Programación orientada a objetos

Tabla de contenido

Definiciones

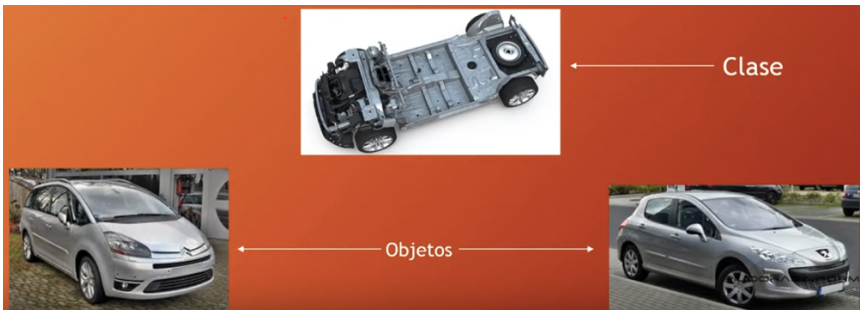
Clases y objetos
La programación orientada a objetos es un paradigma de programación que se basa, como su nombre indica, en la **utilización de objetos**. Estos objetos también se suelen llamar instancias.
Un **objeto** en términos de POO no se diferencia mucho de lo que conocemos como un objeto en la vida real. Pensemos por ejemplo en un coche. Nuestro coche sería un coche más de tantos que hay. Todos esos coches son objetos concretos que podemos ver y tocar.
Tanto mi coche como el coche del vecino tienen algo en común, ambos son coches. En este caso mi coche y el coche del vecino serían instancias (objetos) y Coche (a secas) sería una clase. La palabra Coche define algo genérico, es una abstracción, no es un coche concreto sino que hace referencia a unos elementos que tienen una serie de propiedades como matrícula, marca, modelo, color, etc.; este conjunto de propiedades se denominan atributos o variables de instancia.

Clase
Concepto abstracto que denota una serie de cualidades, por ejemplo Coche.

Instancia
Objeto palpable, que se deriva de la concreción de una clase, por ejemplo mi coche.

Atributos
Conjunto de características que comparten los objetos de una clase, por ejemplo para la clase coche tendríamos matrícula, marca, modelo, color y número de plazas.

En Java, los nombres de **las clases se escriben con la primera letra en mayúscula** mientras que **los nombres de las instancias comienzan con una letra en minúscula**. Por ejemplo, la clase coche se escribe en Java como Coche y el objeto "mi coche" se podría escribir como miCoche.
Definiremos cada clase en un fichero con el mismo nombre más la extensión .java, por tanto, la definición de la clase Coche debe estar contenida en un fichero con nombre Coche.java



Vamos a definir a continuación la clase Libro con los atributos isbn, autor, titulo y numeroPaginas.

```
/*Definición de la clase Libro*/
public class Libro {
// atributos
String isbn;
String titulo;
String autor;
int numeroDePaginas;
 }
```

A continuación creamos varios objetos de esta clase.

```
/*Programa que prueba la clase Libro*/
public class PruebaLibro {
        public static void main(String[] args) {
        Libro libro1 = new Libro();
        Libro libro2 = new Libro();
        Libro libro3 = new Libro();
        }
}
```

Hemos creado tres instancias de la clase libro: libro1, libro2 y libro3.
Las variables de instancia (atributos) determinan las cualidades de los objetos.

Encapsulamiento y ocultación

Uno de los pilares en los que se basa la Programación Orientada a Objetos es el encapsulamiento. Básicamente, el encapsulamiento consiste en definir todas las propiedades y el comportamiento de una clase dentro de esa clase; es decir, en la clase Coche estará definido todo lo concerniente a la clase Coche y en la clase Libro estará definido todo lo que tenga que ver con la clase Libro.
El encapsulamiento hay que tenerlo siempre muy presente al programar utilizando clases y objetos. Conviene no mezclar parte de una clase dentro de otra clase distinta para resolver un problema puntual. Se deben escribir los programas de forma que cada cosa esté en su sitio.

La ocultación es una técnica que incorporan algunos lenguajes (entre ellos Java) que permite esconder los elementos que definen una clase, de tal forma que desde otra clase distinta no se pueden "ver las tripas" de la primera. La ocultación facilita, como veremos más adelante, el encapsulamiento.

Métodos

Un coche arranca, para, se aparca, hace sonar el claxon, se puede llevar a reparar... Un gato puede comer, dormir, maullar, ronronear...
Las acciones asociadas a una clase se llaman métodos. Estos métodos se definen dentro del cuerpo de la clase y se suelen colocar a continuación de los atributos.
Los métodos determinan el comportamiento de los objetos.

Definición de la clase Gato
Para saber qué atributos debe tener esta clase hay que preguntarse qué características tienen los gatos. Todos los gatos son de un color determinado, pertenecen a una raza, tienen una edad, un nombre y un peso (en kg). Éstos serán los atributos de la clase Gato.
Para saber qué métodos debemos implementar hay que preguntarse qué acciones están asociadas a los gatos. Bien, pues los gatos maullan, ronronean, comen y se pelean entre ellos. Esos serán los métodos que definamos en la clase.

```
/* Definición de la clase Gato */
public class Gato {
String color, raza, nombre;
int edad;
double peso;

 // constructores
Gato (String n) {
      this.nombre = n;
}
Gato () {
      this.raza = "callejero";
}

// getter
String getNmbre() {
```

```
	return this.nombre;
}

String getRaza() {
	return this.raza;
}

// setter
void setRaza(String r) {
	this.raza = r;
}

// métodos
//el gato maulla
void maulla() {
	System.out.println("Miauuuu");
}
//el gato ronronea
void ronronea() {
	System.out.println("mrrrrrr");
}
//el gato come
void come(String comida) {
	if (comida.equals("pescado")) {
		System.out.println("Hmmmm, gracias");
	}
	else {
	    System.out.println("Lo siento, yo solo como pescado");
	    }
}
  // el gato se pelea
void peleaCon(Gato contrincante) {
	if (this.raza.equals("peleona")) {
		System.out.println("cuidado que soy peleón");
	} else {
		if (contrincante.getRaza().equals("peleona")) {
			System.out.println("no peleo con abusones");
		} else {
			System.out.println("vamos a dejarlo en tablas");
		}
}

}
```

Constructor

Se encarga de dar un estado inicial a nuestro objeto
El método constructor tiene siempre el mismo nombre que la clase y se utiliza normalmente para inicializar los atributos.
Los atributos de la clase Gato - color, raza, nombre, edad y peso - se declaran igual que las variables que hemos venido usando hasta ahora, pero hay una gran diferencia entre estos atributos y las variables que aparecen en el main (programa principal). Una variable definida en el cuerpo del programa principal es única, sin embargo cada uno de los objetos que se crean en el programa principal tienen sus propios atributos; es decir, si en el programa principal se crean 10 objetos de la clase Gato, cada uno tiene sus valores para los atributos color, raza, etc.

Fíjate en el método void peleaCon(Gato contrincante). Se puede pasar un objeto como parámetro.

Programa para probar la clase Gato.

```
/* Programa que prueba la clase Gato*/
public class Prueba_Gato {
      public static void main(String[] args) {

    Gato gato1 = new Gato("Gardfield");
    System.out.println("Garfield es:"+ gato1.getNombre());
    gato1.maulla();
    System.out.print("Toma tarta..");
    gato1.come("tarta");
    System.out.print("Toma pescado..");
    gato1.come("pescado");
    gato1.setRaza("siames");

    Gato tom = new Gato("tomi");
    System.out.println("El gato tom se llama:"+tom.getNombre());
    System.out.print("Toma sopa..");
    tom.come("sopa de verduras");
    tom.setRaza("peleona");

    Gato lisa = new Gato("Lisa");
    System.out.println("Lisa es:"+lisa.getNombre());
    lisa.maulla();
    lisa.setRaza("persa");

    gato1.peleaCon(lisa);
    lisa.peleaCon(tom);
    tom.peleaCon(gato1);

}
```

Observa cómo al crear una instancia se llama al constructor (tiene el mismo nombre de la clase) y sirve para inicializar los atributos. En este caso se inicializa el atributo nombre.
Se pueden definir distintos constructores dentro de la misma clase (en este ejemplo de Gato en concreto hemos definidos 2 constructores pero luego hemos utilizado solo 1).
Gato garfield = new Gato("garfield");

Hay métodos que no toman ningún parámetro, por ejemplo
garfield.maulla();

Y hay otros métodos que deben tomar parámetros obligatoriamente.
garfield.come("tarta");

Métodos getter y setter

Vamos a crear la clase Cubo. Para saber qué atributos se deben definir, nos preguntamos qué características tienen los cubos - igual que hicimos con la clase Gato. Todos los cubos tienen una determinada capacidad, un color, están hechos de un determinado material - plástico, latón, etc. - y puede que tengan asa o puede que no. Un cubo se fabrica con el propósito de contener líquido; por tanto otra característica es la cantidad de litros de líquido que contiene en un momento determinado. Por ahora, solo nos interesa saber la capacidad máxima y los litros que contiene el cubo en cada momento, así que esos serán los atributos que tendremos en cuenta.

```
/*Definición de la clase Cubo*/
public class Cubo {
	int capacidad; // capacidad máxima en litros
	int contenido; // contenido actual en litros

	// constructor
	Cubo (int c) {
		this.capacidad = c;
	}

	// métodos getter
	int getCapacidad() {
		return this.capacidad;
	}
	int getContenido() {
		return this.contenido;
	}

	// método setter
	void setContenido(int litros) {
		this.contenido = litros;
	}

	//Vacía el contenido del cubo
	void vacia() {
		this.contenido = 0;
	}

	//Llena el cubo al máximo de su capacidad
	void llena() {
		this.contenido = this.capacidad;
	}

	//Pinta los bordes del cubo con el carácter # y el agua con ~
	void pinta() {
		for (int nivel = this.capacidad; nivel > 0; nivel--) {
			if (this.contenido >= nivel) {
				System.out.println("|~~|");
```

```
                } else {
                        System.out.println("|  |");
                }
        }
        System.out.println("----");
    }
    // Vuelca un cubo en otro, antes comprueba cuánto le cabe al
    otro cubo
    void vuelcaEn(Cubo destino) {
    int libres = destino.getCapacidad() - destino.getContenido();
     if (libres > 0) {
        if (this.contenido <= libres) {
        destino.setContenido(destino.getContenido() +
    this.contenido);
        this.vacia();
        } else {
                this.contenido -= libres;
                destino.llena();
        }
     }
    }
```

Los métodos getCapacidad y getContenido, son métodos getter. Su cometido es devolver el valor de un atributo. Podrían tener cualquier nombre pero en Java es costumbre llamarlos con la palabra get (obtener) seguida del nombre del atributo

```
int getCapacidad() {
    return this.capacidad;
}
int getContenido() {
    return this.contenido;
}
```

Fijémonos ahora en este otro método, es un método setter

```
void setContenido(int litros) {
    this.contenido = litros;
}
```

Este tipo de métodos tiene el cometido de establecer un valor para un determinado atributo. Se denomina como por set (asignar) más el nombre del atributo.

¿por qué se crea un método getter para extraer el valor de una variable y no se accede a la variable directamente?

Parece más lógico hacer System.out.print(miCubo.capacidad) que hacer esto otro System.out.print(miCubo.getCapacidad()). Sin embargo, en Java se opta casi siempre por esta última opción. Lo veremos más adelante en el apartado Ámbito/visibilidad de los elementos de una clase - public , protected y private

Probamos, con el siguiente ejemplo, la clase Cubo que acabamos de definir.

```
/*Programa que prueba la clase Cubo*/
public class Prueba_Cubo {
public static void main(String[] args) {
Cubo cubo2 = new Cubo(2);
Cubo cubo4 = new Cubo(4);

    System.out.println("Cubo2:");
    cubo2.pinta();
```

```
        System.out.println("Cubo4:");
        cubo4.pinta();

        System.out.println("Lleno el cubo2:");
        cubo2.llena();
        cubo2.pinta();

        System.out.println("El cubo4 sigue vacío:");
        cubo4.pinta();

        System.out.println("Ahora vuelco cubo2 en cubo4");
        cubo2.vuelcaEn(cubo4);
        System.out.println("Cubo2:");
        cubo2.pinta();
        System.out.println("Cubo4:");
        cubo4.pinta();

        System.out.println("Lleno el cubo4:");
        cubo4.llena();
        cubo4.pinta();

        System.out.println("Ahora vuelco cubo4 en el cubo2");
        cubo4.vuelcaEn(cubo2);
        System.out.println("Cubo2:");
        cubo2.pinta();
        System.out.println("Cubo4:");
        cubo4.pinta();
    }
}
```

Método toString()

El método toString() en Java, como su propio nombre indica, se utiliza para convertir a String (es decir, a una cadena de texto) cualquier objeto Java

El método **toString()** es un método de la clase **Object**. Como todos los objetos en **Java** heredan de dicha clase, todos ellos tienen acceso a este método

Al construir clases, todas heredan de la clase Object, tanto las nuestras como las que están predefinidas en la Api de Java. Para cada clase es como si la instrucción *public class Clase_ejemplo extends Object* se diese por supuesta. Por eso cada vez que ponemos nuestro objeto y el . nos salen nuestros métodos y los de la clase Object ya que estamos heredando de Object.

En Java existe una solución para mostrar información sobre un objeto por pantalla. Si se quiere mostrar el contenido de la variable entera x se utiliza System.out.print(x) y si se quiere mostrar el valor de la variable de tipo cadena de caracteres nombre se escribe System.out.print(nombre). De la misma manera, si se quiere mostrar el objeto miCancion que pertenece a la clase Cancion, también se podría usar System.out.print(miCancion). Java sabe perfectamente cómo mostrar números y cadenas de caracteres pero no sabe a priori cómo se pintan Canciones. Para indicar a Java cómo debe pintar un objeto de la clase Cancion basta con implementar el método toString dentro de la clase, por ejemplo de esta manera:

```
public String toString() {
String cadena = "\n----------------------------------------";
cadena += "\nCódigo: " + this.codigo;
cadena += "\nAutor: " + this.autor;
cadena += "\nTítulo: " + this.titulo;
cadena += "\nGénero: " + this.genero;
cadena += "\nDuración: " + this.duracion;
cadena += "\n----------------------------------------";
return cadena;
 }
```

Ámbito/visibilidad de los elementos de una clase

public, protected y private

Al definir los elementos de una clase, se pueden especificar sus ámbitos (scope) de visibilidad o accesibilidad con las palabras reservadas public (público), protected (protegido) y private (privado).
En la siguiente tabla se muestra desde dónde es visible/accesible un elemento (atributo o método) según el modificador que lleve.

	En la misma clase	En el mismo paquete	En una subclase	Fuera del paquete
private	✔	✖	✖	✖
protected	✔	✔	✔	✖
public	✔	✔	✔	✔
sin especificar	✔	✔	✖	✖

Figura Ámbito de los elementos de una clase.

Por ejemplo, un atributo private solamente es accesible desde la misma clase, sin embargo, a un método protected se podrá acceder desde la misma clase donde esté definido, desde otro fichero dentro del mismo paquete o desde una subclase.
Como regla general, se suelen declarar private los atributos o variables de instancia y public los métodos.

```
/*Definición de la clase Animal */
public class Animal {

private String tipo;
private String nombre;

public Animal (String tipo, String nombre) {
      this.tipo = tipo;
      this.nombre = nombre;
}
public Animal () {
      this.tipo = "salvaje";
      this.nombre = "sin nombre";
}

public String getNombre(){ return nombre; }
public void setNombre(String nombre){ this.nombre = nombre; }
public String getTipo(){ return this.tipo; }
public void setTipo(String tipo) { this.tipo = tipo; }
public String toString(){
 return "Tipo: "+this.tipo+"- Nombre:"+this.nombre + "\n";
}

/*el animal se echa a dormir*/
public void duerme() {
      System.out.println("Zzzzzzz");
}
```

Los atributos tipo y nombre se han definido como private. Eso quiere decir que a esos atributos únicamente se tiene acceso dentro de la clase Animal. Sin embargo, todos los métodos se han definido public, lo que significa que se podrán utilizar desde cualquier otro programa. Es habitual encontrar como private las variables de instancia y public los métodos.

Herencia

La herencia es una de las características más importantes de la POO. Si definimos una serie de atributos y métodos para una clase, al crear una subclase, todos estos atributos y métodos siguen siendo válidos.
En el apartado anterior se define la clase Animal. Uno de los métodos de esta clase es duerme. A continuación podemos crear las clases Tigre y Ave como subclases de Animal. De forma automática, se puede utilizar el método duerme con las instancias de las clases Tigre y Ave.

La clase Ave, subclase de Animal, hereda todos sus atributos y métodos. A su vez de esa clase Ave podemos crear la clase Gallo. Esta clase Gallo heredaría todos los atributos y métodos de Ave y también los de Animal.
De la misma forma que se heredan cosas de padres a hijos en la vida real, en POO se heredan atributos y métodos de clases a subclases.

Para crear en Java una subclase de otra clase existente se utiliza la palabra reservada extends. A continuación se muestra el código de las clases Tigre, Ave y Pinguino, así como el programa que prueba estas clases creando instancias y aplicándoles métodos. Recuerda que la definición de la clase Animal se muestra en el apartado anterior.

```
//Definición de la clase Tigre
public class Tigre extends Animal {
	private String raza;
	public Tigre() {
		super();
		raza = "Bengala";
	}
	public Tigre (String tipo, String clase, String r) {
		super(tipo,clase);
		raza = r;
	}
	public Tigre (String r) {
		super();
		raza = r;
	}

	public String toString() {
		return "Tigre - Raza: " + this.raza + "\n" + super.toString();
	}

	//el tigre ruge
	public void sonido() {
		System.out.println("Grrrrrr");
	}
```

```
	//el tigre come
	public void come(String comida) {
		if (comida.equals("pescado")) {
			System.out.println("No quiero pescado");
		} else {
			System.out.println("Estoy hambriento!!");
		    }
	}
```

Se han definido tres constructores en la clase Tigre. Desde el programa principal se pueden utilizar en función del número y tipo de parámetros que se pasa al método. Por ejemplo, si desde el programa principal se crea un tigre de la forma Tigre miTigre = new Tigre(); entonces se llamaría al constructor definido como

```
	public Tigre() {
		super();
		raza = "Bengala";
	}
```

Por tanto miTigre sería un tigre de raza Bengala, salvaje y sin nombre.

- Bengala porque es el valor que se mete por defecto.
- Salvaje y sin nombre porque al ser una clase que hereda de otra, con super(), estamos ejecutando además el constructor por defecto de la superclase Animal.

```
public Animal () {
 this.tipo = "salvaje";
 this.nombre = "sin nombre";
}
```

En cambio si desde el programa principal se crea un tigre usando otro constructor por ejemplo así: Tigre tigreton = new Tigre("Siberia"); entonces estaría usando el constructor con un parámetro de tipo String.

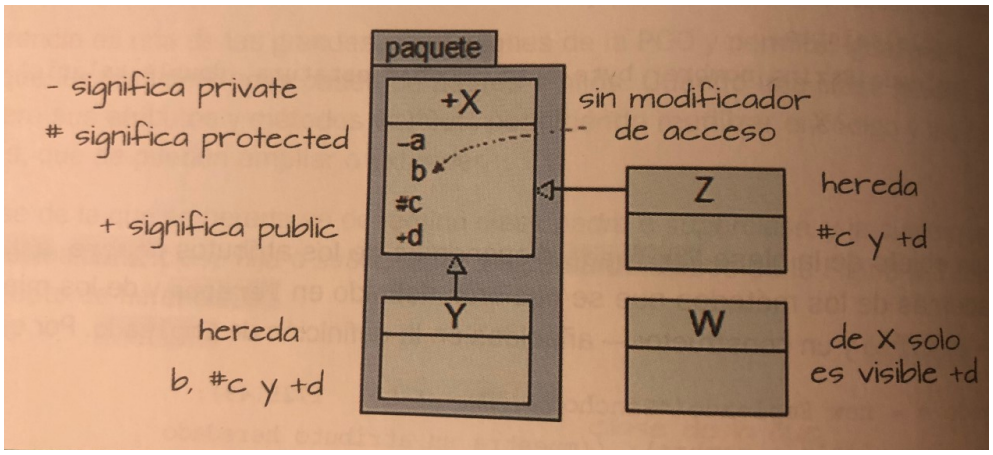
```
public Tigre (String r) {
	super();
	raza = r;
}
```

En este caso tigretón sería un tigre de raza siberiano, salvaje y sin nombre.

A continuación definición la clase Ave como una subclase de Animal.

```
/*Definición de la clase Ave */
public class Ave extends Animal {
	public Ave() {
		super();
	}
	public Ave (String tipo, String nombre) {
		super(tipo, nombre);
	}

	public String toString() {
		return "Ave " + super.toString()};
	}

	//El ave se limpia
	public void limpieza() { System.out.println("Me limpio las plumas"); }

	//El ave vuela
	public void vuela() { System.out.println("Estoy volando"); }
}
```

Ámbito de visibilidad con clases heredadas



Sobrecarga de métodos

Un método se puede redefinir (volver a definir con el mismo nombre) en una subclase. Por ejemplo, el método vuela que está definido en la clase Ave se puede volver a definir en la clase Gallo, con el mismo nombre pero comportándose de forma distinta. Indicaremos nuestra intención de sobreescribir un método mediante la etiqueta @Override. Al ponerlo el programa nos informará si estamos haciendo algo mal en la sobrescritura. Si no escribimos esta etiqueta, la sobreescritura del método se realizará de todas formas ya que @Override indica simplemente una intención.
Por ejemplo, si sobreescribes el método come de Animal declarando un método come específico para los tigres y escribes @Override,si te equivocas en el nombre del método, en el número o en el tipo de parámetros (por ejemplo, escribimos comer), el compilador diría: "¡algo no está bien!, me has dicho que ibas a sobreescribir un método de la superclase y sin embargo el método comer no está definido en tu clase padre.
A continuación se muestra la clase Gallo.

```
public class Gallo extends Ave {
	public Gallo() {
		super();
	}
	public Gallo(String nombre) {
		super(nombre);
	}
	@Override //con @Override decimos que vamos sobrescribir el método de su padre
	public void vuela() { System.out.println("Me cuesta volar"); }
	//el objeto miGallo actúa de manera polimórfica, con el método vuela.
	Polimorfismo en POO es la capacidad que tienen los objetos de distinto
	tipo (de distintas clases) de responder al mismo método.
}
```

Con el siguiente programa se prueba la clase Animal y todas las subclases que derivan de ella.

```
/*Programa que prueba la clase Animal y sus subclases */
public class Prueba_Animal {
	public static void main(String[] args) {
		Animal miAnimal = new Animal("domestico","Bobi");
		System.out.println(miAnimal);

		System.out.println();
		Tigre miTigre = new Tigre();
		System.out.println(miTigre);
		miTigre.sonido();
```

```
            miTigre.come("carne");
            miTigre.duerme();

            System.out.println();
            Ave miAve = new Ave();
            System.out.println(miAve);
            miAve.vuela();
            miAve.limpieza();
            //miAve no tiene el método come

            System.out.println();
            Gallo miGallo = new Gallo();
            System.out.println(miGallo); // toString viene de Ave
            miGallo.vuela(); // vuela viene de Gallo
            miGallo.limpieza(); // limpieza viene de Ave
            miGallo.duerme(); // duerme viene de Animal
        }
}
```

Atributos y métodos de clase (static)

Hemos definido atributos de instancia como raza, nombre o color y métodos de instancia como limpieza, come o vuela. Si en el programa se crean 5 tigres, cada uno de ellos tiene su propia raza (es un atributo de la instancia).
En determinadas ocasiones, nos puede interesar tener atributos de clase (variables de clase) y métodos de clase. **Cuando se define una variable de clase solo existe una copia del atributo para toda la clase y no una para cada objeto**. Esto es útil cuando se quiere llevar la cuenta global de algún parámetro. Los métodos de clase se aplican a la clase y no a instancias concretas.
A continuación se muestra un ejemplo que contiene la variable de clase *kilometrajeTotal*. Cada coche tiene un atributo kilometraje donde se acumulan sus propios kilómetros y en la variable de clase *kilometrajeTotal* (de tipo static) se cuentan los kilómetros que han recorrido todos los coches creados. El método de clase llamado *getKilometrajeTotal* es un getter para la variable de clase *kilometrajeTotal*.

```
public class Coche {

	private String marca;
	private String modelo;
	private int kilometraje;

	// atributo de clase
	private static int kilometrajeTotal = 0;

	public Coche(String ma, String mo) {
		marca = ma;
		modelo = mo;
		kilometraje = 0;
	}

	public int getKilometraje() {
		return kilometraje;
	}

	//Recorre una determinada distancia
	public void recorre(int km) {
		kilometraje += km;
		kilometrajeTotal += km;
	}

	// método de clase
	public static int getKilometrajeTotal() {
		return kilometrajeTotal;
	}

}
```

El atributo kilometraje almacena los kilómetros recorridos por un objeto concreto y tendrá un valor distinto para cada uno de ellos. Si en el programa principal se crean 20 objetos de la clase Coche, cada uno tendrá su propio kilometraje.

A continuación se muestra el programa que prueba la clase Coche.

```
public class Prueba_Coche {
public static void main(String[] args) {
 Coche cocheDeLuis = new Coche("Renault", "Megane");
 Coche cocheDeJuan = new Coche("Toyota", "Avensis");
 cocheDeLuis.recorre(30);
 cocheDeLuis.recorre(40);
 cocheDeLuis.recorre(230);

 cocheDeJuan.recorre(60);
 cocheDeJuan.recorre(140);

System.out.println("El coche de Luis:"+cocheDeLuis.getKilometraje()+"Km");

System.out.println("El coche de Juan:"+cocheDeJuan.getKilometraje()+"Km");

System.out.println("Km totales:"+Coche.getKilometrajeTotal()+ "Km");

}
}
```

El método getKilometrajeTotal() se aplica a la clase Coche por tratarse de un método de clase (método static). Este método no se podría aplicar a una instancia, de la misma manera que un método que no sea static no se puede aplicar a la clase sino a los objetos.

Método equals

El método equals(), se utiliza para comparar dos objetos.
Ojo!, no confundir con el operador ==, que nos servía para comparar tipos de datos primitivos (int, double,…). Si lo usamos con objetos estaríamos comparando referencias de memoria.

equals() se usa para saber si dos objetos son del mismo tipo y tienen los mismos datos. Nos dara el valor true si son iguales y false si no.

Podemos sobreescribir el método equals() para hacer una comparación entre dos objetos, con el criterio o criterios de comparación que queramos.

En la lista de argumentos del método equals() hay que pasarle un argumento de tipo Object. No olvides poner el @Override, sino se sobrecarga el método, no se sobreescribe.

Supongamos que en la clase Gato redefino el método equals de la siguiente manera:

```
@Override
	public boolean equals(Object obj) {

	    Gato other = (Gato) obj;
	  if (!color.equals(other.color))
	            return false;
	  else
	            return true;

	}
```

Así, en la otra clase donde tenga el main, puedo comparar objetos de tipo Gato, comprobando solo su color.

```
Gato garfield = new Gato("persa");
Gato isidoro = new Gato("persa");

garfield.setColor("rojo");
isidoro.setColor("verde");

System.out.println(garfield);
System.out.println(isidoro);


if (garfield.equals(isidoro))
	System.out.println("Los gatos tienen el mismo color");
else System.out.println("Los gatos son distintos");
```

Arrays de objetos

Del mismo modo que se pueden crear arrays de números enteros, decimales o cadenas de caracteres, también es posible crear arrays de objetos.
Definimos la clase Alumno y luego creamos un array de objetos de esa clase.

```
public class Alumno {
	private String nombre;
	private double notaMedia = 0.0;

	public String getNombre() {
		return nombre;
	}
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
	public double getNotaMedia() {
		return notaMedia;
	}
	public void setNotaMedia(double notaMedia) {
		this.notaMedia = notaMedia;
	}
}
```

A continuación se define un array de cinco alumnos que posteriormente se rellena. Por último se muestran los datos de los alumnos por pantalla.
La siguiente línea únicamente define la estructura del array pero no crea los objetos:
Alumno[] alum = new Alumno[5];
Cada objeto concreto se crea de forma individual mediante alum[i] = new Alumno();

```
/*Programa que prueba un array de la clase Alumno*/
import java.util.Scanner;
public class PruebaAlumnos {
public static void main(String[] args) {

	String nombre;
	double nota;
	double sumaDeMedias = 0;

	Scanner sc = new Scanner(System.in);

	// Define un array de 5 alumnos pero no crea los objetos
	Alumno[] alum = new Alumno[5];

	// Pide los datos de los alumnos
	System.out.println("Introduce nombre y nota media de 5 alumnos.");

	for(int i = 0; i < 5; i++) {
		alum[i] = new Alumno();
		System.out.println("Alumno:" + i);

		System.out.print("Nombre: ");
		nombre = sc.next();
		alum[i].setNombre(nombre);

		System.out.print("Nota media: ");
		nota = sc.nextDouble();
		alum[i].setNotaMedia(nota);
	}
```

```
        // Muestra los datos de los alumnos
        System.out.println("Los datos introducidos son:");
        sc.close();

        for(int i = 0; i < 5; i++) {
              System.out.println("Alumno " + i);
              System.out.println("Nombre: " + alum[i].getNombre());
              System.out.println("Nota media: " + alum[i].getNotaMedia());
              System.out.println("--------------------------------");
              sumaDeMedias += alum[i].getNotaMedia();
        }
        System.out.println("La media de la clase es " + sumaDeMedias/5);

        }
}
```

Instanceof

Es un operador en Java que se utiliza para determinar si un objeto es una instancia de una determinada clase o interfaz. Devuelve un valor booleano, true si el objeto es una instancia de una clase o interfaz dada o false en otro caso.

No puede usarse con tipos primitivos, solo puede usarse con objetos.

```
class Figura {
    // atributos y métodos
}
class Circulo extends Figura {
    // atributos y métodos
}
class Cuadrado extends Figura {
    // atributos y métodos
}

public class Main {
    public static void main(String[] args) {
        Figura[] figuras = new Figura[3];
        //puedo meter círculos o cuadrados en el array porque son figuras (heredan de figura)
        figuras[0] = new Circulo();
        figuras[1] = new Cuadrado();
        figuras[2] = new Circulo();

        for (int i = 0; i < figuras.length; i++) {
            if (figuras[i] instanceof Circulo) {
                System.out.println("Figura en el indice " + i + " es un circulo");
            } else if (figuras[i] instanceof Cuadrado) {
                System.out.println("Figura en el indice " + i + " es un cuadrado");
            }
        }
    }
}
```

Composición de otras clases

Se crea una clase Huevo que tiene como atributos dos objetos de las clases Yema y Clara, lo cual indica que una instancia de la clase Huevo está compuesta por dos instancias de las clases Yema y Clara. Esto se conoce como composición, ya que una clase contiene objetos de otras clases como parte de su estructura.

La relación entre Yema, Clara y Huevo puede ser representada como una "tiene un" o "está compuesto por" ya que un objeto de la clase Huevo tiene un objeto de la clase Yema y un objeto de la clase Clara.

```
class Huevo {
    private Yema yema;
    private Clara clara;

    public Huevo() {
        yema = new Yema();
        clara = new Clara();
    }

    public Yema getYema() {
        return yema;
    }

    public Clara getClara() {
        return clara;
    }

    public void romper() {
        System.out.println("El huevo ha sido roto.");
    }

    public void cocinar() {
        System.out.println("El huevo ha sido cocido.");
    }
}

    public Huevo(Yema y, Clara c) {
        this.yema = y;
        this.clara = c;
    }
```

```java
class Yema {
    private String color;

    public Yema() {
        color = "amarillo";
    }

    public String getColor() {
        return color;
    }

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

```java
class Clara {
    private double volumen;

    public Clara() {
        volumen = 0.0;
    }

    public double getVolumen() {
        return volumen;
    }

    public void setVolumen(double volumen) {
        this.volumen = volumen;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Huevo huevo = new Huevo();

        huevo.romper();
        huevo.cocinar();

        Yema yema = huevo.getYema();
        System.out.println("Color de la yema: " + yema.getColor());
        yema.setColor("verde");
        System.out.println("Nuevo color de la yema: " + yema.getColor());

        Clara clara = huevo.getClara();
        System.out.println("Volumen de la clara: " + clara.getVolumen());
        clara.setVolumen(0.5);
        System.out.println("Nuevo volumen de la clara: " +
clara.getVolumen());
    }
}
```

Clases internas

Ejemplo: Huevo, Yema y Clara

Definimos Yema y Clara como clases internas de Huevo, tienen acceso a los atributos y métodos de la clase contenedora, por eso en el constructor de Huevo puede acceder a las variables yema y clara.

```
public class Huevo {

	private Yema yema;
	private Clara clara;

	public class Yema {
		private int tamanio;
		public Yema (int tama) {
			tamanio=tama;
		}
		public int getTamanio() {
			return tamanio;
		}
	}

	public class Clara {
		private int volumen;
		public Clara(int volumen) {
			this.volumen=volumen;
		}

		public int getVolumen() {
			return volumen;

		}
	}

	public Huevo(Yema y, Clara c) {
		this.yema = y;
		this.clara=c;
	}

	public Huevo(int tamanioYema, int volumenClara) {
	  this.yema = new Yema(tamanioYema);
	  this.clara = new Clara(volumenClara);
    }

	public Yema getYema() {
		return yema;
	}

	 public Clara getClara() {
		 return clara;
	 }

}

public class Principal {

	public static void main(String[] args) {

		Huevo h1 = new Huevo(10,20);
		System.out.println(h1.getYema().getTamanio());

	}
}
```

Se ha creado una instancia de la clase Huevo con una yema de tamaño 10 y una clara de volumen 20. Luego se imprimen en pantalla el tamaño con los métodos getter correspondientes.

La principal diferencia con las clases independientes Yema y Clara que vimos en el apartado anterior es la encapsulación: **las clases internas permiten ocultar la complejidad de la implementación de la clase contenedora**, mientras que **las clases independientes no tienen acceso a los atributos y métodos privados de la clase contenedora**.

Se recomienda usar clases internas cuando se quiere ocultar la complejidad de una clase y cuando se quiere que las clases internas tengan acceso a los atributos y métodos privados de la clase contenedora. Sin embargo, si las clases internas van a ser utilizadas en múltiples clases, es mejor definirlas como clases independientes.

Ejemplo: Automóvil

```
class Automovil {
    private String marca;
    private int velocidad;

    private class Rueda {
        private int tamaño;

        public Rueda(int tamaño) {
            this.tamaño = tamaño;
        }

        public int getTamaño() {
            return tamaño;
        }
    }

    public Automovil(String marca, int velocidad, int tamañoRueda) {
        this.marca = marca;
        this.velocidad = velocidad;
        this.rueda = new Rueda(tamañoRueda);
    }

    public Rueda getRueda() {
        return rueda;
    }
```

La clase interna Rueda tiene acceso a los atributos y métodos de la clase Automovil, por eso se puede acceder al atributo "rueda" en el constructor de Automovil.
"rueda" es una variable de instancia que se refiere a una instancia de la clase interna Rueda. En el constructor se esta creando una nueva instancia de la clase Rueda, pasando el argumento "tamañoRueda" como parámetro para el constructor de Rueda y asignándola a la variable rueda.

Ejemplo: Lista de objetos de distinto tipo

La clase Lista tiene una clase interna llamada Nodo. Cada objeto de la clase Nodo representa un elemento en la lista, tiene un atributo "elemento" y un atributo "siguiente" que apunta al siguiente nodo en la lista. Es declarada como clase interna para que solo sea accesible desde la clase Lista y no pueda ser utilizada fuera de esta, nos asegura que los objetos Nodo solo se crean y se utilizan dentro de un objeto de la clase Lista.

La clase Lista tiene una variable de instancia "primero" y "ultimo" que apuntan al primer y último nodo en la lista y una variable de instancia "tamaño" que guarda el número de elementos en la lista. Se podrán agregar distintos tipos de objetos a la lista ya que la clase Lista y la clase Nodo tienen un atributo "elemento" de tipo Object.

```
class Lista {
    private Nodo primero;
    private Nodo ultimo;
    private int tamaño;

    private class Nodo {
        private Object elemento;
        private Nodo siguiente;

        public Nodo(Object elemento) {
            this.elemento = elemento;
            this.siguiente = null;
        }

        public Object getElemento() {
            return elemento;
        }

        public Nodo getSiguiente() {
            return siguiente;
        }
    }
   public Lista() {
       primero = ultimo = null;
       tamaño = 0;
   }

   public void agregar(Object elemento) {
       Nodo nuevo = new Nodo(elemento);
       if (primero == null) {
           primero = nuevo;
       } else {
           ultimo.siguiente = nuevo;
       }
       ultimo = nuevo;
       tamaño++;
   }
```

```
    public Object obtener(int index) {
        Nodo actual = primero;
        for (int i = 0; i < index; i++) {
            actual = actual.siguiente;
        }
        return actual.elemento;
    }

    public int tamaño() {
        return tamaño;
    }
}

    public Object primero(){
        if(primero == null) return null;
        return primero.elemento;
    }

    public boolean borrar(Object elemento) {
        Nodo actual = primero;
        Nodo anterior = null;
        while (actual != null) {
            if (actual.elemento.equals(elemento)) {
                if (anterior == null) {
                    primero = actual.siguiente;
                } else {
                    anterior.siguiente = actual.siguiente;
                }
                tamaño--;
                return true;
            }
            anterior = actual;
            actual = actual.siguiente;
        }
        return false;
    }
    public boolean buscar(Object elemento) {
        Nodo actual = primero;
        while (actual != null) {
            if (actual.elemento.equals(elemento)) {
                return true;
            }
            actual = actual.siguiente;
        }
        return false;
    }
}
```

```
 public Object buscar(Object elemento) {
        Nodo actual = primero;
        while (actual != null) {
            if (actual.elemento.equals(elemento)) {
                return actual.elemento;
            }
            actual = actual.siguiente;
        }
        return null;
    }
```

Es importante mencionar que el método equals() de la clase Object debe ser sobreescrito en las clases de los objetos que se guarden en la lista, para que el método buscar(Object elemento) funcione correctamente.

Un ejemplo de uso:

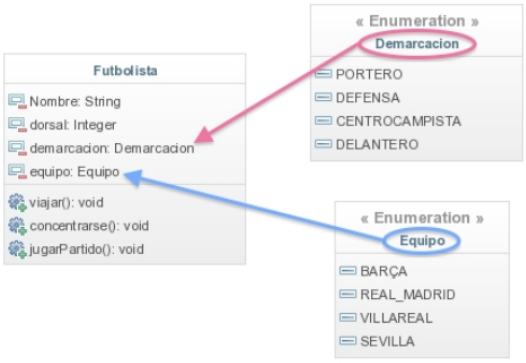
```
class Main {
    public static void main(String[] args) {
        Lista lista = new Lista();
        lista.agregar("Hola");
        lista.agregar(123);
        lista.agregar(true);
        lista.agregar(new Persona("Juan"));
        System.out.println("Tamaño de la lista: " + lista.tamaño());
        System.out.println("Elemento en el índice 0: " +
lista.obtener(0));
        System.out.println("Elemento en el índice 1: " +
lista.obtener(1));
        System.out.println("Elemento en el índice 2: " +
lista.obtener(2));
        System.out.println("Elemento en el índice 3: " +
((Persona)lista.obtener(3)).getNombre());
    }
}
class Persona {
    private String nombre;
    public Persona(String nombre) {
        this.nombre = nombre;
    }
    public String getNombre() {
        return nombre;
    }
}
```

Tipo enumerado

Los enumerados en Java son un tipo especial de datos que se utilizan para representar un conjunto finito de valores.
Mediante enum se puede definir un tipo enumerado, de esta forma un atributo solo podrá tener uno de los posibles valores que se dan como opción. Los valores que se especifican en el tipo enumerado se suelen escribir con todas las letras en mayúscula.

En un fichero que podríamos llamar demarcacion.java

```
public enum Demarcacion
{
    PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO
}
```



Uno de los atributos de Futbolista, podría ser el tipo de dato enumerado Demarcacion.

```
public class Futbolista {

        private int dorsal;
        private String Nombre;
        private Demarcacion demarcacion;
        private Equipo equipo;

        public Futbolista() {
        }

        public Futbolista(String nombre, int dorsal, Demarcacion demarcacion, Equipo equipo) {
                this.dorsal = dorsal;
                Nombre = nombre;
                this.demarcacion = demarcacion;
                this.equipo = equipo;
        }

        // Metodos getter y setter
```

Desde otra clase, con el método main, podemos crear futbolistas, teniendo en cuenta que el constructor es (String, int, Demarcacion,Equipo). Sólo podremos crear futbolistas con los valores posibles que tengamos en los ltipos enumerados Demarcacion y Equipo.

```
Futbolista casillas = new Futbolista("Casillas", 1, Demarcacion.PORTERO, Equipo.REAL_MADRID);
Futbolista capdevila = new Futbolista("Capdevila", 11, Demarcacion.DEFENSA, Equipo.VILLAREAL);
Futbolista iniesta = new Futbolista("Iniesta", 6, Demarcacion.CENTROCAMPISTA, Equipo.BARÇA);
Futbolista navas = new Futbolista("Navas", 22, Demarcacion.DELANTERO, Equipo.SEVILLA);
```

Otro ejemplo de tipo de dato enumerado

Llamamos a un método pinta que recibe como parámetro un color del tipo de dato Colores. Pintará por pantalla un mensaje con el color seleccionado. *Se ha cambiado el color del texto anteponiendo al mensaje caracteres ANSI y reseteamos siempre para que no se mantenga ese color.*

```
package enumerados;

enum Colores { ROJO, VERDE, AZUL }

public class PruebaColores {

    public static final String BLACK = "\u001B[30m";
    public static final String RED = "\u001B[31m";
    public static final String GREEN = "\u001B[32m";
    public static final String YELLOW = "\u001B[33m";
    public static final String BLUE = "\u001B[34m";
    public static final String PURPLE = "\u001B[35m";
    public static final String CYAN = "\u001B[36m";

    public static final String RESET = "\u001B[0m";


        public static void pinta(Colores color) {
            switch (color) {
                case ROJO:
                    System.out.println(RED+"El color es rojo"+RESET);
                    break;
                case VERDE:
                    System.out.println(GREEN+"El color es verde"+RESET);
                    break;
                case AZUL:
                    System.out.println(BLUE+"El color es azul"+RESET);
                    break;
            }
        }
        public static void main(String[] args) {
            pinta(Colores.ROJO);

        }

}
```

Wrappers

A veces puede resultar útil trabajar con los datos primitivos (int, boolean, float, double,...) como si fueran objetos, para trabajar con los métodos que nos proporciona el API de Java de una forma más sencilla (conversiones entre tipos de datos por ejemplo).

Como los datos primitivos NO son objetos, Java lo resuelve utilizando los Wrappers (clases envoltorio), estas clases "envuelven" a los datos primitivos para que se puedan tratar como objetos.

Las clases envoltorio existentes son:

- Byte para byte
- Short para short
- Integer para int
- Long para long
- Boolean para boolean
- Float para float
- Double para double
- Character para char

Ejemplos de uso: convertimos String a int

- Tenemos un String con espacios en blanco y queremos convertirlo a numero. Con parseInt(String) devolvemos directamente un primitivo int.

  ```
  String snumero = "  6  ";
  int numero = Integer.parseInt(snumero.trim());
  ```

- Convertimos un String a un entero. Con valueOf(String) se devuelve un objeto, por eso ejecutamos el método intValue para poder asignarlo a un primitivo int.

  ```
  int  a = Integer.valueOf("22").intValue();
  ```

Otras operaciones con wrappers

Declarando objetos de tipos Integer, podemos usar sus métodos

```
Integer i1 = 6;
Integer i2 = Integer.max(3, 2);
```

¿Por qué no trabajamos directamente con Wrappers?

A día de hoy los tipos primitivos, siguen utilizándose principalmente por el rendimiento que ofrecen, al no tener métodos como los objetos, son tipos de datos más sencillos.
A pesar de que son más eficientes para determinados programas, hay programadores que prefieren utilizar directamente solo objetos.

Clase abstracta (abstract)

Una clase abstracta es aquella que no va a tener instancias de forma directa, aunque sí habrá instancias de las subclases (siempre que esas subclases no sean también abstractas). Por ejemplo, si se define la clase Animal como abstracta, no se podrán crear objetos de clase Animal, es decir, no se podrá hacer Animal mascota = new Animal(), pero sí se podrán crear instancias de la clase Tigre, Ave o Gallo que son subclases de Animal.

Las clases abstractas funcionan como una clase que declara la existencia de métodos pero no su implementación. La implementación de los métodos la haremos después en las diferentes subclases derivadas de la clase abstracta.
Una clase abstracta puede contener métodos no abstractos pero al menos uno de los métodos debe ser abstracto.

Los métodos también se pueden definir como abstracto, cuando se define como abstracto automáticamente la clase a la que pertenece también hay que definirla como abstracta.
Un método no lleva llaves de apertura ni contenido, es solo la definición.
Por ejemplo Public abstract String dameDescripcion ();

Añadir un método abstracto implica seguir un patrón de diseño en toda la jerarquía de clases que tengas: le estás diciendo a las subclases que es obligatorio sobrescribirlo puesto que en su respectiva clase abstracta está vacío.
Ejemplo: definimos una clase abstracta Figura, con el método abstracto area().

```
public abstract class Figura {
    private int numeroLados;

    public Figura() {

        this.numeroLados = 0;
    }
public abstract float area();
}
```

Si ahora queremos que la clase Triangulo herede de Figura, hacemos lo siguiente:

```
public class Triangulo extends Figura {
…
       public float area() {
          System.out.println((base*altura)/2)
       }
…
}
```

Al heredar de una clase abstracta es obligatorio implementar todos sus métodos abstractos, es decir debemos definir el comportamiento, en este caso, en triángulo estaremos obligados a definir cómo se calcula el área del triángulo a realizar la tarea.

Otro ejemplo:

```
abstract class Automovil {
    String marca;
    public abstract void eslogan();
    public String getMarca() {
        return marca;
    }
}

class Deportivo extends Automovil {
    public void eslogan() {
        System.out.println("Veloz como el rayo"); }
    }

class Turismo extends Automovil {
    public void eslogan() {
        System.out.println("Para el uso diario"); }
    }

class Familiar extends Automovil {
    public void eslogan() {
        System.out.println("Cabe hasta el gato"); }
    }

// Clase que contiene el método main
class Herencia {

    public static void main(String [] args) {
        Automovil [] auto=new Automovil [3];
        auto[0]=new Deportivo();
        auto[1]=new Turismo();
        auto[2]=new Familiar();
        imprime_eslogan(auto);
    }

    public static void imprime_eslogan(Automovil [] auto) {
        for (int i=0; i<=2; i++)
                auto[i].eslogan();
    }
}
```

Interfaces

Los Interfaces son clases completamente abstractas que contienen únicamente la cabecera de una serie de métodos (**métodos abstractos)** y opcionalmente también pueden contener **constantes**. Al igual que en las clases abstractas en un Interface se especifica qué se debe hacer pero no su implementación.

Se encarga de especificar un comportamiento que luego tendrá que ser implementado.
Una interfaz puede ser útil en determinadas circunstancias. En principio, separa la definición de la implementación, el "qué" del "cómo".
Tendremos al menos dos ficheros, la interfaz y la clase que implementa esa interfaz.
Se puede dar el caso que un programador escriba la interfaz y luego se la pase a otro programador para que sea éste último quien la implemente.

A tener en cuenta:

- La interfaz se almacena en un fichero .class
- No se pueden instanciar (no uso de new)
- Todos sus métodos son públicos y abstractos. No se implementan.
- Si no pones public y abstract, se da por supuesto
- No pueden tener variables
- Sí pueden tener constantes
- Una interfaz puede tener varias implementaciones.
- La interfaz indica "qué" hay que hacer y la implementación el "cómo" se hace.
- La implementación puede contener métodos adicionales cuyas cabeceras no están en su interfaz.

```
interface MiInterfaz {
	int CONSTANTE = 100;
	int metodoAbstracto (int parametro);
}

class ImplementaInterfaz implements MiInterfaz {
	//se puede usar la constante definida en el interfaz
	int multiplicando=CONSTANTE;

	//es obligatorio la implementación del método
	int metodoAbstracto (int parametro){
		return (parametro * multiplicando);
	}
}
```

¿Qué las diferencia respecto a las clases abstractas?. Las clases abstractas podía contener métodos que no sean abstractos, en cambio en las interfaces **todos los métodos son abstractos.** Principalmente aportan solución al problema de herencia simple, **una clase solo puede heredar de otra clase, pero en cambio una clase puede implementar varios interfaces**

Ejemplo: Class Gato **extends** Animal **implements** Mascota, Raza
La superclase que va a estar por encima de todas las demás será la clase Animal. Eso significa que Gato hereda de Animal e implementa los interfaces Mascota y Raza.

```java
/*Definición de la interfaz Mascota*/
public interface Mascota {
int getCodigo();

void hazRuido();
void come(String comida);
}
```

Únicamente se escriben las cabeceras de los métodos que debe tener la/s clase/s que implemente/n la interfaz Mascota.
Cada interfaz puede tener varias implementaciones asociadas, por ejemplo veremos las clases Gato y Perro que implementan la interfaz Mascota.

```java
public class Gato extends Animal implements Mascota {
	private int codigo;

	public Gato (String nombre, int c) {
		super(nombre);
		this.codigo = c;
	}

	@Override
	public int getCodigo() {
		return this.codigo;
	}

	//el gato emite sonido
	@Override
	public void hazRuido() {
		this.maulla();
		this.ronronea();
	}

	//el gato maulla
	public void maulla() {
		System.out.println("Miauuuu");
	}

	//el gato ronronea
	public void ronronea() {
		System.out.println("mrrrrrr");
	}

	//el gato come
	@Override
	public void come(String comida) {
		if (comida.equals("pescado")) {
		super.come();
		System.out.println("Que rico el pescado!, gracias");
		} else {
			System.out.println("Lo siento, solo como pescado");
		}
	}

	public String toString() {
		return "Gato mascota" + "- Nombre:"+super.getNombre()+
			"- Código:"+getCodigo();
	}
}
```

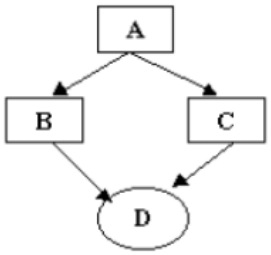
Ojo, no es lo mismo herencia que la implementación.

Observa que los métodos que se indicaban en Mascota con la cabecera (hazRuido y come) ahora están implementados completamente en Gato. Además, Gato contiene otros métodos que no se indicaban en Mascota como maulla y ronronea.

```
public class Perro extends Animal implements Mascota {
	private int codigo;
	public Perro (String nombre, int c) {
		super(nombre);
		this.codigo = c;
	}

	@Override
	public int getCodigo() { return this.codigo; }

	@Override//el perro emite sonido
	public void hazRuido() {
		this.ladra();
	}
	//el perro ladra
	public void ladra() {
		System.out.println("Guau guau!");
	}

	@Override //el perro come
	public void come(String comida) {
		if (comida.equals("pescado")) {
			System.out.println("Pescado noooooo!");
		} else {
			super.come();
			System.out.println("comida rica!");
		}
	}

	public String toString() {
		return "Perro mascota" + "- Nombre:"+super.getNombre()+
		       "-Código:"+getCodigo();
	}
}
public class Prueba_Mascota { //prueba Mascota y sus implementaciones Gato y Perro:
	public static void main(String[] args) {

		System.out.println();
		Gato miGato = new Gato("Isidoro",1);
		System.out.println(miGato);
		miGato.hazRuido();
		miGato.come("pescado");

		System.out.println();
		Perro miPerro = new Perro("Tobi",1);
		System.out.println(miPerro);
		miPerro.hazRuido();
		miPerro.come("pescado");
		miPerro.come("carne");

		System.out.println();
		Mascota garfield = new Gato("Gardfiel", 2);
		System.out.println(garfield);
		garfield.come("carne")
	}
}
```

Observa que para crear una mascota que es un gato también se puede hacer como:
Mascota garfield = new Gato("Gardfield", 2);
Una interfaz no se puede instanciar, por tanto la siguiente línea sería incorrecta:
~~Mascota garfield = new Mascota("Gardfield", 2);~~

Otro ejemplo de herencia múltiple con interfaces



Para poder llevar a cabo un esquema como el anterior en Java es necesario que las clases A, B y C de la figura sean interfaces, y que la clase D sea una clase (que recibe la herencia múltiple):

```
interface A{ }
interface B extends A{ }
interface C extends A{ }
class D implements B,C{ }
```

- Una interfaz puede heredar de más de una interfaz antecesora.

  ```
  interface InterfazMultiple extends Interfaz1,Interfaz2{ }
  ```

- Una clase no puede tener más que una clase antecesora, pero puede implementar más de una interfaz:

  ```
  class MiClase extends SuPadre implements Interfaz1,Interfaz2{ }
  ```

Final en herencia

Clases finales

Cuando una clase se declara con la palabra clave final, se denomina clase final.
Una clase final no se puede extender (heredar). Ejemplo de uso de una clase final:

- Una es definitivamente evitar la herencia, ya que las clases finales no se pueden extender. Por ejemplo, todas las clases Wrapper como Integer, Float, etc. son clases finales. No podemos extenderlos.

```
final class A
{
     // métodos y campos
}
// La siguiente clase es ilegal.
class B extends A
{
    // COMPILACIÓN-ERROR!
}
```

Métodos finales

Cuando se declara un método con la palabra clave final, se denomina método final. Un método final no puede ser anulado. Debemos declarar los métodos con la palabra clave para la cual necesitamos seguir la misma implementación en todas las clases derivadas.

```
class A
{
    final void m1()
    {
        System.out.println("Este es un método final.");
    }
}

class B extends A
{
    void m1()
    {
        // COMPILACIÓN-ERROR!
        System.out.println("Ilegal!");
    }
}
```