HERENCIA EN JAVA

Charly Cimino

Herencia en Java

Charly Cimino

Versión 2023.06.26

Definición

Mecanismo por el cual se logran reaprovechar los miembros de una o varias clases ya existentes.

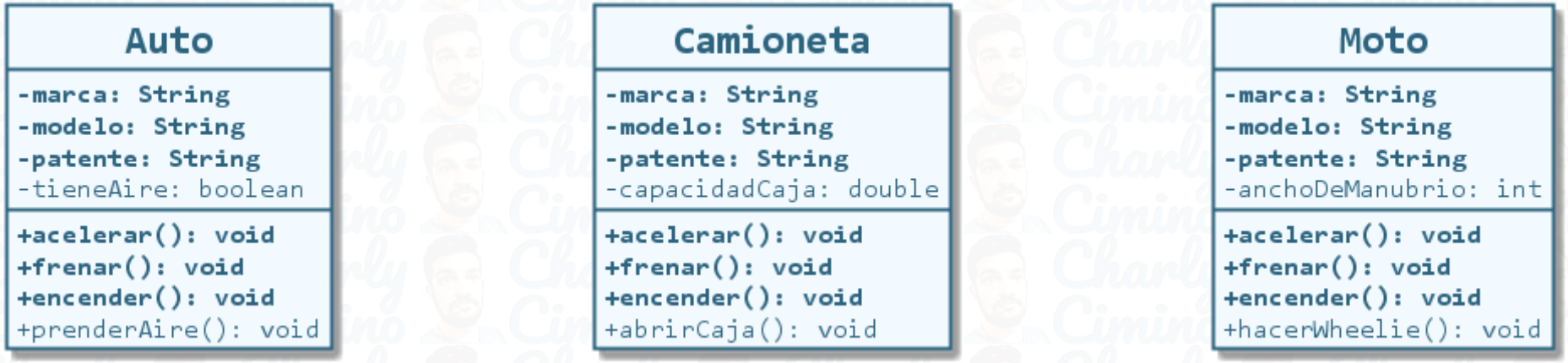
Herencia simple

Herencia múltiple

No soportada en Java

Caso notorio

Aquí parece haber clases que representan diferentes entidades, pero que a su vez tienen varias semejanzas.

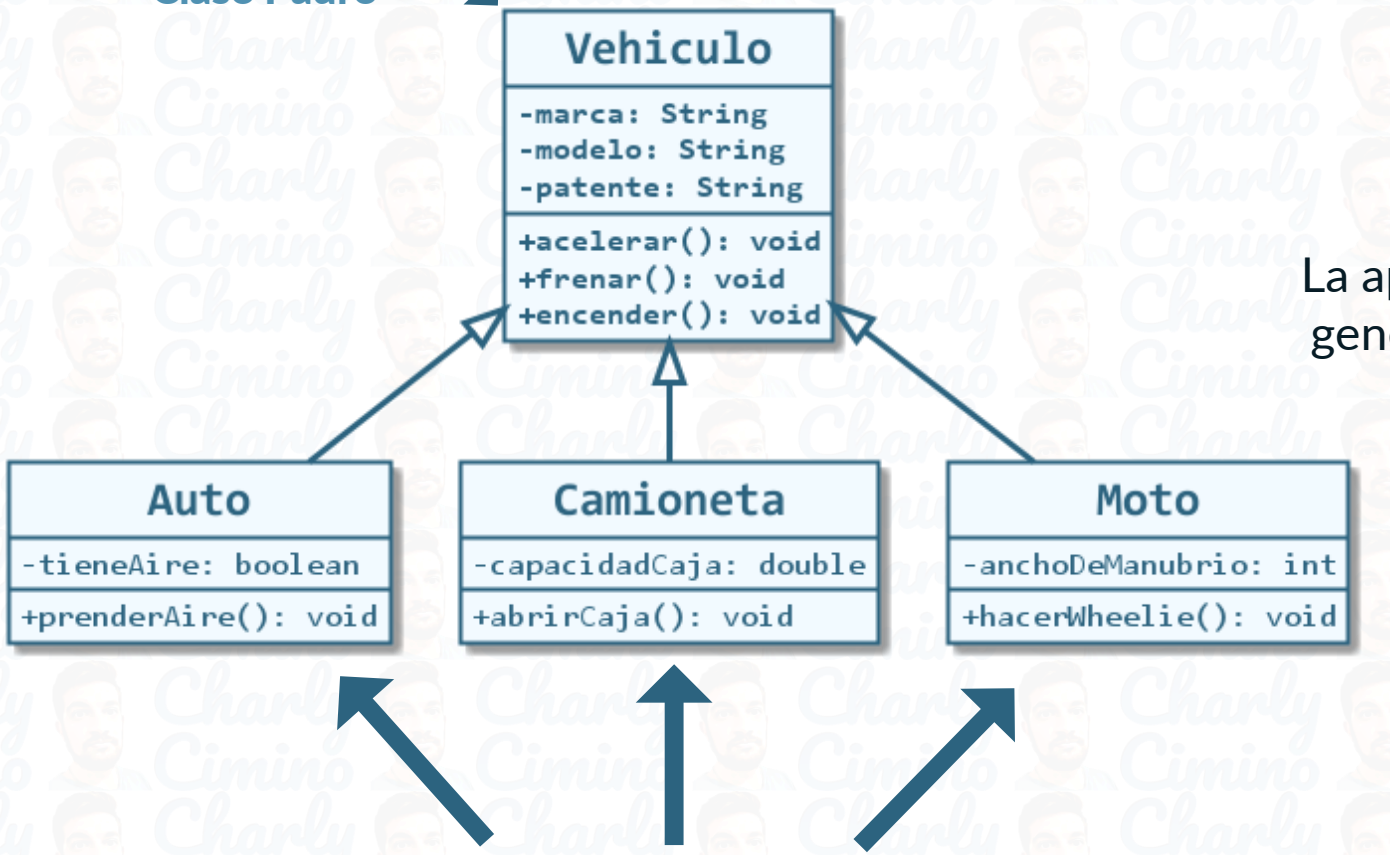


Los miembros marcados en **negrita** son comunes a las tres clases.

La herencia nos permite hacer un 'factor común' de estos miembros, evitando la repetición.

Aplicando herencia

Superclase
Clase Base
Clase Padre

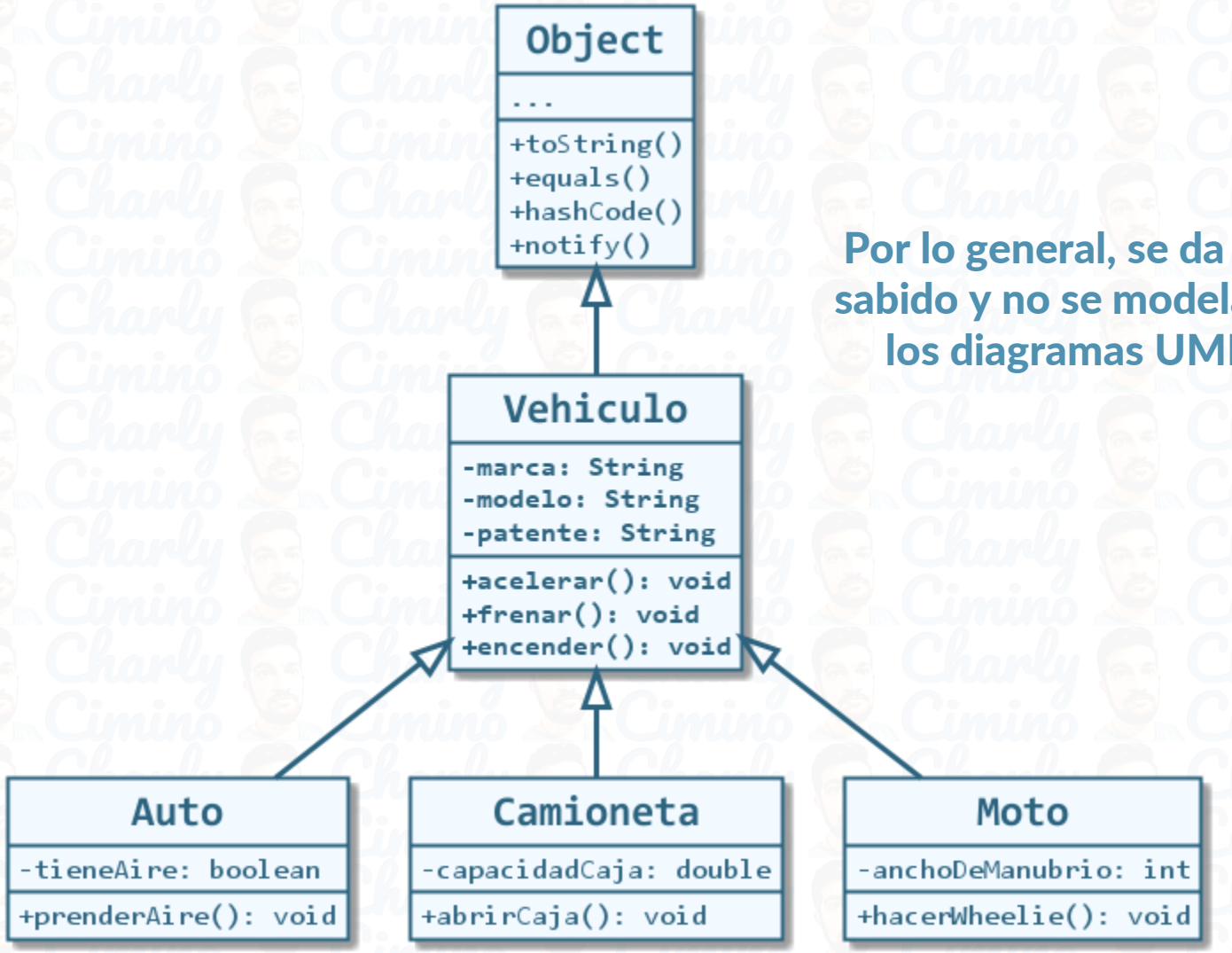


Subclases
Clases Derivadas
Clases Hijas

La aparición de la clase **Vehiculo** permite generalizar ciertos aspectos del modelo.

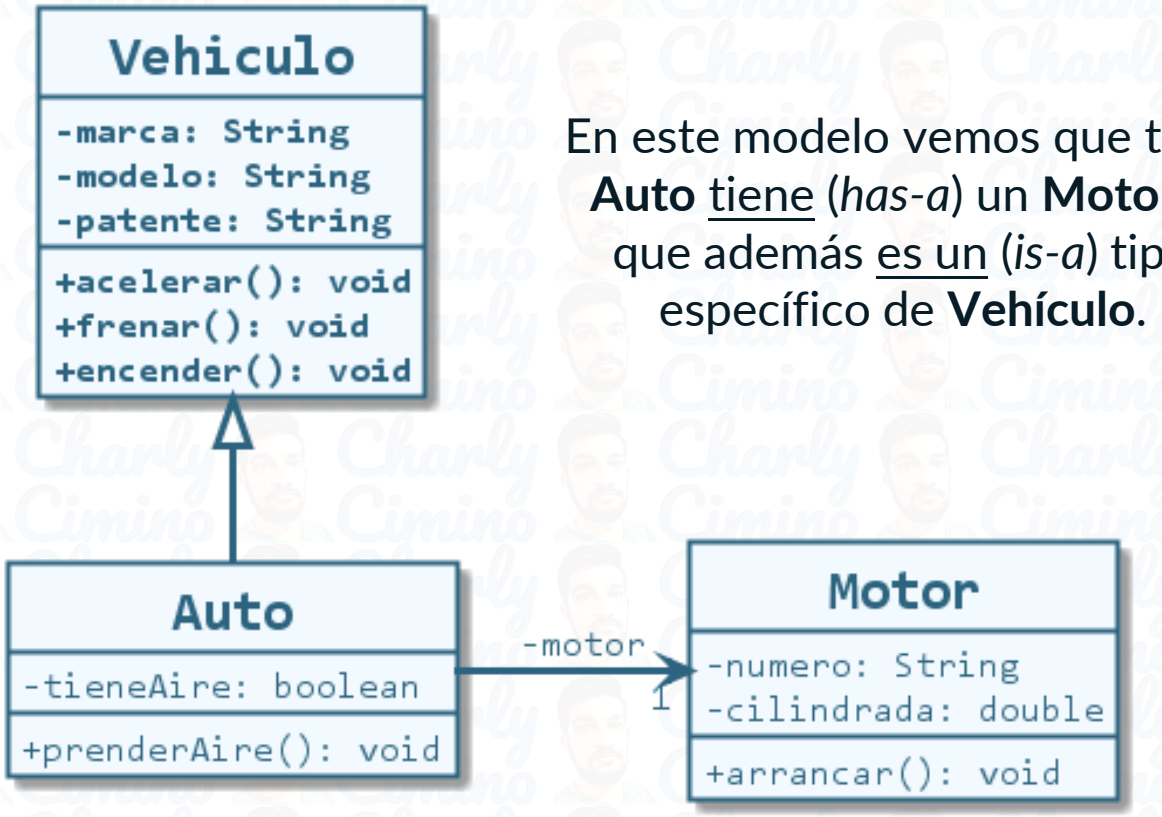
Herencia impuesta por Java

Toda clase en Java hereda directa o indirectamente de la clase `Object`.



Por lo general, se da por sabido y no se modela en los diagramas UML.

‘has-a’ vs. ‘is-a’



En este modelo vemos que todo **Auto** tiene (*has-a*) un **Motor** y que además es un (*is-a*) tipo específico de **Vehículo**.

Asociación

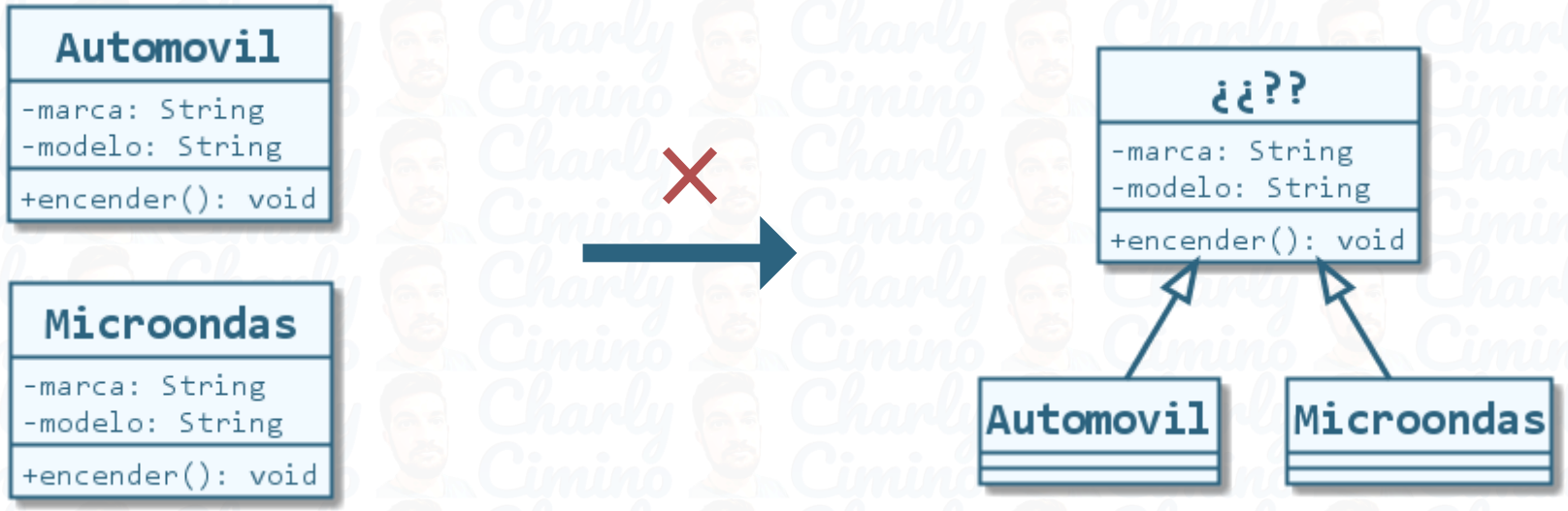
has-a (tiene un/a)

Generalización

is-a (es un/a)

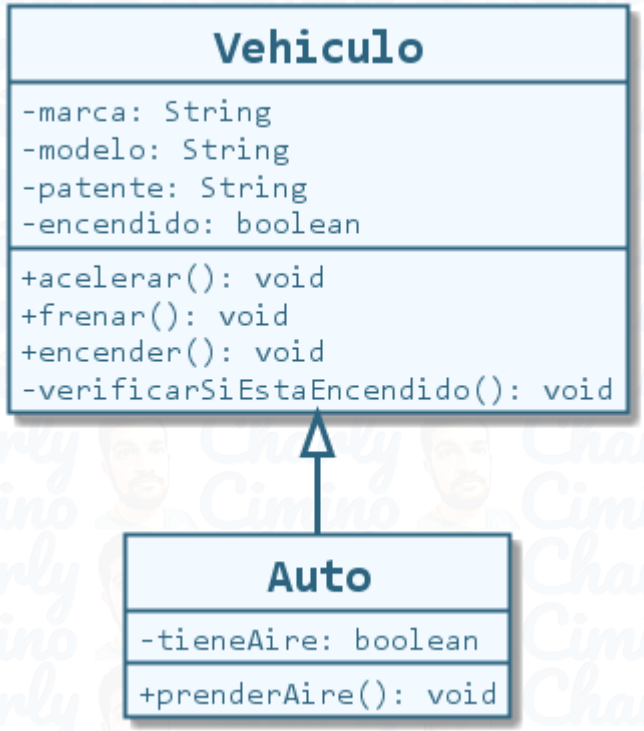
Coherencia conceptual

Que dos clases tengan atributos y métodos homónimos no las vuelve necesariamente parte de la misma jerarquía.



¿Qué se hereda?

Toda subclase hereda todos los miembros **no privados** de la superclase (excepto los constructores).



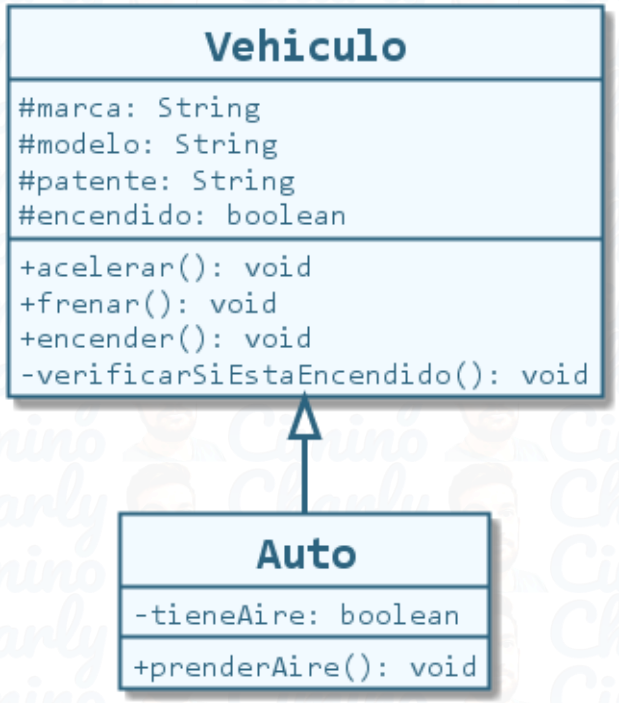
Auto.java

```
public Auto() { // Prueba en el constructor
    System.out.println(marca); ❌ Atributo privado en la superclase
    acelerar(); ✓ Método no privado en la superclase (se hereda)
    verificarSiEstaEncendido(); ❌ Método privado en la superclase
}
```

“Público” y “No privado” no son lo mismo. Veamos por qué…

Modificadores de acceso

Modificador en Java	Modificador en UML	Misma clase	Subclase en mismo paquete	Clase en mismo paquete	Subclase en otro paquete	Clase en otro paquete
`public`	+	✓	✓	✓	✓	✓
`protected`	#	✓	✓	✓	✓	✗
`(sin modificador)`		✓	✓	✓	✗	✗
`private`	-	✓	✗	✗	✗	✗



Colocar los atributos como protegidos (**protected**) puede derivar en problemas de encapsulamiento:

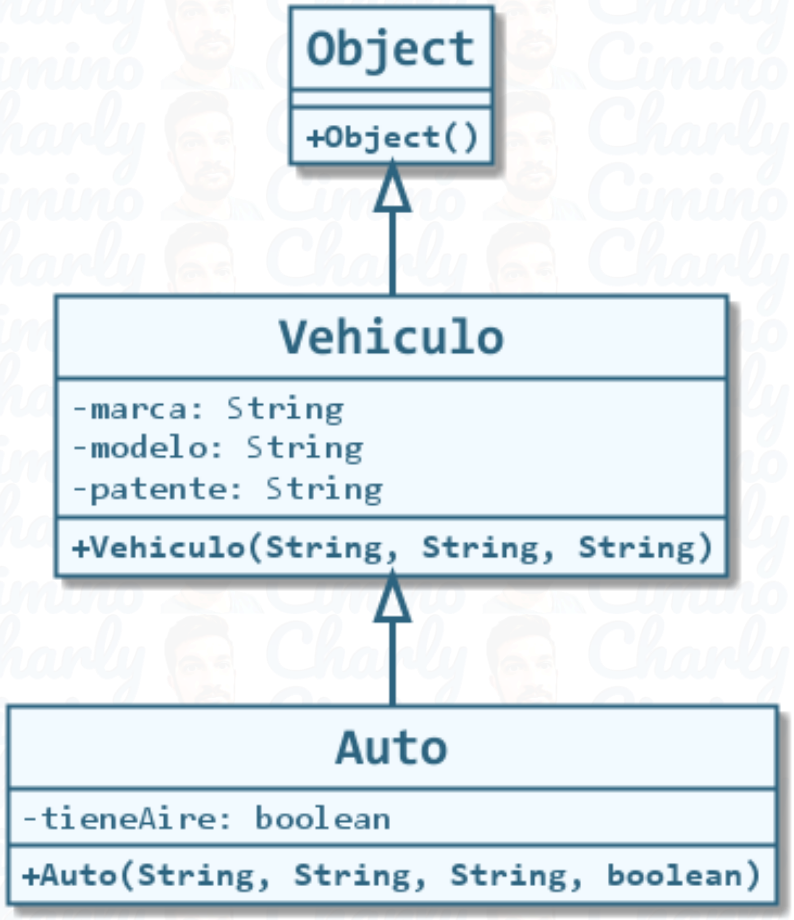
1) Habría que asegurarse que las clases que componen a la jerarquía residan en un paquete de forma exclusiva, ya que cualquier otra clase en el mismo paquete podría acceder directamente a sus atributos (ver tabla).
2) Una subclase podría modificar el valor de alguno de los atributos de la superclase, provocando un posible estado inconsistente (tener en cuenta que los autores de cada clase pudieron haber sido distintas personas).

Conclusión: Atributos privados y métodos getter/setter cuando sean necesarios.

Constructores

Los constructores **no se heredan.**

Cada clase debe tener su/s constructor/es, que deben primero invoque/n al constructor de la superclase.



Vehiculo.java

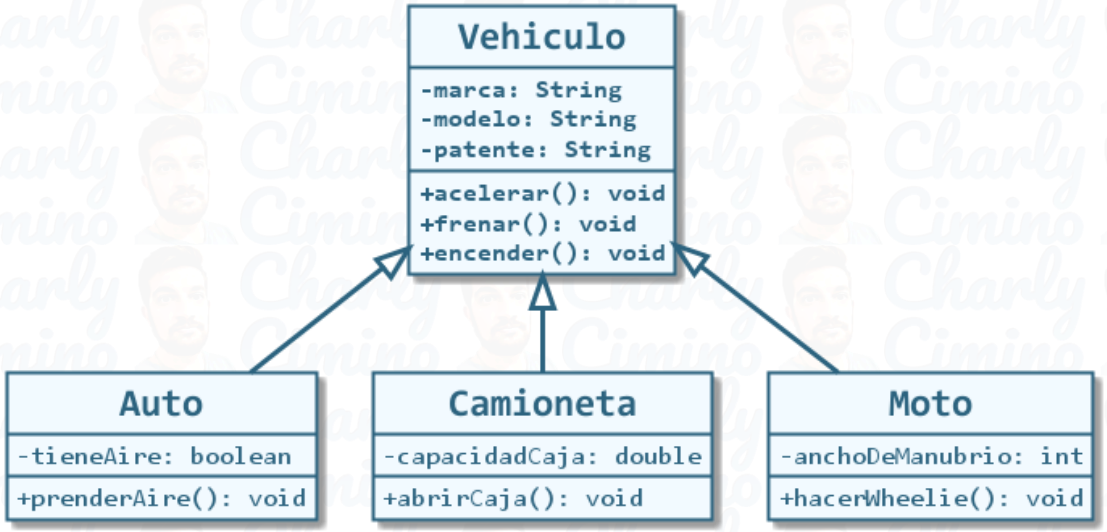
```java
public Vehiculo(String ma, String mo, String pa) {
    super(); // Invoca al constructor de la superclase (Object)
    this.marca = ma;
    this.modelo = mo;
    this.patente = pa;
}
```

Auto.java

```java
public Auto(String ma, String mo, String pa, boolean ta) {
    super(ma,mo,pa); // Invoca al constructor de la superclase (Vehiculo)
    this.tieneAire = ta;
}
```

Generalización (*upcasting*)

Al haber una relación de herencia, podemos guardar en una variable o parámetro de tipo **T** referencias a objetos de tipo **T** o derivados.



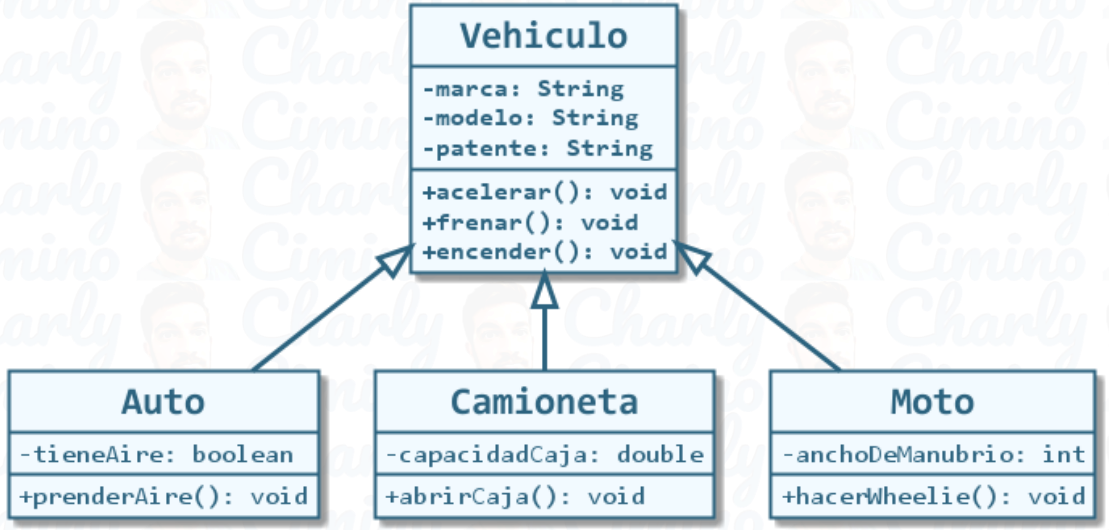
Test.java

```
public static void main(…) {
    Auto a = new Auto(...); ✓ Posible
    Vehiculo v = new Vehiculo(...); ✓ Posible
    Vehiculo v2 = new Auto(...); ✓ Posible: Todo Auto es un Vehículo
    Moto m = new Vehiculo(...); ❌ Imposible: No todo Vehículo es una Moto
}
```

La clave es recordar la relación “es un” (*is-a*)

Generalización (*upcasting*)

El casting hacia arriba (*upcasting*) se hace de manera implícita.



Test.java

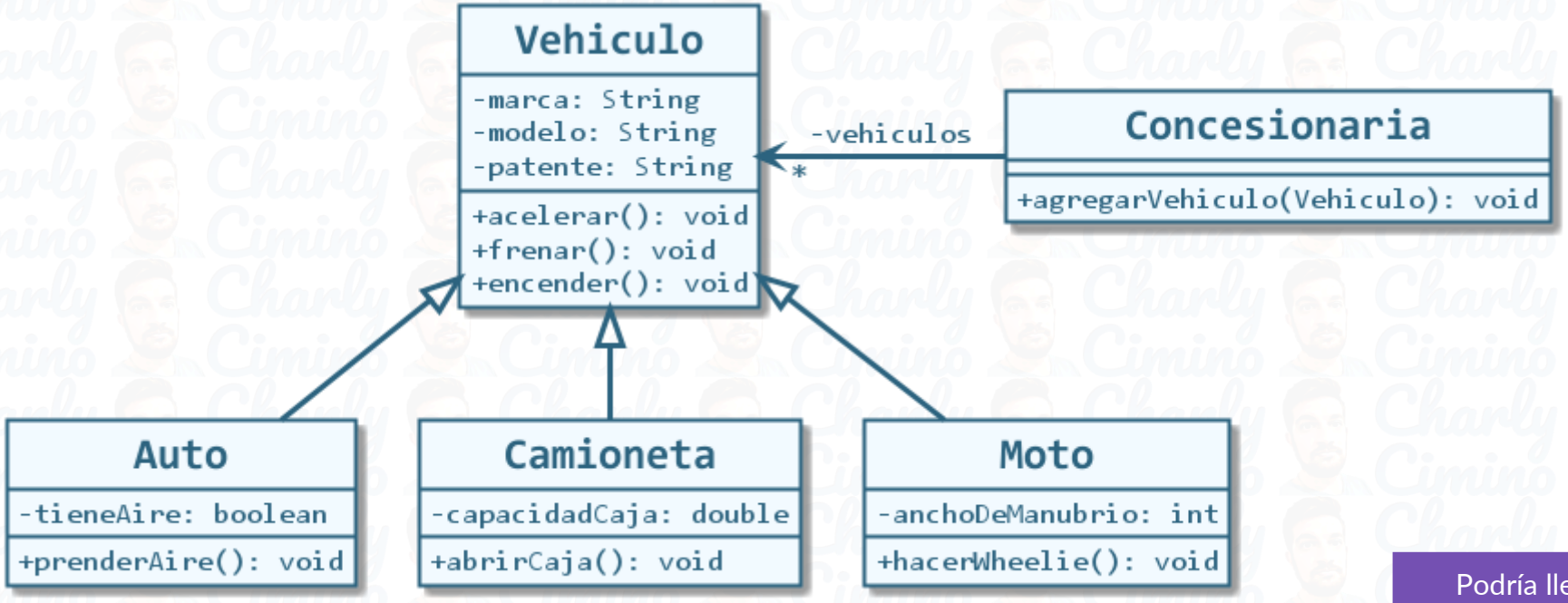
```java
public static void main(…) {
    Auto a = new Auto(...);

    Vehiculo v = a;
    Vehiculo v2 = (Vehiculo) a;
}
```

Es lo mismo

Generalización (*upcasting*)

El *upcasting* nos evita repetir código muy similar.



Nos ahorramos esto:

```
public void agregarAuto(Auto a) {...}
public void agregarCamioneta(Camioneta c) {...}
public void agregarMoto(Moto m) {...}
```

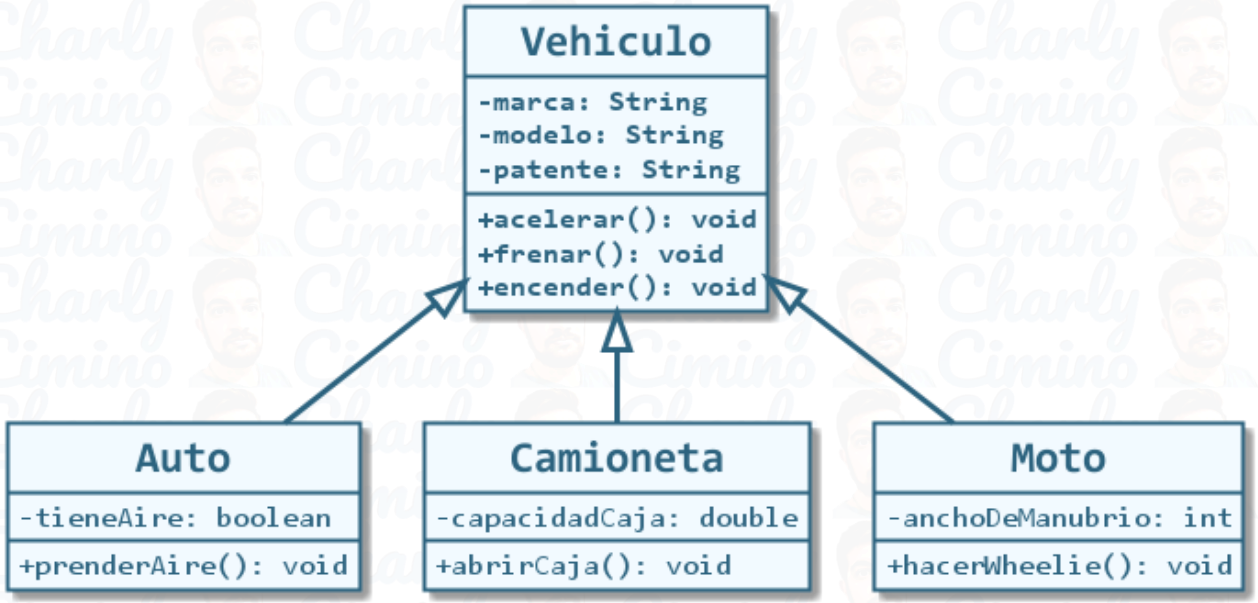
Concesionaria.java

```
public void agregarVehiculo(Vehiculo v) {
    this.vehiculos.add(v);
}
```

Podría llegar un Vehículo propiamente dicho, un Auto, una Camioneta, una Moto o cualquier subclase de Vehículo.

Generalización (*upcasting*)

Generalizar tiene sus consecuencias.



Test.java

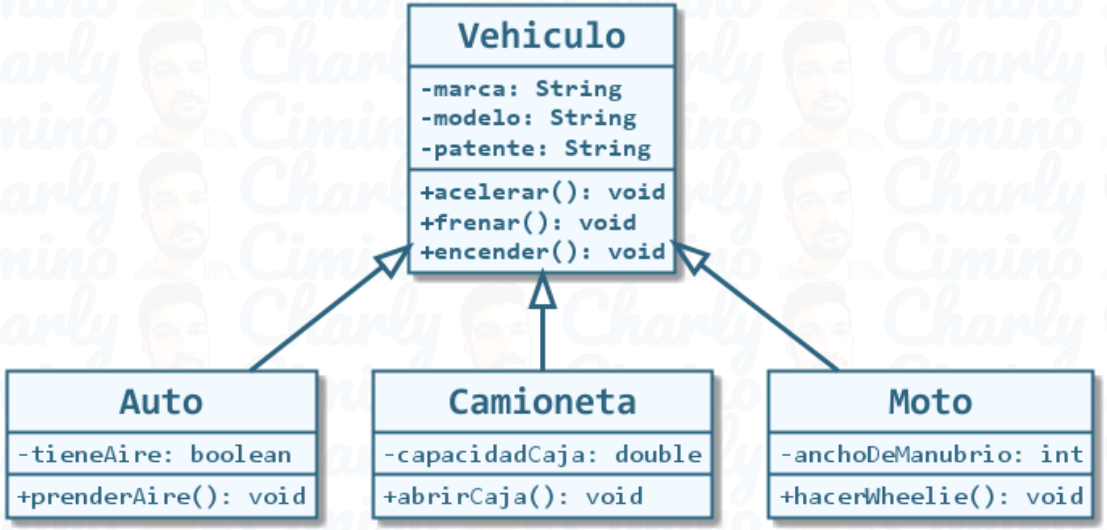
```
public static void main(…) {
    Auto a = new Auto(...);
    a.acelerar(); ✓ Posible
    a.prenderAire(); ✓ Posible

    Vehiculo v = a;
    v.acelerar(); ✓ Posible
    v.prenderAire(); ❌ Imposible (¿Por qué?)

    Object x = a;
    x.acelerar(); ❌ Imposible (¿Por qué?)
    x.prenderAire(); ❌ Imposible (¿Por qué?)
    // ¿A qué métodos se podrá invocar desde 'x'?
}
```

Especialización (*downcasting*)

Como hemos visto, la especialización por sí sola no funciona.

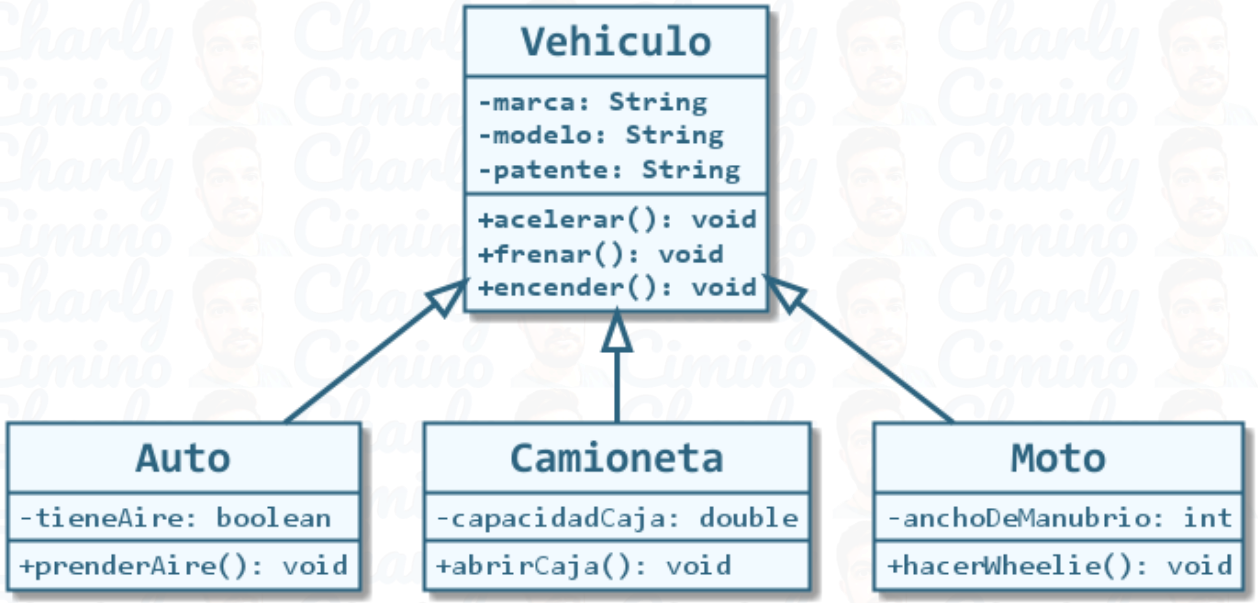


Test.java

```
public static void main(…) {
    Auto a = new Auto(...); ✓ Posible
    Vehiculo v = new Vehiculo(...); ✓ Posible
    Vehiculo v2 = new Auto(...); ✓ Posible: Todo Auto es un Vehículo
    Moto m = new Vehiculo(...); ❌ Imposible: No todo Vehículo es una Moto
}
```

Especialización (*downcasting*)

Para poder especializar, se necesita un casteo explícito.



Test.java

```
public static void main(…) {
    Vehiculo v = new Moto(...);
    Moto m = v; ❌ No compila
    Moto m2 = (Moto) v; ✓ Compila
    m2.hacerWheliee(); ✓ Funciona
}
```

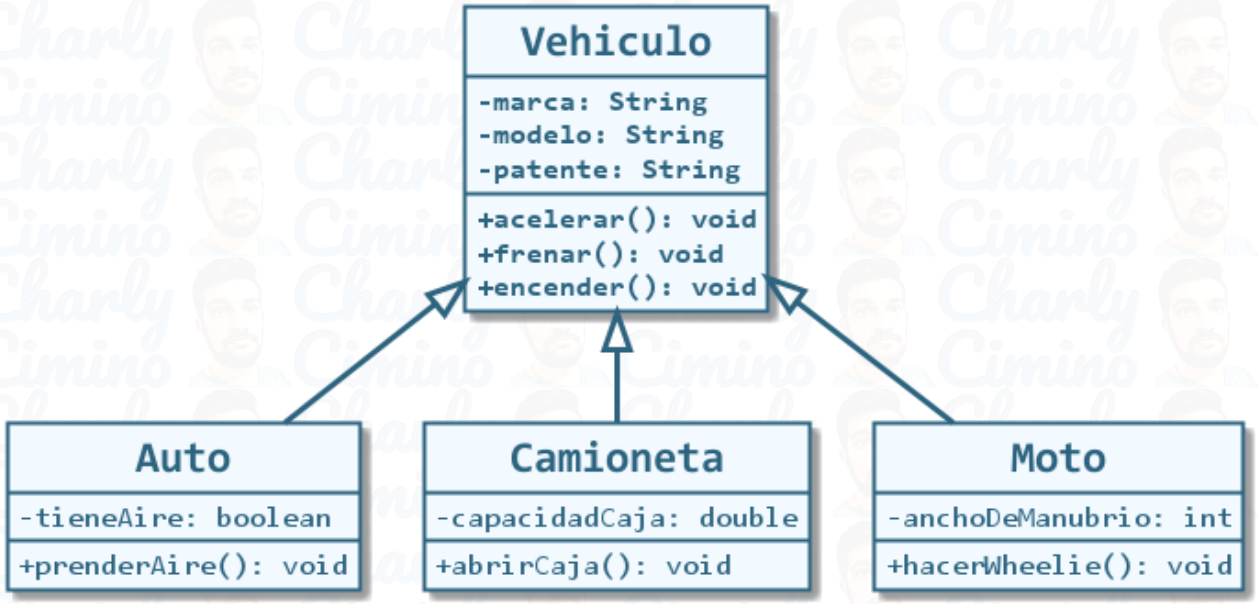
Sin embargo...

Test.java

```
public static void main(…) {
    Vehiculo v = new Auto(...);
    Moto m = (Moto) v; ✓ Compila
    m.hacerWheliee(); - ¿Funciona?
}
```

Especialización (*downcasting*)

Los objetos “nacen” en tiempo de ejecución. Habrá errores no detectables en tiempo de compilación.



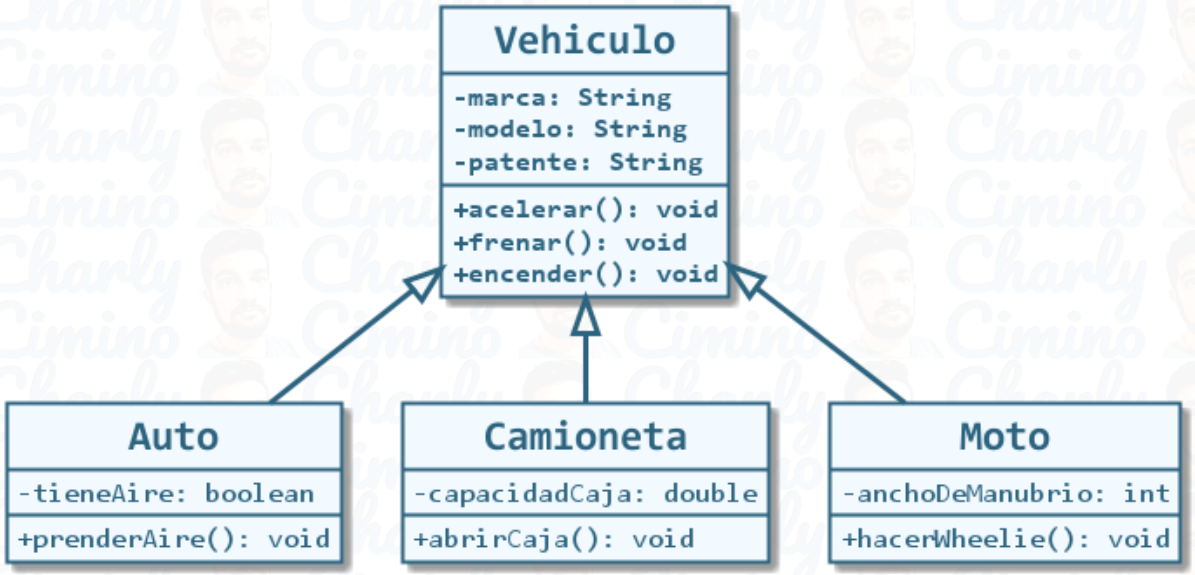
Test.java

```java
public static void main(…) {
    Vehiculo v = new Auto(...);
    Moto m = (Moto) v;
    m.hacerWheliee();
}
```

Falla en tiempo de ejecución. **ClassCastException**: Un Auto no puede ser casteado a Moto.

instanceof

Operador lógico (retorna un **boolean**) que nos permite conocer si una variable contiene un objeto de un determinado tipo.

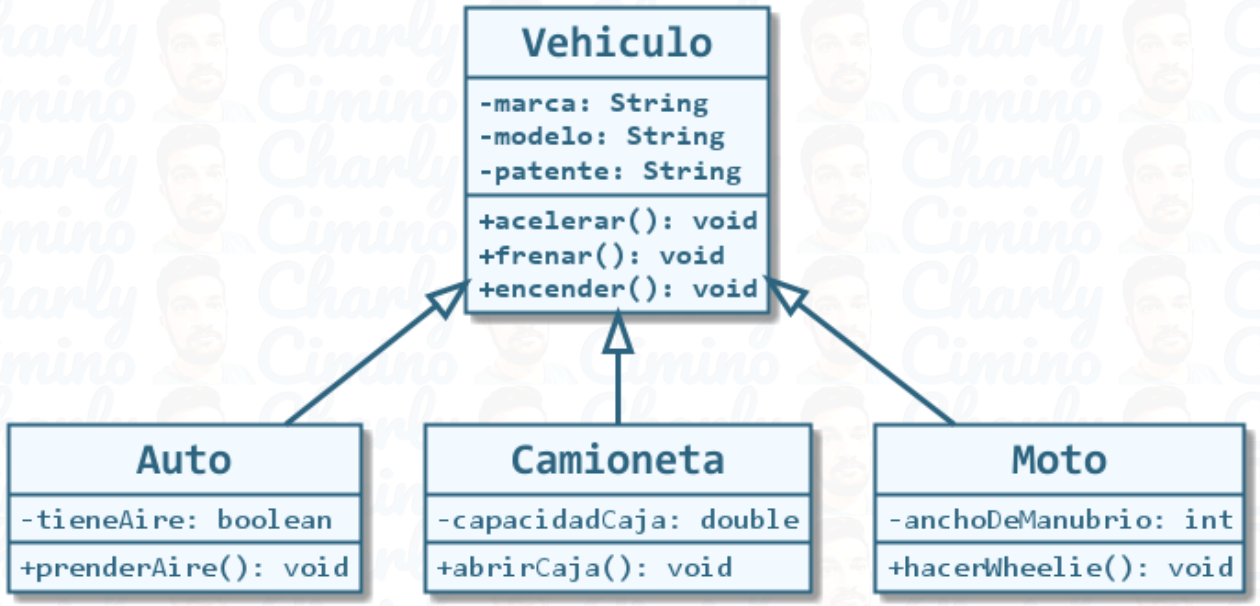


Test.java

```java
public static void main(…) {
    Vehiculo v = new Moto(...);
    System.out.println( v instanceof Moto ); // true
    System.out.println( v instanceof Vehiculo ); // true
    System.out.println( v instanceof Object ); // true
    System.out.println( v instanceof Auto ); // false
    System.out.println( v instanceof Camioneta ); // false
// Solo se puede preguntar por tipos que pertenezcan a la jerarquía
    System.out.println( v instanceof String ); // No compila
}
```

Evitar ClassCastException

Antes de hacer *downcasting*, debemos asegurarnos que tal variable apunte a un objeto del tipo esperado.

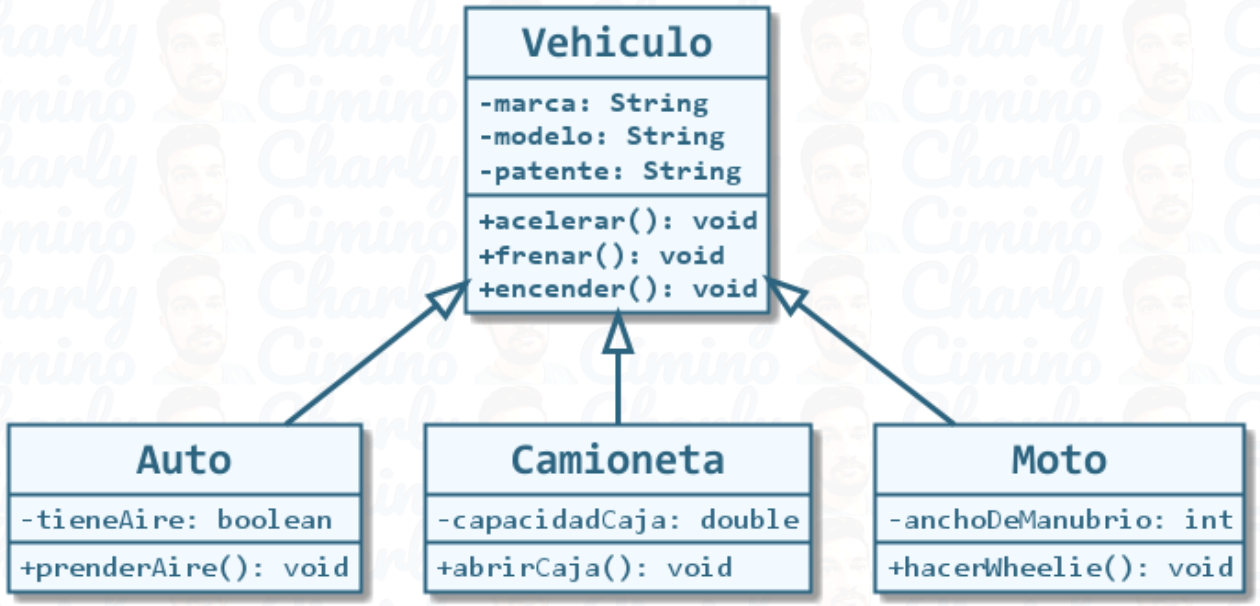


Test.java

```java
public static void main(…) {
    Vehiculo v = new Auto(...);
    if (v instanceof Moto) {
        Moto m = (Moto) v;
        m.hacerWheliee();
    } else {
        System.out.println("No era una Moto");
    }
}
```

¿Existen los vehículos como tales?

La clase `Vehiculo` sirvió para evitar la repetición de código, sin embargo, ¿qué representan sus instancias?



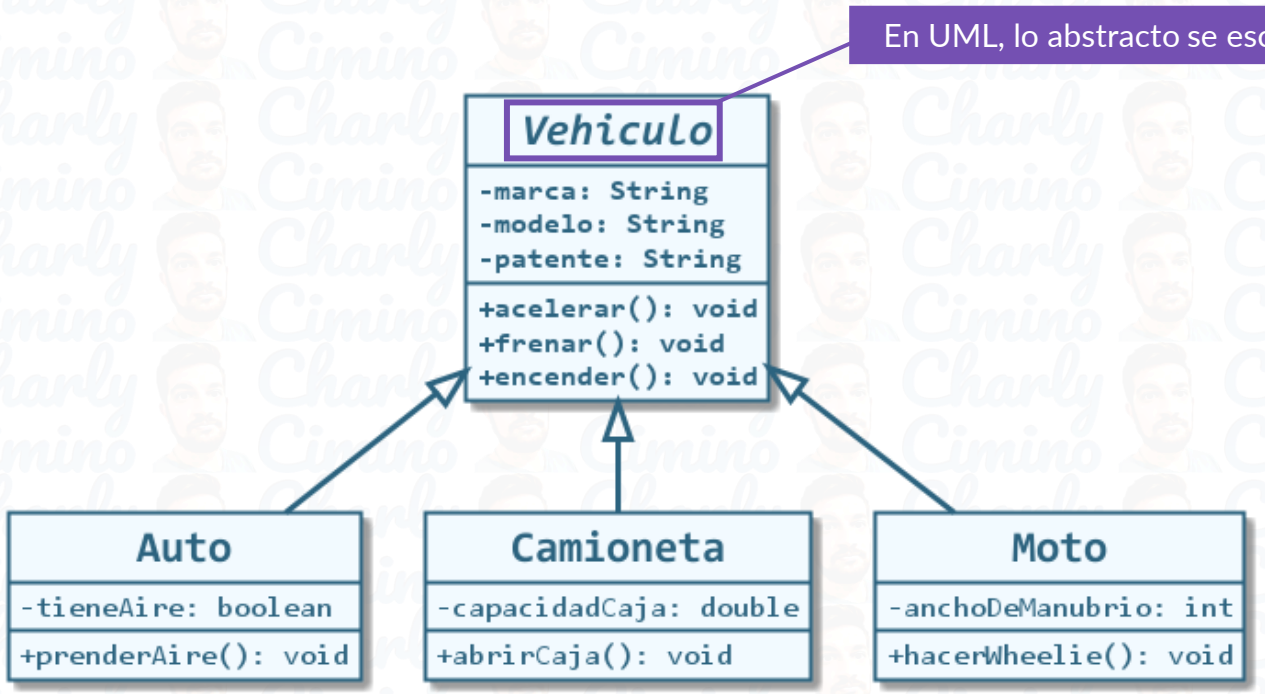
Test.java

```java
public static void main(…) {
    Vehiculo v = new Vehiculo(...);
    /*
        ¿Cómo dibujarías a un 'Vehículo' concreto?
    */
}
```

Clase abstracta

Una clase abstracta es aquella que no puede ser instanciada. Ocupan el rol de superclase en un modelo.

Solo sirven para generalizar aspectos de un conjunto de clases relacionadas.



Test.java

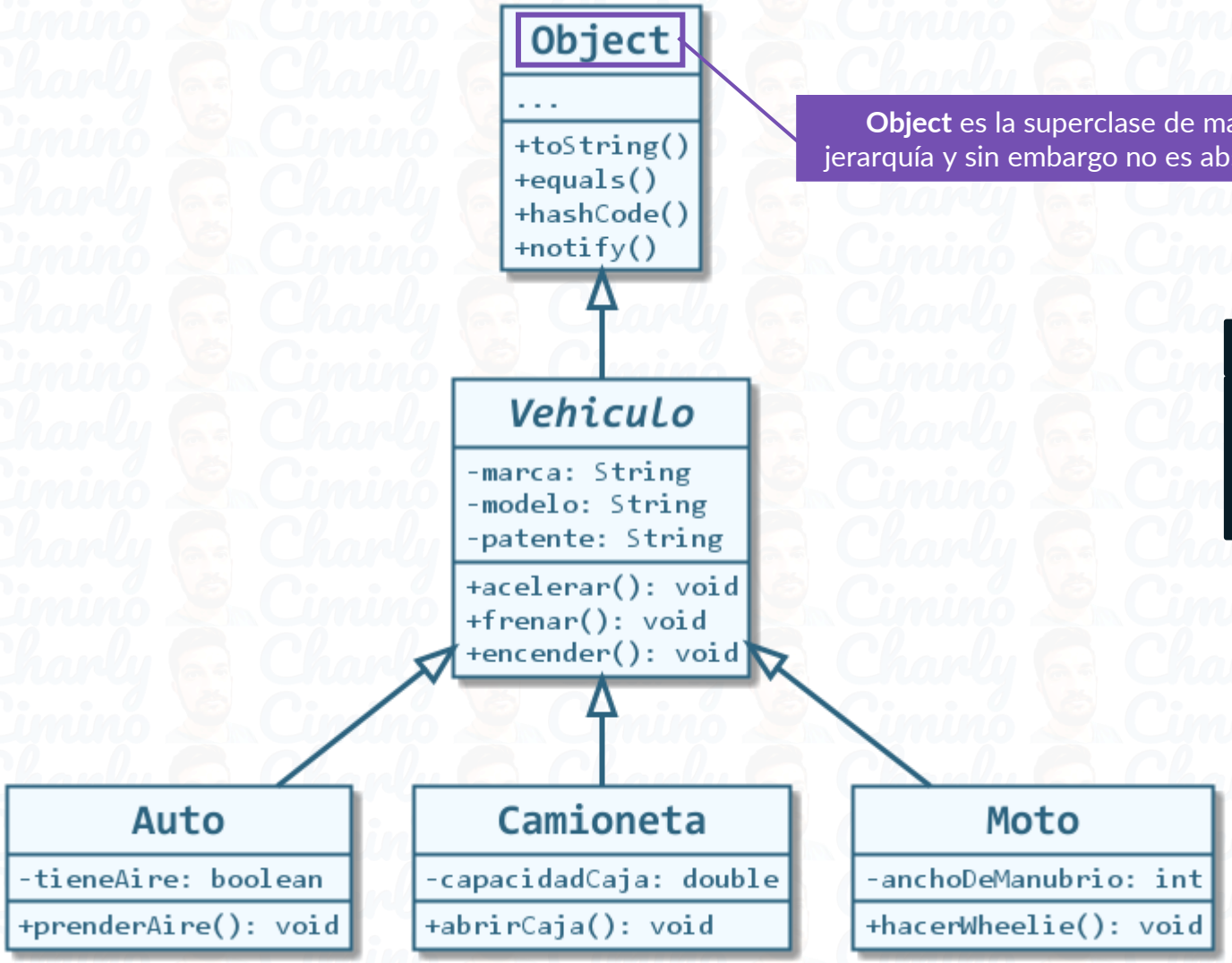
```
public static void main(…) {
    Vehiculo v = new Vehiculo(...);
}
```

No se pueden crear objetos de un tipo abstracto.

Se sigue pudiendo tener variables/parámetros/retornos de tipo **`Vehiculo`** para hacer *upcasting*.
Dentro solo habrá referencias a instancias de subclases concretas.

Clase abstracta

No toda superclase debe ser necesariamente abstracta.



Object es la superclase de mayor jerarquía y sin embargo no es abstracta.

Test.java

```
public static void main(…) {
    Object x = new Object(...); ✓ Válido
}
```

FIN DE LA PRESENTACIÓN

Encontrá más como estas en mi sitio web.