7 Herencia

7 Herencia

1. Introducción
2. Modificadores de acceso
3. Sobrecarga y anulación de métodos
4. Super
5. Constructores
6. El método toString
7. Casting de objetos
8. instanceof
9. Modificador final
10. Polimorfismo
11. Clases abstractas
12. Interfaces
    - 12.1 Herencia entre interfaces
    - 12.2 Métodos por defecto
    - 12.3 Métodos estáticos
    - 12.4 Métodos privados
    - 12.5 Diferencias entre una interfaz y una clase abstracta
    - 12.6 Utilización de una interfaz como un tipo de datos
13. Clases anidadas
    - 13.1 Clases anidadas estáticas
    - 13.2 Clases anidadas no estáticas o clases internas
        - 13.2.1 Clases internas miembro
        - 13.2.2 Clases internas dentro de un método
        - 13.2.3 Clases internas dentro de un bloque
        - 13.2.4 Clases inline anónimas
14. Clases selladas
15. Interfaces selladas

1. Introducción

La herencia permite definir una clase tomando como base a otra clase ya existente. Dicha clase base se conoce como **superclase** o **clase padre** y la clase que hereda se denomina **subclase** o **clase hija**. Por lo tanto, una subclase es una versión especializada de una superclase ya que hereda tanto los atributos como los métodos definidos por la superclase y además añade los suyos propios.

Esto es una de las bases de la **reutilización** de código ya que cuando se quiere crear una clase nueva y ya existe una clase que incluye parte del código que queremos, podemos heredar nuestra nueva clase de la clase existente reutilizando los atributos y métodos de la misma. La herencia facilita el trabajo del programador porque permite crear clases estándar y a partir de ellas crear nuestras propias clases personales. Esto es más cómodo que tener que crear todas las clases desde cero.

Por ejemplo, si quisiéramos realizar una aplicación de vehículos, definiríamos una superclase o clase padre con lo común a todos los vehículos y luego definiríamos una subclase o clase hija para cada tipo de vehículo donde se añadiría lo particular de cada uno.

En java, la herencia se especifica en la subclase añadiendo la palabra **extends** seguida del nombre de la superclase. Por ejemplo, así sería para indicar que *Coche* es hija de *Vehiculo*:

```
public class Coche extends Vehiculo {//Coche es hija de Vehiculo
...
}
```

En Java, solamente se puede tener un padre pero puede haber varios niveles de herencia, es decir, clases hijas que a su vez son padres de otras clases. Por ejemplo, el *Coche* es hija de *Vehiculo* pero puede ser padre de otra clase, como por ejemplo de *Todoterreno*:

```
public class Todoterreno extends Coche {//Coche es hija de Vehiculo y padre de
Todoterreno
...
}
```

Si el padre tiene algún atributo estático, también lo pueden usar los hijos. `ClasePadre.AtributoEstático` y `ClaseHijo.AtributoEstático` acceden a la misma variable porque es el mismo atributo estático.

2. Modificadores de acceso

En el *Tema 4. Programación Orientada a Objetos 5 Modificadores de acceso* vimos los modificadores de acceso y cómo afectaban a la visibilidad. En este tema vamos a incorporar el modificador de acceso *protected* que es el que está pensado para la herencia.

Los modificadores de acceso afectan a la visibilidad y también afectan a la herencia. Visibilidad es lo que una clase puede ver de otra clase y herencia es lo que una clase hereda de otra clase. He aquí dos tablas con los modificadores de acceso, una para la visibilidad y otra para la herencia:

Tabla de visibilidad:

	Private	Friendly	Protected	Public
Misma clase	X	X	X	X
Mismo paquete		X	X	X
Otro paquete				X
Subclase en el mismo paquete		X	X	X
Subclase en distinto paquete				X

Tabla de herencia:

	Private	Friendly	Protected	Public
Subclase en el mismo paquete		X	X	X

	Private	Friendly	Protected	Public
Subclase en distinto paquete			X	X

Si las clases están en un subpaquete, a efectos de visibilidad y herencia se considera que están en otro paquete.

Las conclusiones que se pueden obtener a partir de las dos tablas son las siguientes:

- La visibilidad es la misma independientemente de que la clase sea hija o no.
- En herencia, siempre se hereda el *protected* independientemente del paquete donde se encuentre la clase hija. Por lo tanto, cuando diseñemos una clase que vaya a tener descendientes, es conveniente declarar sus atributos como *protected*.

Un atributo *private* no se hereda pero si los *getters* y *setters* del padre tienen un modificador distinto de *private*, sí puede el hijo utilizar dicho atributo a través de dichos métodos. Pero no es conveniente programar de esta manera, es más adecuado utilizar el modificador *protected*.

Veamos un ejemplo de herencia con el uso del modificador *protected*:

```
package tema7_Herencia.modificadorProtected;

public class Vehicle { //Superclase o clase padre

    protected int wheelCount; //Atributos protected
    protected double speed;
    protected String colour;

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

}
```

```java
package tema7_Herencia.modificadorProtected;

public class Car extends Vehicle { //Car es hijo de Vehicle

    private double gasoline;//Atributo propio de Car

    //Car hereda de Vehicle los atributos wheelCount,speed y colour ya que son
protected
    //También hereda todos los métodos de Vehicle ya que son public

    public double getGasoline() { //Método propio de Car
        return gasoline;
    }

    public void refuel(double liters) {//Método propio de Car
        gasoline += liters;
    }

}
```

```java
package tema7_Herencia.modificadorProtected;

public class Main {

    public void showProtected() {

        Car car;
        car = new Car();

        car.accelerate(100); //Método heredado
        System.out.printf("La velocidad del coche es %.2f km/h", car.getSpeed());

        car.refuel(40.35); //Método propio
        System.out.printf("\nEl coche tiene %.2f litros de gasolina",
car.getGasoline());

    }

    public static void main(String[] args) {

        new Main().showProtected();

    }

}
```

Salida por consola:

```
La velocidad del coche es 100,00 km/h
El coche tiene 40,35 litros de gasolina
```

3. Sobrecarga y anulación de métodos

Se puede **sobrecargar** un método heredado para proporcionar una versión del mismo adaptado a las necesidades de la subclase. Ejemplo:

```
package tema7_Herencia.sobrecarga;

public class Car extends Vehicle {

    private double gasoline;

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

    public void accelerate() { //Sobrecarga del método heredado accelerate
        speed += 10;
    }

}
```

```
package tema7_Herencia.sobrecarga;

public class Main {

    public void showOverload() {

        Car car;
        car = new Car();

        car.accelerate(100);//Método heredado
        System.out.printf("La velocidad del coche es %.2f km/h", car.getSpeed());

        car.accelerate();//Método sobrecargado
        System.out.printf("\nLa velocidad del coche es %.2f km/h",
car.getSpeed());

    }

    public static void main(String[] args) {

        new Main().showOverload();

    }

}
```

Salida por consola:

```
La velocidad del coche es 100,00 km/h
La velocidad del coche es 110,00 km/h
```

Si la subclase define un método con la misma firma que un método heredado, entonces **anula** o **sobrescribe** el método de la superclase. Veamos un ejemplo donde el coche ha anulado el método heredado de acelerar para añadirle el consumo de gasolina producido por la aceleración. Vamos a considerar que se gasta de gasolina un 1% de la cantidad que se acelera:

```
package tema7_Herencia.anulacionMetodos;

public class Car extends Vehicle {

    private double gasoline;

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

    @Override
    public void accelerate(double amount) {//Anula el método heredado accelerate
        speed += amount;
        gasoline -= amount * 0.01;//Se consume de gasolina un 1% de amount
    }

}
```

```
package tema7_Herencia.anulacionMetodos;

public class Main {

    public void showOverridde() {

        Car car;
        car = new Car();

        car.refuel(40.35);
        System.out.printf("El coche tiene %.2f litros de gasolina",
car.getGasoline());
        car.accelerate(100);//Método que ha anulado al heredado
        System.out.printf("\nEl coche tiene %.2f litros de gasolina",
car.getGasoline());

    }

    public static void main(String[] args) {

        new Main().showOverridde();

    }
```

```
}
```

Salida por consola:

```
El coche tiene 40,35 litros de gasolina
El coche tiene 39,35 litros de gasolina
```

En la clase *Car*, se puede observar la anotación *@Override* antes de la firma del método *accelerate*.

Las anotaciones de Java comienzan con @ y permiten incrustar información suplementaria en un programa para que pueda ser utilizada por varias herramientas.

La anotación *@Override* le indica al compilador que el método debe sobrescribir un método de la superclase. Si no lo hace, el compilador generará un error. Se utiliza para asegurar que un método de superclase esté anulado, y no simplemente sobrecargado. Es una manera de comprobar en tiempo de compilación que se está anulando correctamente un método, y de este modo evitar errores en tiempo de ejecución los cuales serían mucho más difíciles de detectar.

La visibilidad de lo que se hereda es con respecto al paquete de la superclase, no con respecto al paquete de la subclase. Si no interesa, la subclase tendrá que sobrescribir lo heredado aunque no haga ningún cambio para que la visibilidad sea con respecto al paquete de la subclase. Veamos un ejemplo donde *Vehicle* está en un paquete distinto que *Car*. *Vehicle* tiene el método *accelerate* como *protected*, por lo tanto *Car* lo hereda aunque esté en otro paquete como podemos observar en la tabla de herencia. La clase *Main* se encuentra en el mismo paquete que *Car* y quiere acceder al método *accelerate* del mismo. Pero la visibilidad de *accelerate* es con respecto al paquete de *Vehicle* ya que la visibilidad de lo que se hereda es con respecto al paquete de la superclase, no con respecto al paquete de la subclase. Dicho método es *protected* y si nos fijamos en la tabla de visibilidad, un *protected* no es visible desde otro paquete, por lo que no se le va a permitir dando un error de compilación:

```
package tema7_Herencia.visibilidad1;

public class Vehicle {

    protected int wheelCount;
    protected double speed;
    protected String colour;

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }
```

```
    protected void accelerate(double amount) { //Método protected
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

}
```

```
package tema7_Herencia.visibilidad2;

import tema7_Herencia.visibilidad1.Vehicle;

public class Car extends Vehicle {

    private double gasoline;

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

}
```

```
package tema7_Herencia.visibilidad2;

public class Main {

    public void showVisibility() {

        Car car;
        car = new Car();

        car.accelerate(100);//Error de compilación: el método accelerate de
Vehicle no es visible
        System.out.printf("La velocidad del coche es %.2f km/h", car.getSpeed());

        car.refuel(40.35);
        System.out.printf("\nEl coche tiene %.2f litros de gasolina",
car.getGasoline());

    }

    public static void main(String[] args) {

        new Main().showVisibility();

    }

}
```

La solución es que *Car* anule el método *accelerate* heredado de *Vehicle* para que la visibilidad de dicho método sea con respecto al paquete de *Car*. *Car* no va a realizar ningún cambio en dicho método, es decir, lo va a anular para dejarlo exactamente igual, pero de esta forma modifica el paquete para la visibilidad:

```
package tema7_Herencia.visibilidad3;

import tema7_Herencia.visibilidad1.Vehicle;

public class Car extends Vehicle {

    private double gasoline;

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

    @Override
    protected void accelerate(double amount) { //Anula el método heredado y lo
deja exactamente igual
        speed += amount;
    }

}
```

```
package tema7_Herencia.visibilidad3;

public class Main {

    public void showVisibility() {

        Car car;
        car = new Car();

        car.accelerate(100);//No hay error de compilación, el método ya es
visible
        System.out.printf("La velocidad del coche es %.2f km/h", car.getSpeed());

        car.refuel(40.35);
        System.out.printf("\nEl coche tiene %.2f litros de gasolina",
car.getGasoline());

    }

    public static void main(String[] args) {

        new Main().showVisibility();

    }
```

```
}
```

Salida por consola:

```
La velocidad del coche es 100,00 km/h
El coche tiene 40,35 litros de gasolina
```

Pero, ¿qué ocurriría si la clase *Main* estuviera en un paquete distinto a *Car*? Daría un error de compilación porque un *protected* no es visible desde otro paquete:

```
package tema7_Herencia.visibilidad4;

import tema7_Herencia.visibilidad1.Vehicle;

public class Car extends Vehicle {

    private double gasoline;

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

    @Override
    protected void accelerate(double amount) { //Anula el método heredado y lo
deja exactamente igual
        speed += amount;
    }

}
```

```
package tema7_Herencia.visibilidad5;

import tema7_Herencia.visibilidad4.Car;

public class Main {

    public void showVisibility() {

        Car car;
        car = new Car();

        car.accelerate(100);//Error de compilación: el método accelerate de Car
no es visible
        System.out.printf("La velocidad del coche es %.2f km/h", car.getSpeed());

        car.refuel(40.35);
        System.out.printf("\nEl coche tiene %.2f litros de gasolina",
car.getGasoline());

    }
```

```
    public static void main(String[] args) {

        new Main().showVisibility();

    }

}
```

La solución sería que *Car* cambiara la visibilidad del método. Si una subclase quiere anular algún método de la superclase para cambiar la visibilidad, se permite únicamente si amplia la visibilidad, no si la reduce. La escala de valores de más restrictivo a menos es: *private*, *friendly*, *protected* y *public*. Por ejemplo, no se puede cambiar de *protected* a *friendly* pero sí al revés.

Solucionemos el ejemplo anterior para que la clase *Main* pueda acceder al método *accelerate* de *Car*. Si nos fijamos en la tabla de visibilidad, el único modificador que nos permite visibilidad desde otro paquete es *public*. Entonces, tendríamos que cambiar el *protected* a *public* y se permite porque se amplia la visibilidad, no se reduce:

```
package tema7_Herencia.visibilidad6;

import tema7_Herencia.visibilidad1.Vehicle;

public class Car extends Vehicle {

    private double gasoline;

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

    @Override
    //Anula el método heredado cambiando el modificador de acceso de protected a
public
    public void accelerate(double amount) {
        speed += amount;
    }

}
```

```
package tema7_Herencia.visibilidad7;

import tema7_Herencia.visibilidad6.Car;

public class Main {

    public void showVisibility() {

        Car car;
        car = new Car();
```

```
        car.accelerate(100);//No hay error de compilación, el método ya es
visible
        System.out.printf("La velocidad del coche es %.2f km/h", car.getSpeed());

        car.refuel(40.35);
        System.out.printf("\nEl coche tiene %.2f litros de gasolina",
car.getGasoline());

    }

    public static void main(String[] args) {

        new Main().showVisibility();

    }

}
```

Salida por consola:

```
La velocidad del coche es 100,00 km/h
El coche tiene 40,35 litros de gasolina
```

4. Super

A veces se requiere llamar a un método de la superclase. Eso se realiza con la palabra reservada **super**. En el *Tema 4. Programación Orientada a Objetos 14 This* vimos que *this* es una variable que hace referencia al objeto actual, pues *super* es una variable que hace referencia a la superclase del objeto actual, por lo tanto es un método imprescindible para poder acceder a métodos anulados por herencia.

```
package tema7_Herencia.usoSuper;

public class Car extends Vehicle {

    private double gasoline;

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

    @Override
    public void accelerate(double amount) {
        super.accelerate(amount);//Uso de super para llamar al método del padre
        gasoline -= amount * 0.01;
    }

}
```

```
package tema7_Herencia.usoSuper;

public class Main {

    public void showUsoSuper() {

        Car car;
        car = new Car();

        car.refuel(40.35);
        System.out.printf("El coche tiene %.2f litros de gasolina",
car.getGasoline());
        System.out.printf(" y va a %.2f km/h", car.getSpeed());
        car.accelerate(100);
        System.out.printf("\nEl coche tiene %.2f litros de gasolina",
car.getGasoline());
        System.out.printf(" y va a %.2f km/h", car.getSpeed());

    }

    public static void main(String[] args) {

        new Main().showUsoSuper();

    }

}
```

Salida por consola:

```
El coche tiene 40,35 litros de gasolina y va a 0,00 km/h
El coche tiene 39,35 litros de gasolina y va a 100,00 km/h
```

En el ejemplo anterior, *super.accelerate(amount)* llama al método *accelerate* de la clase *Vehicle* el cual acelerará la marcha. Es necesario redefinir el método *accelerate* en la clase *Car* ya que aunque la velocidad varía igual que en la superclase, hay que tener en cuenta el consumo de gasolina.

5. Constructores

Los constructores no se heredan de la superclase a las subclases pero sí se pueden invocar los constructores de la superclase desde los constructores de las subclases mediante *super*.

```
package tema7_Herencia.constructores1;

public class Vehicle {

    protected int wheelCount;
    protected double speed;
    protected String colour;

    public Vehicle(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
```

```
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

}
```

```
package tema7_Herencia.constructores1;

public class Car extends Vehicle {

    private double gasoline;

    public Car(int wheelCount, String colour) {
        super(wheelCount, colour);//Uso de super para llamar al constructor de la
superclase
        gasoline = 0;
    }

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

    @Override
    public void accelerate(double amount) {
        super.accelerate(amount);
        gasoline -= amount * 0.01;
    }

}
```

```
package tema7_Herencia.constructores1;

public class Main {

    public void showConstructors1() {

        Vehicle vehicle;
        Car car;

        vehicle = new Vehicle(2, "azul");
        System.out.printf("Vehículo: %d ruedas y de color %s",
vehicle.getWheelCount(), vehicle.getColour());

        car = new Car(4, "rojo");
        System.out.printf("\nCoche: %d ruedas y de color %s",
car.getWheelCount(), car.getColour());

    }

    public static void main(String[] args) {

        new Main().showConstructors1();

    }

}
```

Salida por consola:

```
Vehículo: 2 ruedas y de color azul
Coche: 4 ruedas y de color rojo
```

Si una clase no tiene constructor, Java crea uno por defecto. Pero en el caso de que sea una subclase, Java lo crea con la línea de código *super()*, es decir, con una llamada al constructor de la superclase.

```
package tema7_Herencia.constructores2;

public class Vehicle {

    protected int wheelCount;
    protected double speed;
    protected String colour;

    //No tiene constructor, por lo que Java lo crea por defecto: Vehicle()

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }
```

```
    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

}
```

```
package tema7_Herencia.constructores2;

public class Car extends Vehicle {

    private double gasoline;
/*
 * No tiene constructor. Java lo crea por defecto con la línea de
 * código super(), es decir, con una llamada al constructor por
 * defecto de la superclase. Por lo tanto, está llamando a Vehicle().
 */

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

    @Override
    public void accelerate(double amount) {
        super.accelerate(amount);
        gasoline -= amount * 0.01;
    }

}
```

```
package tema7_Herencia.constructores2;

public class Main {

    public void showConstructors2() {

        Vehicle vehicle;
        Car car;
```

```
        vehicle = new Vehicle();//Constructor creado por Java
        car = new Car();//Constructor creado por Java que contiene super()
        vehicle.accelerate(3);
        car.accelerate(100);
        System.out.printf("El vehículo va a %.2f km/h", vehicle.getSpeed());
        System.out.printf("\nEl coche va a %.2f km/h", car.getSpeed());

    }

    public static void main(String[] args) {

        new Main().showConstructors2();

    }

}
```

Salida por consola:

```
El vehículo va a 3,00 km/h
El coche va a 100,00 km/h
```

Pero si la superclase tuviera un constructor con parámetros, Java ya no crearía el constructor por defecto *Vehicle()* y a las subclases les daría un error de compilación con la llamada del *super()*:

```
package tema7_Herencia.constructores3;

public class Vehicle {

    protected int wheelCount;
    protected double speed;
    protected String colour;

    /*
     * Tiene un constructor con parámetros, por lo tanto, Java no crea
     * el constructor por defecto Vehicle()
     */
    public Vehicle(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }
```

```
    public void setColour(String colour) {
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

}
```

```
package tema7_Herencia.constructores3;

public class Car extends Vehicle {

    private double gasoline;

/*
 * No tiene constructor. Java lo crea por defecto con la línea de
 * código super(), es decir, con una llamada al constructor por
 * defecto de la superclase. Por lo tanto, está llamando a Vehicle(),
 * dando un error de compilación ya que no se ha creado.
 */

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

    @Override
    public void accelerate(double amount) {
        super.accelerate(amount);
        gasoline -= amount * 0.01;
    }

}
```

Incluso aunque se defina un constructor con parámetros en la clase *Car*, Java sigue añadiendo de manera implícita la llamada *super()* en el constructor y como no hay constructor por defecto en la clase *Vehicle*, continúa dando error de compilación.

```
package tema7_Herencia.constructores4;

public class Car extends Vehicle {

    private double gasoline;
```

```
    /*
     * Java sigue añadiendo de manera implícita la llamada super()
     * en el constructor y como no hay constructor por defecto en
     * la clase Vehicle, da un error de compilación
     */
    public Car(int wheelCount, String colour) { //Error de compilación
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
        gasoline = 0;
    }

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

    @Override
    public void accelerate(double amount) {
        super.accelerate(amount);
        gasoline -= amount * 0.01;
    }

}
```

Se puede arreglar realizando una llamada explícita al constructor con parámetros de la superclase:

```
package tema7_Herencia.constructores5;

public class Car extends Vehicle {

    private double gasoline;

    public Car(int wheelCount, String colour) {
        super(wheelCount, colour);//Está llamando al constructor con parámetros
de Vehicle
        gasoline = 0;
    }

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

    @Override
    public void accelerate(double amount) {
        super.accelerate(amount);
        gasoline -= amount * 0.01;
```

```
    }

}
```

Si una subclase tiene más constructores, este problema habría que arreglarlo en cada uno de ellos:

```
package tema7_Herencia.constructores6;

public class Car extends Vehicle {

    private double gasoline;

    public Car(int wheelCount, String colour) {
        super(wheelCount, colour);
        gasoline = 0;
    }

    /*
     * Java sigue añadiendo de manera implícita la llamada super()
     * en el constructor y como no hay constructor por defecto en
     * la clase Vehicle, da un error de compilación
     */
    public Car(int wheelCount) { //Error de compilación
        this.wheelCount = wheelCount;
        this.colour = "blanco";
        speed = 0;
        gasoline = 0;
    }

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

    @Override
    public void accelerate(double amount) {
        super.accelerate(amount);
        gasoline -= amount * 0.01;
    }

}
```

Se puede solucionar de dos maneras:

1. Haciendo una llamada a un constructor de la superclase mediante super.
2. Haciendo una llamada a un constructor de la propia clase mediante this.

En ambos casos, tienen que ser la primera instrucción del constructor, por lo que el uso de super y this no puede ser simultáneo, lo que significa que hay que elegir entre ambas.

Veamos en código las dos soluciones:

1. Haciendo una llamada a un constructor de la superclase mediante super:

```
package tema7_Herencia.constructores7;

public class Car extends Vehicle {

    private double gasoline;

    public Car(int wheelCount, String colour) {
        super(wheelCount, colour);
        gasoline = 0;
    }

    public Car(int wheelCount) {
        super(wheelCount, "blanco");//Está llamando al constructor de Vehicle
mediante super
        gasoline = 0;
    }

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

    @Override
    public void accelerate(double amount) {
        super.accelerate(amount);
        gasoline -= amount * 0.01;
    }

}
```

2. Haciendo una llamada a un constructor de la propia clase mediante this.

```
package tema7_Herencia.constructores8;

public class Car extends Vehicle {

    private double gasoline;

    public Car(int wheelCount, String colour) {
        super(wheelCount, colour);
        gasoline = 0;
    }

    public Car(int wheelCount) {
        this(wheelCount, "blanco");//Está llamando al otro constructor de Car
mediante this
    }

    public double getGasoline() {
        return gasoline;
```

```
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

    @Override
    public void accelerate(double amount) {
        super.accelerate(amount);
        gasoline -= amount * 0.01;
    }

}
```

6. El método toString

La clase *Object* es la clase raíz de todo el árbol de la jerarquía de clases Java, es decir, es una superclase implícita de todas las demás clases. En otras palabras, todas las demás clases son subclases de *Object*. Esto significa que una variable de referencia de tipo *Object* puede referirse a un objeto de cualquier otra clase.

La clase *Object* proporciona un cierto número de métodos de utilidad general que pueden utilizar todos los objetos ya que los heredan. Pero normalmente hay que sobrescribirlos para que funcionen adecuadamente adaptándolos a la clase correspondiente. Esto se hace con la idea de que todas las clases utilicen el mismo nombre y prototipo de método para hacer operaciones comunes. Como por ejemplo, `toString()` que se utiliza para obtener una cadena de texto que represente al objeto. El método *toString()* de la clase *Object* devuelve una cadena que consiste en el nombre de la clase del objeto, el carácter arroba '@' y la representación hexadecimal sin signo del código hash del objeto. Siempre se recomienda sobrescribir el método *toString()* para obtener nuestra propia representación del objeto.

```
package tema7_Herencia.toString;

public class Vehicle {

    protected int wheelCount;
    protected double speed;
    protected String colour;

    public Vehicle(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
```

```
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

    @Override
    public String toString() {//Se anula el toString de Object
        return String.format("Número de ruedas: %d, color: %s, velocidad: %.2f",
wheelCount, colour, speed);
    }

}
```

```
package tema7_Herencia.toString;

public class Car extends Vehicle {

    private double gasoline;

    public Car(int wheelCount, String colour) {
        super(wheelCount, colour);
        gasoline = 0;
    }

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

    @Override
    public void accelerate(double amount) {
        super.accelerate(amount);
        gasoline -= amount * 0.01;
    }

    @Override
    public String toString() {//Se anula el toString de Vehicle y se llama al
super.toString
        return String.format("%s, gasolina: %.2f", super.toString(), gasoline);
    }

}
```

```
package tema7_Herencia.toString;

public class Main1 {

    public void showToString() {

        Vehicle vehicle;
        Car car;
        vehicle = new Vehicle(2, "azul");
        vehicle.accelerate(200);
        System.out.println(vehicle.toString());
        car = new Car(4,"rojo");
        car.refuel(100);
        car.accelerate(45);
        System.out.println(car.toString());

    }

    public static void main(String[] args) {

        new Main1().showToString();

    }

}
```

Salida por consola:

```
Número de ruedas: 2, color: azul, velocidad: 200,00
Número de ruedas: 4, color: rojo, velocidad: 45,00, gasolina: 99,55
```

Cuando se muestra por consola un objeto y solo se pone el nombre de la variable del objeto, se llama al toString del objeto:

```
package tema7_Herencia.toString;

public class Main2 {

    public void showToString() {

        Vehicle vehicle;
        Car car;
        vehicle = new Vehicle(2, "azul");
        vehicle.accelerate(200);
        System.out.println(vehicle);//Se llama al toString del objeto
        car = new Car(4,"rojo");
        car.refuel(100);
        car.accelerate(45);
        System.out.println(car);//Se llama al toString del objeto

    }

    public static void main(String[] args) {
```

```
        new Main2().showToString();

    }

}
```

7. Casting de objetos

Como ocurre con los tipos primitivos, también es posible realizar casting entre objetos siempre y cuando esté dentro de la estructura jerárquica en su herencia.

Para crear un objeto, hay que declarar una variable cuyo tipo es una clase:

```
Vehicle vehicle; //vehicle es una variable de tipo Vehicle
```

Eso significa que *vehicle* es de tipo *Vehicle* y que nunca va a cambiar de tipo, siempre va a ser de tipo *Vehicle*.

Creemos ahora una variable de tipo *Car*:

```
Car car; //car es una variable de tipo Car
```

Lo mismo que antes, *car* es una variable de tipo *Car* y nunca va a cambiar de tipo, siempre va a ser de tipo *Car*.

A la variable *vehicle* se le puede asignar un objeto de tipo *Vehicle* y también se le pueden asignar objetos cuyo tipo pertenezca a cualquiera de sus descendientes (hijos, nietos, bisnietos, tataranietos, etc):

```
vehicle = car; //A vehicle se le está asignando un objeto de tipo Car, que es una subclase de Vehicle
```

La variable *vehicle* contiene un objeto de tipo *Car* pero su tipo es *Vehicle*, entonces solamente podrá acceder a los atributos y métodos de *Vehicle*:

```
vehicle.refuel(50);//Error de compilación: el método refuel no está definido para el tipo Vehicle
```

La variable *vehicle* no puede acceder al método *refuel* porque dicho método está definido en la clase *Car*. Es decir, una variable padre puede contener un objeto de tipo hijo pero solamente podrá acceder a los atributos y métodos definidos en el padre.

¿y si la asignación la hiciéramos al revés, es decir, a *car* le asignamos *vehicle*?

```
car = vehicle;//Error de compilación: falta de coincidencia de tipos, no puede convertir de Vehicle a Car
```

Daría un error de coincidencia de tipos, pero se podría solucionar con un casting:

```
car = (Car) vehicle;//Solucionado con un casting ya que vehicle contiene un Car
```

Hay que tener en cuenta que para que el casting funcione, la variable *vehicle* debe contener un objeto de tipo *Car*, porque si no, dará un error de ejecución *ClassCastException*.

```
vehicle = new Vehicle(2, "blanco");
car = (Car) vehicle; //Error de ejecución: ClassCastException ya que vehicle no contiene un objeto de tipo Car
```

Veamos en el siguiente código todos los ejemplos anteriores:

```
package tema7_Herencia.castingObjetos;

public class Main {

    public void showObjectCasting() {

        Vehicle vehicle;
        Car car;

        car = new Car(4, "rojo");
        System.out.printf("Coche: %d ruedas y de color %s", car.getWheelCount(), car.getColour());

        vehicle = car;//A vehicle se le está asignando un objeto de tipo Car, que es una subclase de Vehicle
        System.out.printf("\nCoche: %d ruedas y de color %s", vehicle.getWheelCount(), vehicle.getColour());
        vehicle.refuel(50);//Error de compilación: el método refuel no está definido para el tipo Vehicle

        car = vehicle;//Error de compilación: falta de coincidencia de tipos, no puede convertir de Vehicle a Car
        car = (Car) vehicle;//Solucionado con un casting ya que vehicle contiene un Car

        vehicle = new Vehicle(2, "blanco");
        car = (Car) vehicle; //Error de ejecución: ClassCastException ya que vehicle no contiene un objeto de tipo Car
    }

    public static void main(String[] args) {

        new Main().showObjectCasting();

    }

}
```

8. instanceof

El operador *instanceof* permite comprobar si un determinado objeto pertenece a una clase concreta. Se utiliza de esta forma:

```
objeto instanceof clase
```

Devuelve *true* si el objeto pertenece a dicha clase.

```
package tema7_Herencia.instanceofObjetos;

public class Main {

    public void showInstanceof() {

        Vehicle vehicle;
        Car car;

        vehicle = new Vehicle(2, "azul");
        car = new Car(4, "rojo");
        System.out.println(vehicle instanceof Vehicle);//true
        System.out.println(vehicle instanceof Car);//false
        System.out.println(car instanceof Car);//true
        System.out.println(car instanceof Vehicle);//true
        vehicle = car;
        System.out.println(vehicle instanceof Car);//true

    }

    public static void main(String[] args) {

        new Main().showInstanceof();

    }

}
```

Podemos observar en el ejemplo lo siguiente:

- *car* también devuelve *true* con *Vehicle* ya que los objetos de las subclases también devuelven *true* con la superclase.
- `vehicle instanceof Car` devuelve *true* solamente si el objeto que contiene es un *Car*.

No es aconsejable programar en una superclase de esta forma:

```
public class Vehicle {

    ...

    public void accelerate(double amount) {

        if(this instanceof Car){//No es aconsejable esta manera de programar
            ...
        }
        else if(this instanceof Bike){
            ...
        }
    }

    ...
```

```
}
```

El motivo es porque una superclase puede tener muchísima descendencia (hijos, nietos, etc). Si en dicha superclase se hace referencia a la descendencia, cada vez que se cree un descendiente nuevo o que haya algún tipo de modificación en alguno, hay que estar modificando el padre. Es decir, en el padre no debe existir ningún tipo de referencia hacia su posible descendencia para que su código sea independiente.

En Java 14 se introduce una mejora en el instanceof para evitar el casting explícito después de comprobar el tipo de una variable. Con este cambio, Java automáticamente realiza el casting si la comprobación es verdadera. Esto se logra utilizando lo que se llama **Pattern Matching** para `instanceof`. Veamos un ejemplo de comparación de cómo se hacía antes de Java 14 y cómo se puede hacer a partir de Java 14:

```
package tema7_Herencia.instanceofObjetos;

public class Main2 {

    public void showInstanceof() {

        Vehicle vehicle = new Car(4, "rojo");
        Car car;

        if(vehicle instanceof Car) {//Antes de Java 14
            car = (Car) vehicle;
            car.refuel(50);
            System.out.println(car);
        }

        if(vehicle instanceof Car car2) {//A partir de Java 14
            car2.refuel(100);
            System.out.println(car2);
        }

    }

    public static void main(String[] args) {

        new Main2().showInstanceof();

    }

}
```

Salida por consola:

```
Número de ruedas: 4, color: rojo, velocidad: 0,00, gasolina: 50,00
Número de ruedas: 4, color: rojo, velocidad: 0,00, gasolina: 150,00
```

La variable `car3` solo existe dentro del bloque `if`, ni siquiera se puede usar en un bloque `else`:

```
package tema7_Herencia.instanceofObjetos;
```

```
public class Main3 {

    public void showInstanceof() {

        Vehicle vehicle = new Car(4, "rojo");

        if(vehicle instanceof Car car) {
            car.refuel(100);
            System.out.println(car);
        }else {
            car.refuel(50);//Error de compilación
        }

    }

    public static void main(String[] args) {

        new Main3().showInstanceof();

    }

}
```

Se puede combinar con operadores lógicos:

```
package tema7_Herencia.instanceofObjetos;

public class Main4 {

    public void showInstanceof() {

        Vehicle vehicle = new Car(4, "rojo");

        if(vehicle instanceof Car car && car.getGasoline()==0) {
            car.refuel(100);
            System.out.println(car);
        }

    }

    public static void main(String[] args) {

        new Main4().showInstanceof();

    }

}
```

9. Modificador final

El modificador *final* tiene varios usos en función de dónde se utilice:

- Delante de una variable en su declaración, crea una constante. La constante puede recibir el valor en tiempo de compilación o en tiempo de ejecución.

Ejemplo de constante en tiempo de compilación:

```
final double PI=3.141591;
```

Ejemplo de constante en tiempo de ejecución:

```
public class MyClass {

    private final int NUMBER;

    public MyClass(int n) {
        NUMBER = n;
    }

}
```

La constante *NUMBER* recibe el valor en la construcción del objeto.

- Delante de una variable que referencia a un objeto: dicha variable no puede referenciar a otro objeto.

```
final Car car = new Car(4, "rojo");
car = new Car(4, "blanco");//Error: la variable car no puede referenciar a
otro objeto
```

- En los parámetros de un método: el valor del parámetro no puede cambiar dentro del método.

```
public void refuel(final double liters) {
    liters++;//Error: no se puede modificar liters
    gasoline += liters;
}
```

- En la declaración de un método: dicho método no se puede anular por las subclases:

```
public class Vehicle {
    ...
    final public void accelerate(double amount) {//Este método no puede ser
anulado por las subclases
        speed += amount;
    }
    ...
}
```

```
public class Car extends Vehicle {
    ...
    @Override
    public void accelerate(double amount) {//No se puede anular el método
accelerate de Vehicle
        super.accelerate(amount);
        gasoline -= amount * 0.01;
    }
    ...
}
```

- En la definición de una clase: significa que esa clase no puede tener descendencia.

Veamos en el siguiente código algunos de los ejemplos anteriores:

```
package tema7_Herencia.usosFinal;

public class Vehicle {

    protected int wheelCount;
    protected double speed;
    protected String colour;

    public Vehicle(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    final public void accelerate(double amount) {//Este método no puede ser
anulado por las subclases
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

}
```

```java
package tema7_Herencia.usosFinal;

public class Car extends Vehicle {

    private double gasoline;

    public Car(int wheelCount, String colour) {
        super(wheelCount, colour);
        gasoline = 0;
    }

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(final double liters) {
        liters++;//Error: no se puede modificar liters
        gasoline += liters;
    }

    @Override
    public void accelerate(double amount) {//No se puede anular el método
accelerate de Vehicle
        super.accelerate(amount);
        gasoline -= amount * 0.01;
    }

}
```

```java
package tema7_Herencia.usosFinal;

public class Main {

    public void showUsosFinal() {

        final Car car = new Car(4, "rojo");
        car = new Car(4, "blanco");//Error: la variable car no puede referenciar
a otro objeto

    }

    public static void main(String[] args) {

        new Main().showUsosFinal();

    }

}
```

10. Polimorfismo

Polimorfismo es la capacidad de un objeto de adquirir varias formas.

La sobrecarga de métodos es un tipo de **polimorfismo estático** porque se resuelve en tiempo de compilación el método apropiado a ser llamado basado en la lista de argumentos.

La anulación de métodos es un tipo de **polimorfismo dinámico** porque se resuelve en tiempo de ejecución atendiendo al tipo del objeto. Con una variable de tipo padre, si se utiliza un objeto del padre para invocar al método, entonces se ejecutará el método de la clase padre, pero si se utiliza un objeto de la clase hija para invocar al método, entonces se ejecutará el método de la clase hija.

Veamos un ejemplo de polimorfismo dinámico:

```
package tema7_Herencia.polimorfismoDinamico;

public class Main {

    public void showDynamicPolymorphism() {

        Vehicle vehicle;//Variable de tipo padre

        vehicle = new Vehicle(2, "azul");//Objeto del padre
        vehicle.accelerate(100.39);//Método del padre
        System.out.printf("La velocidad del vehículo es %.2f km/h",
vehicle.getSpeed());

        vehicle = new Car(4, "rojo");//Objeto del hijo
        vehicle.accelerate(50.89);//Método del hijo
        System.out.printf("\nLa velocidad del coche es %.2f km/h",
vehicle.getSpeed());

    }

    public static void main(String[] args) {

        new Main().showDynamicPolymorphism();

    }

}
```

11. Clases abstractas

Cuando se crea una estructura con herencia, puede darse el caso de que algún método del padre no se pueda implementar porque los detalles de la implementación dependan de cada uno de los hijos. Entonces, dicho método se declara como abstracto en el padre y solamente se define su firma, no se implementa código en él. Los hijos pueden hacer dos cosas:

1. Implementar el código de dicho método.
2. Declararlo también como abstracto.

Si la clase contiene algún método abstracto, se convierte en una clase abstracta. Una clase abstracta puede contener métodos no abstractos pero al menos uno de los métodos debe ser abstracto.

Para indicar en Java que un método o una clase son abstractos, se utiliza la palabra reservada **abstract**.

Veamos un ejemplo donde el hijo implementa el código del método abstracto del padre:

```
package tema7_Herencia.clasesAbstractas;

public abstract class Vehicle { //Clase abstracta

    protected int wheelCount;
    protected double speed;
    protected String colour;

    public Vehicle(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public abstract void accelerate(double amount); //Método abstracto

    public void brake(double amount) {
        speed -= amount;
    }

}
```

```
package tema7_Herencia.clasesAbstractas;

public class Car extends Vehicle {

    private double gasoline;

    public Car(int wheelCount, String colour) {
        super(wheelCount, colour);
        gasoline = 0;
    }

    public double getGasoline() {
        return gasoline;
    }
```

```
    public void refuel(double liters) {
        gasoline += liters;
    }

    @Override
    public void accelerate(double amount) { //Implementación del método abstracto
        speed += amount;
        gasoline -= amount * 0.01;
    }

}
```

```
package tema7_Herencia.clasesAbstractas;

public class Main {

    public void showAbstractClass() {

        Car car;

        car = new Car(4, "rojo");

        System.out.printf("La velocidad del coche es %.2f km/h", car.getSpeed());
        car.refuel(40.35);
        System.out.printf("\nEl coche tiene %.2f litros de gasolina",
car.getGasoline());

        car.accelerate(100);
        System.out.printf("\nLa velocidad del coche es %.2f km/h",
car.getSpeed());
        System.out.printf("\nEl coche tiene %.2f litros de gasolina",
car.getGasoline());

    }

    public static void main(String[] args) {

        new Main().showAbstractClass();

    }

}
```

Salida por consola:

```
La velocidad del coche es 0,00 km/h
El coche tiene 40,35 litros de gasolina
La velocidad del coche es 100,00 km/h
El coche tiene 39,35 litros de gasolina
```

Una clase abstracta no se puede instanciar, es decir, no se pueden crear objetos de ella haciendo uso del *new*:

```
package tema7_Herencia.clasesAbstractas;
```

```
public class Main2 {

    public void showAbstractClass() {

        Vehicle vehicle;
        vehicle = new Vehicle(2, "azul");//Error de compilación: no se puede
instanciar el tipo Vehicle
    }

    public static void main(String[] args) {

        new Main2().showAbstractClass();

    }

}
```

Haciendo uso del polimorfismo, se puede declarar una variable de una clase abstracta que referencie a un objeto hijo, de tal forma que si se invoca el método abstracto, se ejecutará la implementación realizada por el hijo de dicho método abstracto:

```
package tema7_Herencia.clasesAbstractas;

public class Main3 {

    public void showAbstractClass() {

        Vehicle vehicle;

        vehicle = new Car(4, "rojo");
        vehicle.accelerate(50.89);//Se ejecuta la implementación realizada en Car
del método abstracto accelerate
        System.out.printf("La velocidad del coche es %.2f km/h",
vehicle.getSpeed());

    }

    public static void main(String[] args) {

        new Main3().showAbstractClass();

    }

}
```

12. Interfaces

Una interfaz en Java es una colección de métodos abstractos, es decir, en una interfaz se especifica qué se debe hacer pero no cómo hacerlo. Serán las clases que implementen estas interfaces las que describen la lógica del comportamiento de los métodos.

Sirven para especificar un comportamiento que ciertas clases pueden seguir sin dictar cómo se debe implementar ese comportamiento.

A estos métodos abstractos no hace falta ponerles la palabra reservada *abstract*. Los métodos de una interfaz son públicos por defecto así que tampoco hace falta ponerles la palabra reservada *public*.

Una clase puede implementar más de una interfaz, lo que implica que debe realizar todos los métodos de cada una de ellas. Si algún método lo deja como abstracto, entonces se convierte en una clase abstracta.

En una interfaz también se pueden declarar constantes que luego puedan ser utilizadas por las clases que implementen dicha interfaz. A estas constantes de la interfaz no hace falta ponerles las palabras reservadas *static* ni *final*.

Una interfaz se define en un archivo con el mismo nombre de la interfaz y con extensión .java. Las clases que quieran implementarla, tienen que añadir la palabra reservada *implements* detrás del nombre de la clase.

Para crear una interface en el Eclipse, nos situamos en el paquete donde queramos crearla y en el botón derecho pulsamos *new* → *interface*.

Veamos un ejemplo de una interfaz que va a contener acciones que pueda realizar un vehículo, como por ejemplo, acelerar y frenar:

```
package tema7_Herencia.interfaces;

public interface ActionsVehicle {

    void accelerate(double amount);

    void brake(double amount);

}
```

La clase *Vehicle* implementa dicha interfaz por lo que el compilador le va a obligar a implementar ambos métodos:

```
package tema7_Herencia.interfaces;

public class Vehicle implements ActionsVehicle {//La clase Vehicle implementa la
interfaz ActionsVehicle

    protected int wheelCount;
    protected double speed;
    protected String colour;

    public Vehicle(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
```

```
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    @Override
    public void accelerate(double amount) {//Método a implementar debido a la
interfaz ActionsVehicle
        speed += amount;
    }

    @Override
    public void brake(double amount) {//Método a implementar debido a la interfaz
ActionsVehicle
        speed -= amount;
    }

}
```

El operador *instanceof* también se puede usar con interfaces:

```
objeto instanceof interfaz
```

Devuelve *true* si el objeto pertenece a una clase que implemente la interfaz:

```
vehicle instanceof ActionsVehicle  //true
```

12.1 Herencia entre interfaces

Las interfaces también pueden heredar de otras interfaces. En este caso, la clase que implemente la interfaz hija tendrá que realizar los métodos de la interfaz hija y los métodos de la interfaz padre.

En este caso, no ocurre como las clases que solamente pueden tener un único padre. En el caso de las interfaces, pueden tener más de un padre.

Veamos un ejemplo de interfaz que contenga métodos de un vehículo con motor de gasolina y que herede de la interfaz *ActionsVehicle*:

```java
package tema7_Herencia.interfaces;

public interface GasolineMotor extends ActionsVehicle {//GasolineMotor hereda de la interfaz ActionsVehicle

    double getGasoline();

    void refuel(double liters);

}
```

La clase *Car* va a implementar dicha interfaz por lo que tendrá que realizar los métodos de *GasolineMotor* y los métodos de *ActionsVehicle*:

```java
package tema7_Herencia.interfaces;

public class Car implements GasolineMotor {//La clase Car implementa la interfaz GasolineMotor

    private int wheelCount;
    private double speed;
    private String colour;
    private double gasoline;

    public Car(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
        gasoline = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    @Override
    public void accelerate(double amount) {//Método a implementar debido a la interfaz padre ActionsVehicle
        speed += amount;
        gasoline -= amount * 0.01;
    }
```

```
    @Override
    public void brake(double amount) {//Método a implementar debido a la interfaz
padre ActionsVehicle
        speed -= amount;
    }

    @Override
    public double getGasoline() {//Método a implementar debido a la interfaz hija
GasolineMotor
        return gasoline;
    }

    @Override
    public void refuel(double liters) {//Método a implementar debido a la
interfaz hija GasolineMotor
        gasoline += liters;
    }

}
```

12.2 Métodos por defecto

A partir de Java 8, se pueden definir métodos con una implementación por defecto dentro de las interfaces. Las clases que implementan la interfaz heredan dicho método. También pueden anularlo o incluso convertirlo en abstracto.

Para indicar que un método es por defecto se utiliza la palabra reservada *default*.

Por ejemplo, si definimos la siguiente interfaz con el siguiente método por defecto:

```
public interface Interfaz1 {
  default void metodo1(String str) {
    // Implementación por defecto del metodo1.
  }
}
```

y ahora queremos crear una clase que implemente la interfaz, dicha clase no estará obligada a implementar el método `metodo1()`. Por ejemplo, el siguiente código es totalmente válido:

```
public class Clase1 implements Interfaz1 {
    /* No estamos obligados a implementar metodo1(), aunque
       tenemos la posibilidad de anularlo.*/
}
```

Pero ¿qué ocurre si una clase implementa dos interfaces que tienen el mismo método con implementación por defecto? En ese caso la clase estará obligada a anular dicho método, porque no se puede decidir qué implementación por defecto usar. Por ejemplo:

```
public interface Interfaz1 {
  default void metodo1(String str) {
    // Implementación por defecto de metodo1.
  }
}
```

```
public interface Interfaz2 {
  default void metodo1(String str) {
    // Implementación por defecto de metodo1.
  }
}

public class Clase1 implements Interfaz1, Interfaz2 {
  // Estamos obligados a deshacer la ambigüedad
  @Override
  void metodo1(String str) {
    // Implementación específica de metodo1 en Clase1
  }
}
```

Otro aspecto interesante es que un método con implementación por defecto no puede anular a un método de la clase `java.lang.Object`, ya que es la clase base de todas las clases.

12.3 Métodos estáticos

A partir de Java 8, las interfaces también puede contener métodos estáticos con implementación por defecto. Se usan sobre todo para métodos de utilidad. Veamos un ejemplo:

```
public interface Interfaz3 {
  default void print(String str) {
    // Se llama al método estático de la interfaz desde otro
    // método default de la interfaz.
    if (!isNull(str))
      System.out.println("Cadena: " + str);
  }
  static boolean isNull(String str) {
    System.out.println("Interface Null Check");
    return str == null ? true : "".equals(str) ? true : false;
  }
}
```

Un método estático con implementación por defecto de una interfaz, como el método `isNull()` anterior, puede ser llamado desde otro método de la propia interfaz , por ejemplo desde el método `print()`.

También puede ser llamado estáticamente usando el nombre de la interfaz:

```
boolean resultado = Interfaz3.isNull("abc");
```

Se puede llamar desde cualquier clase que tenga visibilidad, aunque no implemente la interfaz. Pero no puede ser llamado a través de una instancia de una clase que implemente la interfaz, por ejemplo, el siguiente código da un error de compilación:

```
Clase miObjeto = new Clase();
miObjeto.isNull("hola");//Error de compilación
```

No se pueden definir en una interfaz métodos estáticos con implementación que tengan la misma firma que métodos de la clase `java.lang.Object`, ya que es la clase base de todas las clases.

12.4 Métodos privados

A partir de Java 9, las interfaces pueden contener métodos privados, que sólo pueden ser llamados desde métodos *default* de dicha interfaz u otros métodos privados de la misma. Sirven, básicamente, para poder separar el código de métodos con implementación por defecto. Veamos un ejemplo:

```
public interface Interfaz4 {

    // Este método privado sólo puede ser llamado por métodos default
    // de la misma interfaz.
    private int randomNumberGenerator() {
        return (new Random()).nextInt(100);
    }

    default String method(String s) {
        // Un método default puede llamar a un método privado de la interfaz.
        return s + randomNumberGenerator();
    }

}
```

A partir de Java 9 también podemos definir en una interfaz métodos estáticos privados, que sólo podrán ser llamados desde otros métodos estáticos de la interfaz. Sirven, básicamente, para poder separar el código de métodos estáticos de la interfaz. Por ejemplo:

```
public interface Interfaz5 {

    private static String prefix(String p) {
        return p.equals("male")? "Mr. " : "Ms. ";
    }
    static String name(String n, String p) {
        return prefix(p) + n;
    }

}
```

12.5 Diferencias entre una interfaz y una clase abstracta

Pero entonces, si las interfaces pueden tener métodos con implementación por defecto y métodos privados ¿qué diferencia hay entre una interfaz con métodos por defecto y una clase abstracta? La diferencia principal es que una interfaz no tiene estado, es decir, no podemos almacenar atributos en ella, mientras que una clase abstracta sí.

12.6 Utilización de una interfaz como un tipo de datos

Al declarar una interfaz, se declara un nuevo tipo de datos, lo que significa que se puede declarar una variable cuyo tipo es una interfaz. Pero, ¿qué va a contener dicha variable? Puede contener un objeto de cualquier clase que implemente dicha interfaz. Con dicha variable, las acciones que se pueden realizar son los métodos de la interfaz.

```
package tema7_Herencia.interfaces;
```

```
public class Main1 {

    public void showInterfaces() {

        Vehicle vehicle;
        ActionsVehicle actionsVehicle;//El tipo de la variable es la interfaz
ActionsVehicle

        vehicle = new Vehicle(2, "azul");
        System.out.printf("La velocidad del vehículo es %.2f km/h",
vehicle.getSpeed());

        actionsVehicle = vehicle;//Se le asigna un objeto de la clase Vehicle,
que implementa la interfaz ActionsVehicle
        actionsVehicle.accelerate(100.39);//Ejecuta un método de la interfaz
        System.out.printf("\nLa velocidad del vehículo es %.2f km/h",
vehicle.getSpeed());

    }

    public static void main(String[] args) {

        new Main1().showInterfaces();

    }

}
```

Salida por consola:

```
La velocidad del vehículo es 0,00 km/h
La velocidad del vehículo es 100,39 km/h
```

Con una variable de tipo interfaz, las acciones que se pueden realizar son los métodos de la interfaz, es decir, con una variable de tipo *ActionsVehicle* lo que se puede realizar son los métodos *accelerate* y *brake* que son los métodos que pertenecen a la interfaz. Si intentáramos realizar un *getSpeed* que pertenece a la clase *Vehicle* y no se encuentra entre los métodos de la interfaz, nos daría un error de compilación:

```
package tema7_Herencia.interfaces;

public class Main2 {

    public void showInterfaces() {

        Vehicle vehicle;
        ActionsVehicle actionsVehicle;

        vehicle = new Vehicle(2, "azul");
        System.out.printf("La velocidad del vehículo es %.2f km/h",
vehicle.getSpeed());

        actionsVehicle = vehicle;
```

```
        actionsVehicle.accelerate(100.39);
        System.out.printf("\nLa velocidad del vehículo es %.2f km/h",
actionsVehicle.getSpeed());//Error de compilación: el método getSpeed no está
definido para el tipo ActionsVehicle

    }

    public static void main(String[] args) {

        new Main2().showInterfaces();

    }

}
```

El error de compilación se puede solucionar con un casting:

```
package tema7_Herencia.interfaces;

public class Main3 {

    public void showInterfaces() {

        Vehicle vehicle;
        ActionsVehicle actionsVehicle;

        vehicle = new Vehicle(2, "azul");
        System.out.printf("La velocidad del vehículo es %.2f km/h",
vehicle.getSpeed());

        actionsVehicle = vehicle;
        actionsVehicle.accelerate(100.39);
        System.out.printf("\nLa velocidad del vehículo es %.2f km/h", ((Vehicle)
actionsVehicle).getSpeed());//Casting para solucionar el error

    }

    public static void main(String[] args) {

        new Main3().showInterfaces();

    }

}
```

Salida por consola:

```
La velocidad del vehículo es 0,00 km/h
La velocidad del vehículo es 100,39 km/h
```

También podemos usar el casting para asignar una variable de tipo interfaz a una variable de tipo clase, siempre y cuando la variable de tipo interfaz contenga un objeto de dicha clase:

```
Vehicle vehicle = (Vehicle) actionsVehicle;
```

El casting también se puede utilizar para asignar un array de tipo clase a un array de tipo interfaz, como por ejemplo `actionsVehicle = (ActionsVehicle[]) vehicle;`:

```
package tema7_Herencia.interfaces;

public class Main4 {

    public void showInterfaces() {

        Vehicle vehicle[] = new Vehicle[3];
        ActionsVehicle actionsVehicle[];

        vehicle[0] = new Vehicle(2, "azul");
        vehicle[1] = new Vehicle(4, "rojo");
        vehicle[2] = new Vehicle(4, "blanco");
        for(int i=0;i<vehicle.length;i++) {
            System.out.printf("La velocidad del vehículo es %.2f km/h\n",
vehicle[i].getSpeed());
        }
        //Casting entre arrays clase-interfaz:
        actionsVehicle = (ActionsVehicle[]) vehicle;
        for(int i=0;i<vehicle.length;i++) {
            actionsVehicle[i].accelerate(i*40);
        }
        for(int i=0;i<vehicle.length;i++) {
            System.out.printf("La velocidad del vehículo es %.2f km/h\n",
vehicle[i].getSpeed());
        }

    }

    public static void main(String[] args) {

        new Main4().showInterfaces();

    }

}
```

Sin embargo al revés no se puede, es decir, no se puede asignar un array de tipo interfaz a un array de tipo clase, porque da un error `ClassCastException`:

```
vehicle = (Vehicle[]) actionsVehicle;//Error de ejecución ClassCastException
```

En Java, aunque Vehicle implemente ActionsVehicle, los arrays no son covariantes de esta manera. Si necesitáramos realizar algo así, tendríamos que hacer un casting en un bucle para cada elemento individual.

13. Clases anidadas

En Java se permite escribir una clase dentro de otra clase. La clase de dentro se llama clase **anidada** y la que la contiene, clase **contenedora o externa**.

Estas clases se utilizan con los siguientes propósitos:

- Agrupación de clases relacionadas.
- Control de visibilidad de las clases.
- Proximidad entre la definición y el uso de las clases.
- Definición de clases simples de ayuda o adaptación.
- Código más claro que evita el exceso de clases muy pequeñas que no necesitan conocer los usuarios de un paquete.

Las clases anidadas se dividen en dos categorías: las clases anidadas estáticas y las clases anidadas no estáticas o clases internas.

13.1 Clases anidadas estáticas

Son clases declaradas de tipo *static* que se comportan como una clase normal de Java pero que se encuentran dentro de otra clase. También se pueden usar dentro de una interfaz.

Desde la clase anidada estática solamente se pueden acceder a los atributos estáticos de la clase contenedora.

Se pueden crear objetos sin crear ningún objeto de la clase contenedora.

Para hacer referencia a una clase anidada estática hay que indicar también la clase contenedora: `ClaseContenedora.ClaseAnidada`.

Se usan principalmente con excepciones y enumerados.

```
package tema7_Herencia.clasesAnidadas;

public class ContainerClass {

    public static class StaticNestedClass {

        public void staticNestedMethod() {
            System.out.println("Clase anidada estática");
        }

    }

    public void containerMethod() {
        System.out.println("Clase externa o contenedora");
    }

}
```

```
package tema7_Herencia.clasesAnidadasEstaticas;

public class Main {
```

```
    public void showStaticNestedClass() {

        ContainerClass.StaticNestedClass nested = new
ContainerClass.StaticNestedClass();
        nested.staticNestedMethod();

        ContainerClass container = new ContainerClass();
        container.containerMethod();

    }

    public static void main(String[] args) {

        new Main().showStaticNestedClass();

    }

}
```

Salida por consola:

```
Clase anidada estática
Clase externa o contenedora
```

13.2 Clases anidadas no estáticas o clases internas

Las clases anidadas no estáticas o clases internas tienen acceso a todos los atributos y métodos de la clase contenedora, por lo tanto, para que exista un objeto de una clase interna es necesario que exista un objeto de la clase contenedora.

```
package tema7_Herencia.clasesInternas;

public class ContainerClass {

    private int numContainer = 10;

    public class InnerClass {

        public void innerMethod() {
            System.out.printf("Clase interna. Puede acceder a numContainer: %d",
numContainer);
        }

    }

    public void containerMethod() {
        InnerClass inner = new InnerClass();
        inner.innerMethod();
    }

}
```

```
package tema7_Herencia.clasesInternas;
```

```
public class Main {

    public void showInnerClass() {

        ContainerClass container = new ContainerClass();
        container.containerMethod();

    }

    public static void main(String[] args) {

        new Main().showInnerClass();

    }

}
```

Salida por consola:

```
Clase interna. Puede acceder a numContainer: 10
```

También se puede crear un objeto de la clase interna desde fuera de la clase externa siempre y cuando la clase interna sea visible:

```
package tema7_Herencia.clasesInternas;

public class Main2 {

    public void showInnerClass() {

        ContainerClass container = new ContainerClass();
        ContainerClass.InnerClass inner = container.new InnerClass();
        inner.innerMethod();

    }

    public static void main(String[] args) {

        new Main2().showInnerClass();

    }

}
```

Salida por consola:

```
Clase interna. Puede acceder a numContainer: 10
```

Se puede hacer una clasificación de las clases internas en función de dónde y cómo se utilicen:

- Clases miembro
- Clases dentro de un método.

- Clases dentro de un bloque.
- Clases anónimas.

13.2.1 Clases internas miembro

Se utilizan como atributos de la clase contenedora. Si se declaran como privadas, la clase contenedora es la única que conoce su existencia.

```
package tema7_Herencia.clasesInternasMiembro;

public class ContainerClass {

    private int num = 10;
    private InnerClass inner = new InnerClass();

    private class InnerClass {

        private int num = 20;

        public void innerMethod() {
            System.out.printf("Número de la contenedora: %d. Número de la
interna: %d", ContainerClass.this.num, num);
        }

    }

    public void containerMethod() {
        inner.innerMethod();
    }

}
```

```
package tema7_Herencia.clasesInternasMiembro;

public class Main {

    public void showInnerMemberClass() {

        ContainerClass container = new ContainerClass();
        container.containerMethod();

    }

    public static void main(String[] args) {

        new Main().showInnerMemberClass();

    }

}
```

Salida por consola:

```
Número de la contenedora: 10. Número de la interna: 20
```

Al usar *this* dentro de una clase interna, éste se refiere al objeto de la clase interna. Para poder referirse al objeto de la clase contenedora, hay que anteponerle al *this* el nombre de dicha clase, tal y como se puede observar en el ejemplo: `ContainerClass.this.num`.

13.2.2 Clases internas dentro de un método

Se definen dentro de un método de la clase contenedora por lo que solamente se pueden utilizar dentro de dicho método.

Se utilizan cuando el método intenta solucionar un problema y necesita apoyarse en una clase pero no se necesita que esta clase esté disponible fuera, por lo tanto, son clases que quedan fuera del diseño.

La clase interna tiene acceso a los métodos y atributos de la clase contenedora y a las variables locales y parámetros del método donde se la declara.

```
package tema7_Herencia.clasesInternasMetodo;

public class ContainerClass {

    private int attribute = 10;

    public void containerMethod(int parameter) {

        int localVariable = 20;

        class InnerClass {

            public void innerMethod() {
                System.out.printf("Clase interna a método--->\nAtributo de la
contenedora: %d", attribute);
                System.out.printf("\nVariable local: %d", localVariable);
                System.out.printf("\nParámetro: %d", parameter);
            }

        }

        InnerClass inner = new InnerClass();
        inner.innerMethod();
    }

}
```

```
package tema7_Herencia.clasesInternasMetodo;

public class Main {

    public void showInnerMethodClass() {

        ContainerClass container = new ContainerClass();
        container.containerMethod(30);
```

```
    }

    public static void main(String[] args) {

        new Main().showInnerMethodClass();

    }

}
```

Salida por consola:

```
Clase interna a método--->
Atributo de la contenedora: 10
Variable local: 20
Parámetro: 30
```

13.2.3 Clases internas dentro de un bloque

Sólo son visibles y utilizables dentro del bloque de código en el que se encuentran definidas.

```
package tema7_Herencia.clasesInternasBloque;

public class ContainerClass {

    private int attribute = 10;

    public void containerMethod(int parameter) {

        int localVariable = 20;

        if (parameter > localVariable) {
            class InnerClass {

                public void innerMethod() {
                    System.out.printf("Clase interna a bloque--->\nAtributo de la
contenedora: %d", attribute);
                    System.out.printf("\nVariable local: %d", localVariable);
                    System.out.printf("\nParámetro: %d", parameter);
                }

            }
            InnerClass inner = new InnerClass();
            inner.innerMethod();
        }

    }

}
```

```
package tema7_Herencia.clasesInternasBloque;

public class Main {
```

```
    public void showInnerBlockClass() {

        ContainerClass container = new ContainerClass();
        container.containerMethod(30);

    }

    public static void main(String[] args) {

        new Main().showInnerBlockClass();

    }

}
```

Salida por consola:

```
Clase interna a bloque--->
Atributo de la contenedora: 10
Variable local: 20
Parámetro: 30
```

Si se intenta utilizar la clase fuera del bloque, da un error de compilación informando que la clase no puede ser resuelta como un tipo:

```
package tema7_Herencia.clasesInternasBloque;

public class ContainerClass2 {

    private int attribute = 10;

    public void containerMethod(int parameter) {

        int localVariable = 20;

        if (parameter > localVariable) {
            class InnerClass {

                public void innerMethod() {
                    System.out.printf("Clase interna a bloque--->\nAtributo de la
contenedora: %d", attribute);
                    System.out.printf("\nVariable local: %d", localVariable);
                    System.out.printf("\nParámetro: %d", parameter);
                }

            }

        }

        InnerClass inner = new InnerClass();//Error de compilación: InnerClass no
puede ser resuelto como un tipo
        inner.innerMethod();

    }
```

```
}
```

13.2.4 Clases inline anónimas

Son clases sin nombre que se definen e instancian en una sola operación. Este tipo de clases se utiliza cuando se quiere anular el método de una clase o implementar una interfaz solamente para un momento puntual evitando crear una clase nueva para un solo uso. El término **inline** se debe a que en el cuerpo de un método se puede escribir una clase ahí mismo, en la línea, es decir, sin necesidad de hacerlo en otro archivo.

Por ejemplo, tenemos la siguiente clase *MyClass*:

```
package tema7_Herencia.clasesInlineAnonimas;

public class MyClass {

    protected String message = "Clases inline anónimas";

    public void showMessage() {
        System.out.println(message);
    }

}
```

Supongamos que queremos anular el método `showMessage()` para mostrar el mensaje en rojo en un momento puntual y creamos una subclase o clase hija de *MyClass* que sobrescribiera el método:

```
package tema7_Herencia.clasesInlineAnonimas;

import static tema1_11_EscrituraEnPantalla.colores.Colors.*;

public class Subclass extends MyClass {

    @Override
    public void showMessage() {
        System.out.println(RED + message + RESET);
    }

}
```

Entonces estaríamos creando una clase para algo que vamos a hacer solamente una vez. Y si quisiéramos en otro momento hacer lo mismo pero en azul, tendríamos que crear otra subclase para hacerlo. Entonces, la solución es crear una clase inline anónima. Se llama anónima porque en ningún momento aparece el nombre `Subclass`, es decir, estamos haciendo lo mismo que antes pero sin crear la subclase y lo estamos haciendo sobre la marcha:

```
package tema7_Herencia.clasesInlineAnonimas;

import static tema1_4_EscrituraEnPantalla.colores.Colors.RED;
import static tema1_4_EscrituraEnPantalla.colores.Colors.RESET;
```

```
public class Main {

    public void showAnonymousInnerClass() {

        MyClass anonymousInner = new MyClass() {
            @Override
            public void showMessage() {
                System.out.println(RED + message + RESET);
            }
        };
        anonymousInner.showMessage();

        new MyClass() {//Lo mismo pero sin crear la variable anonymousInner
            @Override
            public void showMessage() {
                System.out.println(RED + message + RESET);
            }
        }.showMessage();

    }

    public static void main(String[] args) {

        new Main().showAnonymousInnerClass();

    }

}
```

Salida por consola:

Clases inline anónimas

Clases inline anónimas

Veamos otro ejemplo pero con interfaces:

```
package tema7_Herencia.clasesInlineAnonimas;

public interface Message {

    void showMessage();

}
```

Creamos una clase que implemente la interfaz y que escriba el mensaje en rojo:

```
package tema7_Herencia.clasesInlineAnonimas;

import static tema1_11_EscrituraEnPantalla.colores.Colors.*;

public class MyClass2 implements Message {

    @Override
    public void showMessage() {
        System.out.println(RED + "Clases inline anónimas" + RESET);
    }

}
```

Pero igual que antes, si solamente vamos a hacer ésto una vez, estamos creando una clase para un solo uso. Incluso si quisiéramos hacer también lo mismo en azul, tendríamos que crear otra clase para ello.

Entonces, lo podemos hacer mediante una clase inline anónima:

```
package tema7_Herencia.clasesInlineAnonimas;

import static tema1_4_EscrituraEnPantalla.colores.Colors.BLUE;
import static tema1_4_EscrituraEnPantalla.colores.Colors.RED;
import static tema1_4_EscrituraEnPantalla.colores.Colors.RESET;

public class Main2 {

    public void showAnonymousInnerClass() {

        new Message() {
            @Override
            public void showMessage() {
                System.out.println(RED + "Clases inline anónimas" + RESET);
            }
        }.showMessage();

        new Message() {
            @Override
            public void showMessage() {
                System.out.println(BLUE + "Clases inline anónimas" + RESET);
            }
        }.showMessage();

    }

    public static void main(String[] args) {

        new Main2().showAnonymousInnerClass();

    }

}
```

Salida por consola:

Clases inline anónimas

Clases inline anónimas

14. Clases selladas

Las clases selladas (en inglés, *sealed classes*) permiten restringir qué clases pueden heredar de una clase. Son útiles para definir jerarquías de clases más controladas y seguras, mejorando el diseño de tipos y ayudando a evitar errores.

Esto se hace utilizando la palabra clave `sealed` en la declaración de la clase y `permits` para listar las clases que tienen permitido extenderla. Las clases que extienden una clase sellada deben ser marcadas como una de estas opciones:

- `final`: la clase no puede ser extendida más.
- `sealed`: la clase sigue siendo sellada y permite que otras clases la extiendan, pero también deben ser especificadas con `permits`.
- `non-sealed`: la clase puede ser extendida libremente sin restricciones.

Todas las subclases permitidas deben ser conocidas en tiempo de compilación y deben estar ubicadas en el mismo módulo o paquete.

Veamos un ejemplo de una clase sellada `Vehicle` que solamente permite que hereden de ella las clases `Car` y `Bike`:

```
package tema7_Herencia.clasesSelladas;

public sealed class Vehicle permits Car, Bike {

    protected int wheelCount;
    protected double speed;
    protected String colour;

    public Vehicle(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }
```

```
    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

    @Override
    public String toString() {
        return String.format("Número de ruedas: %d, color: %s, velocidad: %.2f",
wheelCount, colour, speed);
    }

}
```

```
package tema7_Herencia.clasesSelladas;

public final class Car extends Vehicle {

    private double gasoline;

    public Car(int wheelCount, String colour) {
        super(wheelCount, colour);
        gasoline = 0;
    }

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

    @Override
    public void accelerate(double amount) {
        super.accelerate(amount);
        gasoline -= amount * 0.01;
    }

    @Override
    public String toString() {
        return String.format("Coche -> %s, gasolina: %.2f", super.toString(),
gasoline);
    }

}
```

```
package tema7_Herencia.clasesSelladas;

public final class Bike extends Vehicle {
```

```
    public Bike(int wheelCount, String colour) {
        super(wheelCount, colour);
    }

    @Override
    public String toString() {
        return String.format("Bicicleta -> %s", super.toString());
    }

}
```

Si se intenta crear una motocicleta que herede de `Vehicle`, nos daría error de compilación:

```
package tema7_Herencia.clasesSelladas;

public final class Motorbike extends Vehicle {//Error de compilación indicando
que no es una subclase permitida

    private double gasoline;

    public Motorbike(int wheelCount, String colour) {
        super(wheelCount, colour);
        gasoline = 0;
    }

    public double getGasoline() {
        return gasoline;
    }

    public void refuel(double liters) {
        gasoline += liters;
    }

    @Override
    public void accelerate(double amount) {
        super.accelerate(amount);
        gasoline -= amount * 0.05;
    }

    @Override
    public String toString() {
        return String.format("Moto -> %s, gasolina: %.2f", super.toString(),
gasoline);
    }

}
```

```
package tema7_Herencia.clasesSelladas;

public class Main {

    public void showSealedClasses() {

        Vehicle v;
```

```
        v = new Car(4, "rojo");
        System.out.println(v);
        v = new Bike(2, "verde");
        System.out.println(v);
        v = new Motorbike(2, "azul");//Error de ejecución
        System.out.println(v);

    }

    public static void main(String[] args) {

        new Main().showSealedClasses();

    }

}
```

Salida por consola que aborta con un error de ejecución:

```
Coche -> Número de ruedas: 4, color: rojo, velocidad: 0,00, gasolina: 0,00
Bicicleta -> Número de ruedas: 2, color: verde, velocidad: 0,00
Exception in thread "main" java.lang.IncompatibleClassChangeError: class
tema7_Herencia.clasesSelladas.Motorbike cannot inherit from sealed class
tema7_Herencia.clasesSelladas.Vehicle
    at java.base/java.lang.ClassLoader.defineClass1(Native Method)
    at java.base/java.lang.ClassLoader.defineClass(ClassLoader.java:1027)
    at
java.base/java.security.SecureClassLoader.defineClass(SecureClassLoader.java:150)
    at
java.base/jdk.internal.loader.BuiltinClassLoader.defineClass(BuiltinClassLoader.j
ava:862)
    at
java.base/jdk.internal.loader.BuiltinClassLoader.findClassOnClassPathOrNull(Built
inClassLoader.java:760)
    at
java.base/jdk.internal.loader.BuiltinClassLoader.loadClassOrNull(BuiltinClassLoad
er.java:681)
    at
java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.jav
a:639)
    at
java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoaders.
java:188)
    at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:526)
    at tema7_Herencia.clasesSelladas.Main.showSealedClasses(Main.java:13)
    at tema7_Herencia.clasesSelladas.Main.main(Main.java:20)
```

15. Interfaces selladas

Las interfaces selladas (en inglés, *sealed interfaces*) permiten restringir qué clases pueden implementarla y qué interfaces pueden extenderla. Son útiles para definir jerarquías de comportamientos más controladas y seguras, mejorando el diseño de tipos y ayudando a evitar errores.

Para declarar una interfaz sellada, se utiliza la palabra clave `sealed` en la declaración de la interfaz y `permits` para listar las clases y/o interfaces que están permitidas implementarla o extenderla. Ejemplo:

```
public sealed interface ActionsVehicle permits Vehicle,GasolineMotor {

    void accelerate(double amount);

    void brake(double amount);

}
```

Las clases que implementan una interfaz sellada deben ser marcadas como final, sealed o non-sealed. Las interfaces que heredan de una interfaz sellada deben ser marcadas como sealed o non-sealed.