiente código es totalmente funcional:

```
	// Coche
	Coche coche = new Coche(180, "Dacia", "Duster", false);
	mostrarVehiculo(coche);

	// Moto
	Motocicleta moto = new Motocicleta(168, "Suzuki", "GSX", 1000);
	mostrarVehiculo(moto);

	// Vehículo
	Vehiculo vehiculo = new Vehiculo(30, "Xiaomi", "Scooter 4 Pro");
	mostrarVehiculo(vehiculo);
```

11.2 Paso de una colección de objetos

Cuando lo que queremos es recibir una colección (lista o conjunto) de vehículos podemos pensar en realizar el siguiente código:

```
private static void mostrarVehiculos(List<Vehiculo> vehiculos) {
	// ...
}
```

El anterior método únicamente permitirá pasarle por parámetro una colección de `Vehiculo` pero no de `Coche` o de `Motocicleta`, es decir, no permitirá llamar al método pasándolo ni `List<Coche>` ni `List<Motocicleta>`. A continuación, se muestra en rojo qué líneas de código son las que el compilador nos indica que no son correctas y en verde la llamada al método que es permitida.

```
	// Coche
	Coche coche = new Coche(180, "Dacia", "Duster", false);
	List<Coche> listaCoches = new ArrayList<Coche>();
	listaCoches.add(coche);
	mostrarVehiculos(listaCoches);

	// Moto
	Motocicleta moto = new Motocicleta(168, "Suzuki", "GSX", 1000);
	List<Motocicleta> listaMotocicletas = new ArrayList<Motocicleta>();
	listaMotocicletas.add(moto);
	mostrarVehiculos(listaMotocicletas);

	// Vehículos
	Vehiculo vehiculo = new Coche(100, "Opel", "Kadett", false);
	List<Vehiculo> listaVehiculos = new ArrayList<Vehiculo>();
	listaVehiculos.add(vehiculo);
	listaVehiculos.add(moto);
	listaVehiculos.add(coche);
	mostrarVehiculos(listaVehiculos);
```

Como vemos, si queremos implementar un método que pueda recibir una colección de la superclase (`Vehiculo`) pero que también permita ser llamado explícitamente con objetos de las subclases (`Coche` y `Motocicleta`) debemos implementar la firma del método de la siguiente forma:

```
private static void mostrarVehiculos(List<? extends Vehiculo> vehiculos) {
	// ...
}
```

Con ello conseguiremos que los dos errores que nos aparecían antes, desaparezcan, pudiendo ejecutar las siguientes líneas de código:

```
	List<Coche> listaCoches = new ArrayList<Coche>();
	mostrarVehiculos(listaCoches);

	List<Motocicleta> listaMotocicletas = new ArrayList<Motocicleta>();
	mostrarVehiculos(listaMotocicletas);
```

12. Comparativa getClass() vs instanceof

La principal diferencia radica en que `getClass()` únicamente va a devolver verdadero si el objeto es una instancia de la clase indicada e `instanceof` también tiene en cuenta si es una sublcase de la clase especificada. No obstante, vamos a ver un ejemplo para entender mejor la diferencia, basándonos en el siguiente código fuente:

```
Coche cocheDisfrazadoDeCoche = new Coche(120, "Coche", "Coche", false);
Vehiculo cocheDisfrazadoDeVehiculo = new Coche(120, "Vehículo", "Coche", false);
Vehiculo vehiculoDisfrazadoDeVehiculo = new Vehiculo(120, "Vehículo", "Vehículo");
```

La tabla de salida de `instanceof` y `getClass` quedaría de la siguiente forma:

	instanceof Vehiculo	instanceof Coche	getClass() Vehiculo	getClass() Coche
`cocheDisfrazadoDeCoche`	true	true	X	true
`cocheDisfrazadoDeVehiculo`	`true`	true	`false`	true
`vehiculoDisfrazadoDeVehiculo`	true	false	true	false

En una aproximación, poco técnica pero útil, podemos seguir el siguiente "*truco*":

- ¿Cuándo va a devolver verdadero `instanceof`?

  Cuando el objeto **pueda hacerse pasar por** esa clase o sea de esa clase. Por ejemplo, un Coche disfrazado de Vehículo se pueda hacer pasar por un Vehículo pero un Vehículo no se puede hacer pasar por un Coche.

- ¿Cuándo va a devolver verdadero `getClass()`?

  Cuando el objeto **ha sido creado como** ese tipo de clase, es decir, lo que realmente es sin disfraz. Por ejemplo, un Coche disfrazado de Vehículo no es de la clase Vehículo (no fue creado o instanciado como Vehículo).

13. UML

En UML, la herencia se representa con una flecha de punta cerrada y línea continua desde las subclases (o clases hijas) hacia la superclase (o clase padre).