Centro Asociado Palma de Mallorca

Introducción Práctica de Programación Java

Antonio Rivero Cuesta

Sesión

IX

Composición

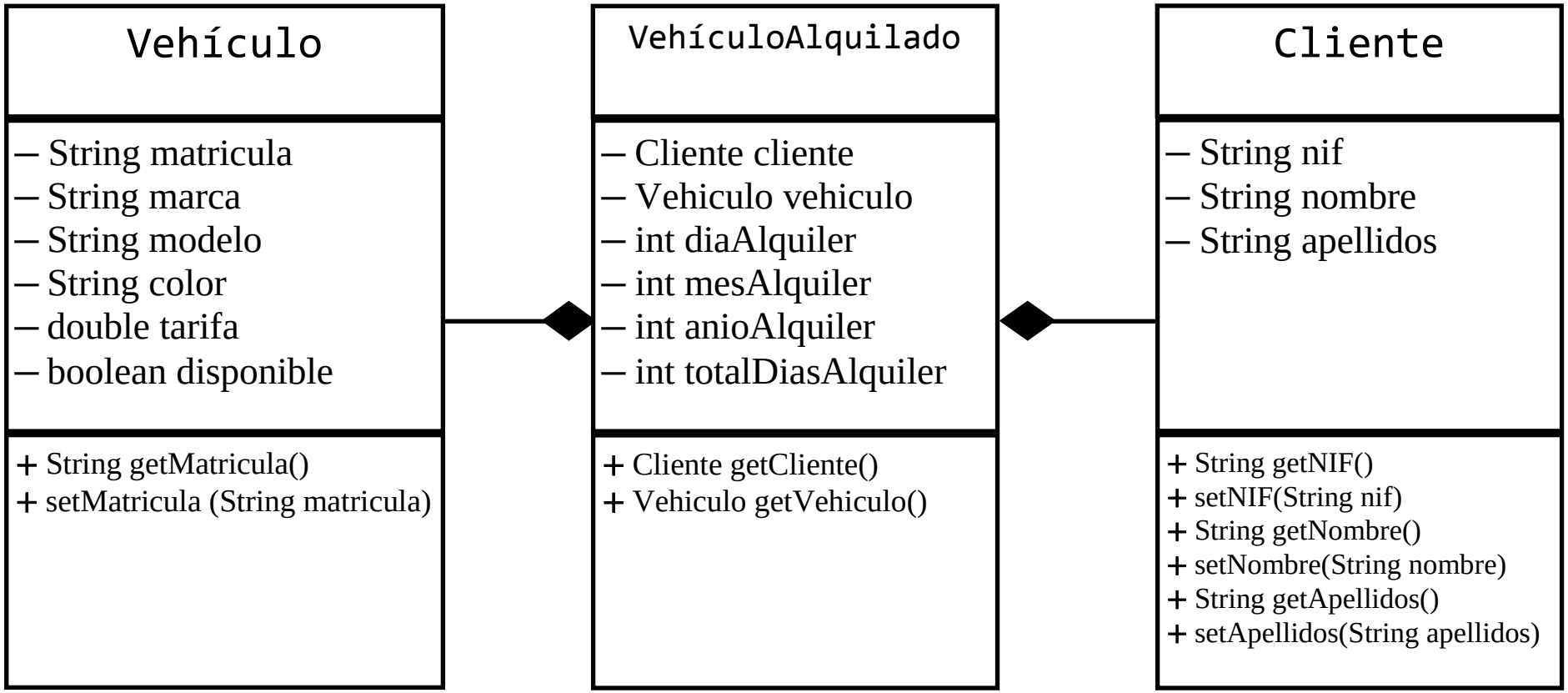
Consiste en crear una clase nueva agrupando objetos de clases que ya existen.

Agrupa uno o más objetos para construir una clase.

Las instancias de esta nueva clase contienen uno o más objetos de otras clases.

Normalmente los objetos contenidos se declaran con acceso **`private`** y se inicializan en el constructor de la clase.

Relación: "Tiene un"



Visibilidad	public	protected	default	private
UML	+	#	~	–
Desde la misma clase	SI	SI	SI	SI
Desde cualquier clase del mismo paquete	SI	SI	SI	NO
Desde una subclase del mismo paquete	SI	SI	SI	NO
Desde una subclase fuera del mismo paquete	SI	SI Herencia	NO	NO
Desde cualquier clase fuera del paquete	SI	NO	NO	NO

Herencia

Es la capacidad que tienen los lenguajes orientados a objetos para extender clases.

Esto produce una nueva clase que hereda el comportamiento y los atributos de la clase que ha sido extendida.

La clase original se denomina ***clase base*** o ***superclase***.

La nueva clase se denomina ***clase derivada*** o ***subclase***.

Para declarar la herencia en Java usamos la palabra clave **`extends`**.

```
public class Clase2 extends Clase1{
```

Una subclase es una especialización de la superclase.

La capacidad para extender clases se llama herencia porque la nueva clase hereda todos los atributos y los métodos de la superclase a la que extiende.

Normalmente una subclase añade nuevos atributos y métodos que le dan un comportamiento diferente al de la superclase.

Permite la reutilización del código.

Herencia e Inicialización

Cuando creamos un objeto, el constructor de dicho objeto se encarga de inicializar todos los campos del objeto en algún estado razonable.

La superclase tendrá un constructor aun cuando no tengamos intención de crear, de manera directa, una instancia de la superclase.

Este constructor recibe los parámetros necesarios para inicializar los campos de una instancia y contiene el código para llevar a cabo esta inicialización.

El constructor de la subclase recibe los parámetros necesarios para inicializar tanto los campos propios como los de la superclase.

El constructor de la subclase contendrá el siguiente código:

```
super(<lista de parámetros>);
```

La palabra clave **`super`** es, en realidad, una llamada al constructor de la superclase.

El efecto de esta llamada es que se ejecuta el constructor de la superclase, formando parte de la ejecución del constructor de la subclase.

Para que esta operación funcione, los parámetros necesarios para la inicialización de los campos de la superclase se pasan a su constructor como parámetros en la llamada a **`super`**.

Constructor de Superclase

El constructor de una ***subclase*** debe tener siempre como primera sentencia una invocación al constructor de su ***superclase***.

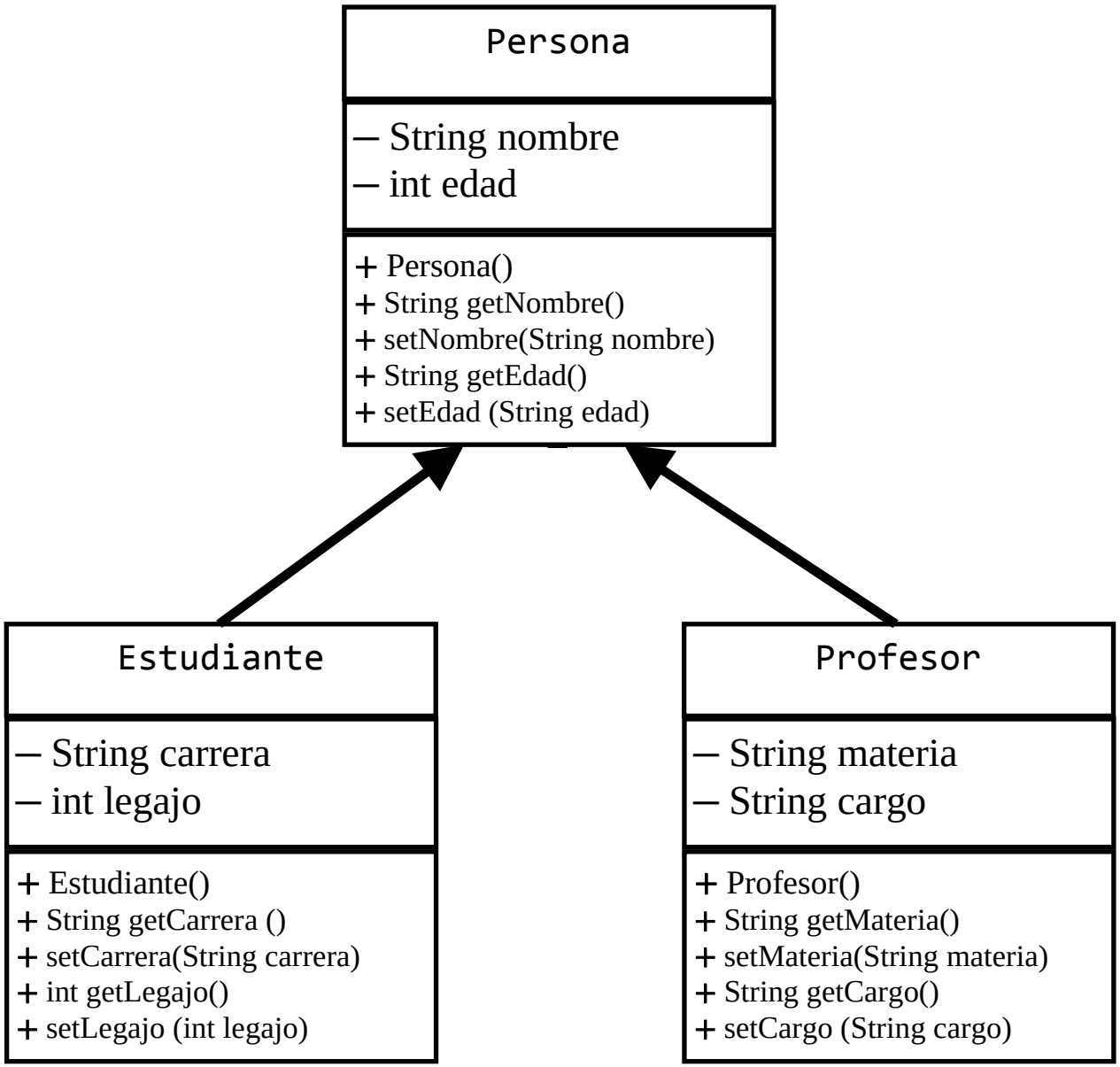
Si el código no incluye esta llamada, Java intentará insertarla automáticamente.

La inserción automática de la llamada a la superclase sólo funciona si la superclase tiene un constructor sin parámetros; en el caso contrario, Java informa un error.

En general, es una buena idea la de incluir siempre en los constructores llamadas explícitas a la superclase, aun cuando sea una llamada que el compilador puede generar automáticamente.

Consideramos que esta inclusión forma parte de un buen estilo de programación.

Evita la posibilidad de una mala interpretación y de confusión en el caso de que un lector no esté advertido de la generación automática de código.


Persona
– String nombre
– int edad
+ Persona()
+ String getNombre()
+ setNombre(String nombre)
+ String getEdad()
+ setEdad (String edad)
Estudiante
– String carrera
– int legajo
+ Estudiante()
+ String getCarrera ()
+ setCarrera(String carrera)
+ int getLegajo()
+ setLegajo (int legajo)
Profesor
– String materia
– String cargo
+ Profesor()
+ String getMateria()
+ setMateria(String materia)
+ String getCargo()
+ setCargo (String cargo)

```java
public class Persona{

    private String nombre;
    private int edad;

    public Persona(){ // Constructor por defecto
    }

    public int getEdad(){
        return edad;
    }

    public void setEdad(int edad){
        this.edad = edad;
    }

    public String getNombre(){
        return nombre;
    }

    public void setNombre(String nombre){
        this.nombre = nombre;
    }
}
```

```java
public class Estudiante extends Persona{

    private String carrera;
    private int legajo;

    public Estudiante(){
    }

    public String getCarrera(){
        return carrera;
    }

    public void setCarrera(String carrera){
        this.carrera = carrera;
    }

    public int getLegajo(){
        return legajo;
    }

    public void setLegajo(int legajo){
        this.legajo = legajo;
    }
}
```

```java
public class Profesor extends Persona{
    private String materia;
    private String cargo;

    public Profesor(){
    }

    public String getMateria(){
        return materia;
    }

    public void setMateria(String materia){
        this.materia = materia;
    }

    public String getCargo(){
        return cargo;
    }

    public void setCargo(String cargo){
        this.cargo = cargo;
    }
}
```

```java
public class PersonaApp {

    public static void main (String[] args) {

        Persona kuki = new Persona();
        kuki.setNombre("Pepe");
        kuki.setEdad(25);

        Estudiante chacho = new Estudiante();
        chacho.setNombre("Juan Jose");
        chacho.setLegajo(240682);
        chacho.setCarrera("Industriales");

        Profesor toni = new Profesor();
        toni.setNombre("Rivero");
        toni.setMateria("Fundamentos de Informatica");
        toni.setCargo("Titular");

        System.out.println("Objeto kuki de tipo Persona");
        System.out.printf("Nombre: %s, Edad: %d \n", kuki.getNombre(),kuki.getEdad());
        System.out.println("");

        System.out.println("Objeto chacho de tipo Estudiante");
        System.out.printf("Nombre: %s, Carrera: %s, Legajo: %d \n", chacho.getNombre(),
                          chacho.getCarrera(), chacho.getLegajo());
        System.out.println("");

        System.out.println("Objeto toni de tipo Profesor");
        System.out.printf("Nombre: %s, Materia: %s, Cargo: %s \n", toni.getNombre(),
                          toni.getMateria(), toni.getCargo());
        System.out.println("");
    }
```

instanceof

Este operador nos permite comprobar si un objeto es de una **clase concreta.**

Es decir si el objeto pasaría el test **ES-UN** para esa clase o interfaz, especificado a la derecha del operador.

Cuando utilicemos este operador, debemos recordar que sólo puede usarse con variables que contengan la referencia a un objeto.

Es decir, variables que contendrán un conjunto de bytes que representarán a la dirección en memoria en la que está almacenado el objeto.

Polimorfismo

Es uno de los cuatro pilares de la programación orientada a objetos junto con la ***Abstracción***, ***Encapsulación*** y ***Herencia.***

El polimorfismo es muy parecido o más bien tiene sus bases en la capacidad de **herencia** que presentan los lenguajes orientados a objetos.

En la herencia, las clases padres comparten métodos con las clases hijas.

Con el polimorfismo se hace prácticamente lo mismo pero en vez de clases padres se tiene ***clases abstractas***.

Las clases abstractas tienen métodos abstractos.

Solo están declarados sus nombres

Su forma de actuar difiere de una clase hija a otra.

Los métodos actúan dependiendo de la clase que haga mención del método declarado en la clase abstracta.

Clases Abstractas

La abstracción permite resaltar lo más representativo de algo sin importar los detalles.

La estructura es prácticamente igual, poseen nombre, atributos y métodos.

Para que una clase sea abstracta la condición es que ***al menos*** uno de sus métodos sea abstracto

Una clase Abstracta es similar a una clase normal.

Se agrega la palabra reservada **`abstract`** y no se especifica el cuerpo del método.

Características de una Clase Abstracta.

Una clase Abstracta no puede ser instanciada.

No se pueden crear objetos directamente, solo puede ser heredada.

Si al menos un método de la clase es **`abstract`**, obliga a que la clase completa sea definida **`abstract`**, sin embargo la clase puede tener el resto de métodos no abstractos.

Los ***métodos abstract*** no llevan cuerpo, no llevan los caracteres {}.

Todas las clases hijas que hereden de una clase ***abstract*** debe implementar obligatoriamente todos los métodos abstractos de la superclase.

La etiqueta "**`@Override`**" sirve para indicar en el código que estamos reescribiendo o especializando un método que se encuentra en la clase padre y que queremos redefinir en la clase hija

¿Cuándo Utilizarlas?

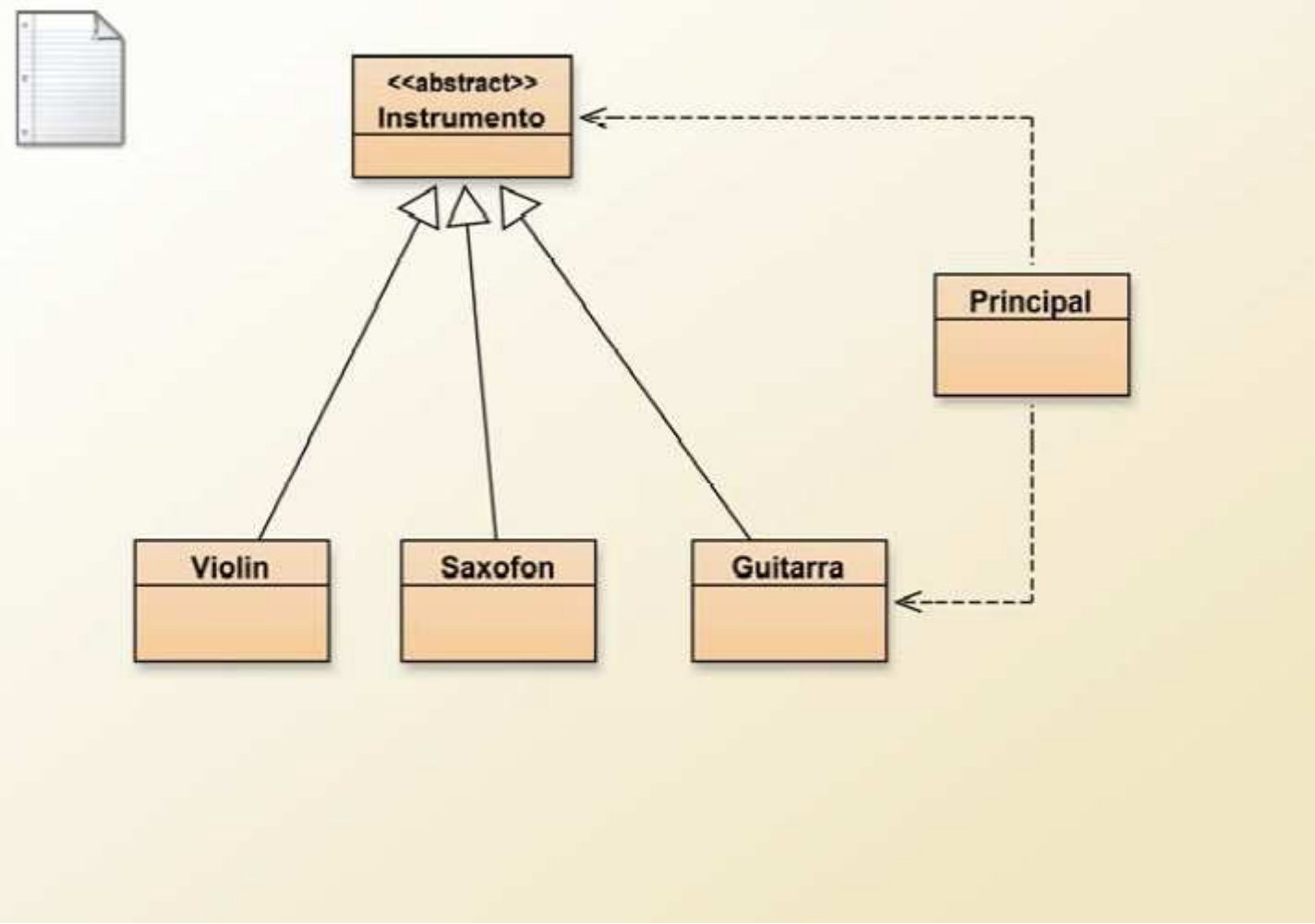
Al trabajar clases y métodos abstractos, no solo mantenemos nuestra aplicación más organizada y fácil de entender sino que también al no poder instanciar una clase abstracta nos aseguramos de que las propiedades específicas de esta, solo estén disponibles para sus clases hijas.

Con las ***Clases Abstractas*** lo que hacemos es definir un proceso general que luego será implementado por

las clases concretas que hereden dichas funcionalidades.

Es decir, si tengo una clase que hereda de otra Abstracta, estoy obligado a poner en el código, todos los métodos abstractos de la clase padre, pero esta vez serán métodos concretos y su funcionalidad o cuerpo será definido dependiendo de para que la necesite, de esa manera si tengo otra clase que también hereda del mismo padre, implementaré el mismo método pero con un comportamiento distinto.

Ejemplo Instrumentos



En el diagrama vemos una clase Abstracta **Instrumento**, posee una propiedad tipo String y un método abstracto **tocar**().

Vemos también las clases hijas **Guitarra**, **Saxofon** y **Violin**.

Todos los instrumentos musicales se pueden tocar, por ello creamos este método abstracto, ya que es un proceso común en todos los instrumentos sin importar el detalle de como se tocan, pues sabemos que una guitarra no se toca de la misma manera que el saxofón, así al heredar de la clase Instrumento, todas

sus clases hijas están obligadas a implementar este método y darle la funcionalidad que le corresponda.

Como vemos cada una de las clases concretas implementan el método **tocar()** y le dan la funcionalidad dependiendo de cómo se toque el instrumento.

También en cada constructor de las clases definimos el **tipo**, pero si nos fijamos bien en las clases concretas no tenemos la variable tipo declarada, pues estamos usando la variable heredada de la clase **Instrumento**.

Cuando trabajamos con clases Abstractas, estas solo pueden ser heredadas pero no instanciadas, esto quiere decir que no podemos crear objetos directamente de estas clases, con **new**.

Como vemos en la clase Principal tenemos la lógica para ejecutar nuestra aplicación y usamos el concepto de Polimorfismo para crear los objetos de tipo Instrumento por medio de sus clases Hijas, pero en ningún momento creamos un objeto como instancia directa de la clase abstracta.

Interfaces

La Interface avanza un paso el concepto de una clase abstracta.

Es una clase abstracta pura.

Todos los métodos son abstractos y no se implementa ninguno.

Permite al diseñador establecer la forma de una clase (nombres de métodos, listas de argumentos y tipos de retorno, pero no bloques de código).

No se pueden definir atributos salvo que estos sean estáticos o constantes, **`static`** o **`final`**.

Una interface sirve para establecer un ***protocolo*** entre clases.

Para crear una interface, se utiliza la palabra clave **`interface`** en lugar de **`class`**.

La interface puede definirse **`public`** o sin modificador de acceso, y tiene el mismo significado que para las clases.

Todos los métodos que declara una interface son siempre **`public`**.

Para indicar que una clase implementa los métodos de una interface se utiliza la palabra clave **`implements`**.

El compilador se encargará de verificar que la clase efectivamente declare e implemente todos los métodos de la interface.

Una clase puede implementar más de una interface.