

¿Cómo podemos crear e invocar otros métodos para dividir nuestro código?

Dentro de la misma clase, puede que nos venga bien dividir el código en diferentes métodos en lugar de tenerlo todo junto dentro del método “main”. Normalmente, esto se hace persiguiendo estas ventajas:

- a) Claridad en la lectura del código, ya que todo aparece más ordenado
- b) Reutilización de código: si tengo que hacer varias veces lo mismo no es necesario copiar el código, basta con llamar al método que lo hace
- c) Estructurar el código siguiendo el patrón de diseño “divide y vencerás”

Los métodos pueden:

1. Devolver un único valor o no devolver ningún valor. Este valor puede ser cualquier cosa: un String, un Boolean, un Integer, un array, etc.
2. Recibir ninguno, uno o varios valores que se denominan parámetros. También pueden ser de distinto tipo.

La forma de escribir un método es así:

```
public static TIPO_DEV nombreMetodo(TIPO_PARAM1 nombreParam1, TIPO_PARAM2 nombreParam2, ...) {  
  
    // AQUÍ IRÍA EL CÓDIGO  
  
}
```

Donde:

- TIPO_DEV → es el tipo de dato que vamos a devolver. Si no devolvemos nada, escribimos “void”
- TIPO_PARAM1, TIPO_PARAM2... → son los tipos de parámetros que vamos a recibir
- nombreParam1, nombreParam2... → son los nombres de esos parámetros

Veamos ejemplos:

```
public static void main(String[] args) {  
    // Ejemplo de llamada a un método que no devuelve nada ni recibe nada  
    imprimirNumero();  
}  
  
// Ejemplo de método que no devuelve nada (void) y que no recibe nada  
public static void imprimirNumero() {  
    System.out.println(7);  
}
```

```
public static void main(String[] args) {  
    // Ejemplo de llamada a un método que no devuelve nada y recibe dos parámetros  
    imprimirNumero("Hola", 7);  
}  
  
// Ejemplo de método que no devuelve nada (void) y que recibe dos parámetros  
public static void imprimirNumero(String saludo, Integer numero) {  
    System.out.println(saludo + " " + numero);  
}
```

```
public static void main(String[] args) {  
    // Ejemplo de llamada a un método que devuelve un entero y recibe dos parámetros  
    Integer cuadrado = imprimirNumero("Hola", 7);  
}  
  
// Ejemplo de método que devuelve un entero y que recibe dos parámetros  
public static Integer imprimirNumero(String saludo, Integer numero) {  
    System.out.println(saludo + " " + numero);  
    Integer cuadrado = numero*numero;  
    return cuadrado;  
}
```

```
public static void main(String[] args) {  
    // Ejemplo de llamada a un método que devuelve un entero y no recibe nada  
    Integer numero = imprimirNumero();  
}  
  
// Ejemplo de método que devuelve un entero y no recibe nada  
public static Integer imprimirNumero() {  
    Integer numero = 7;  
    System.out.println(numero);  
    return numero;  
}
```

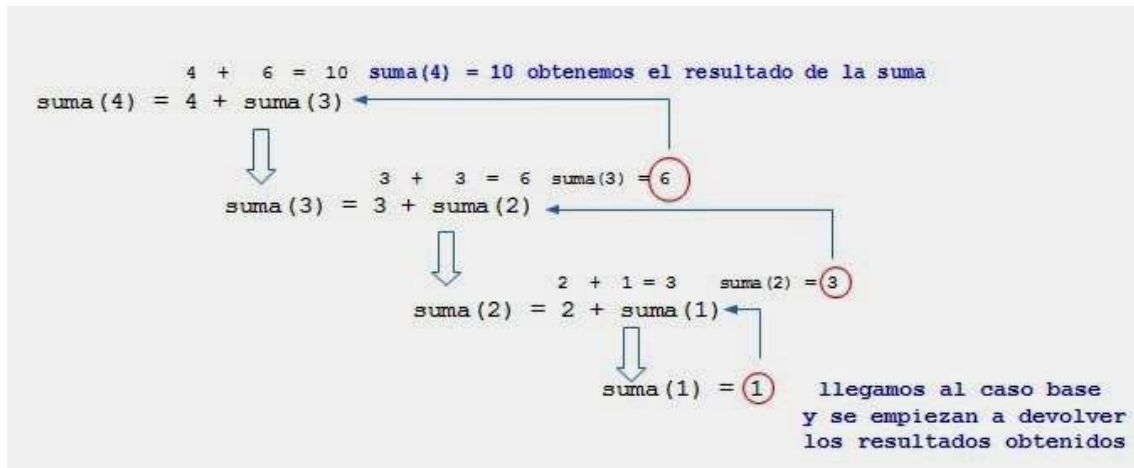
¿Qué es la recursividad?

Con un bucle lo que hacíamos era repetir código. La recursividad es algo similar, pero consiste en llamarse a uno mismo para resolver un problema. Todo lo que podemos hacer con un bucle lo podemos hacer con recursividad.

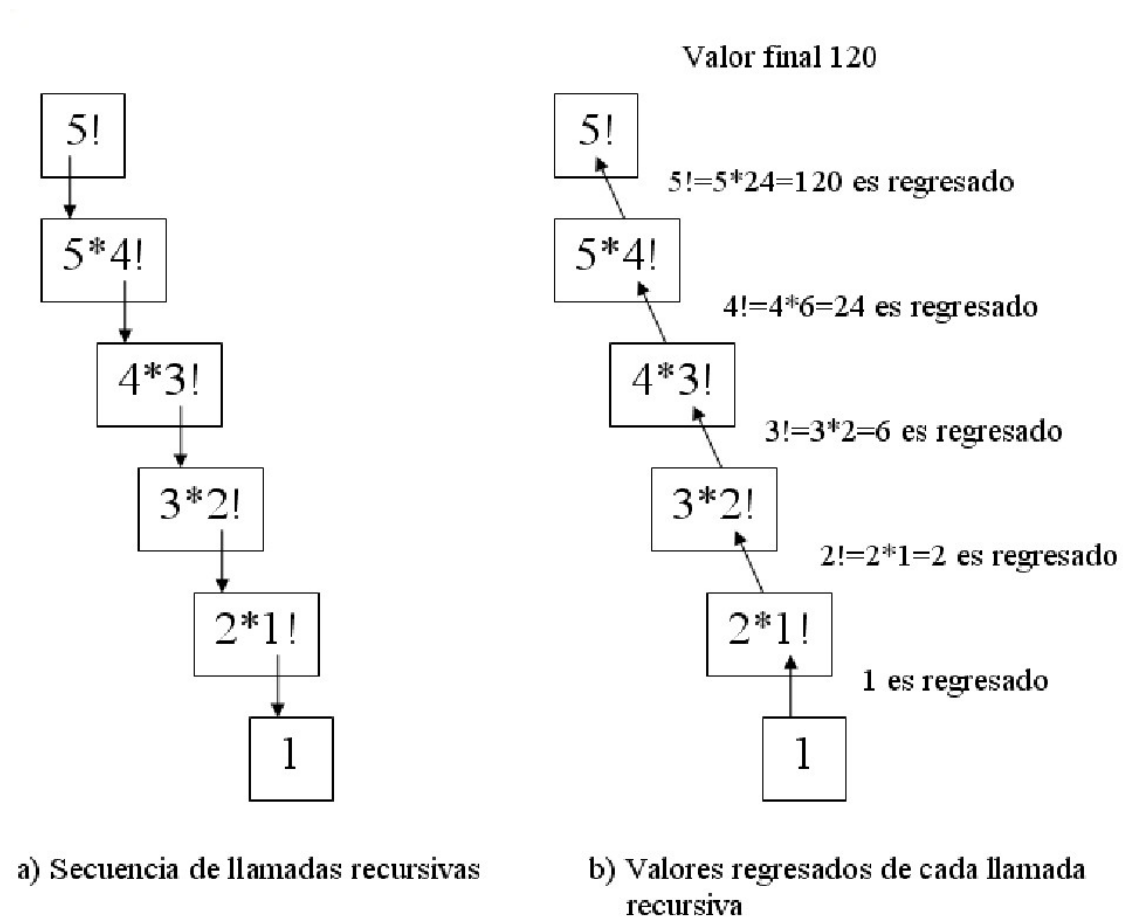
Para aplicarla basta con que un método se llame a sí mismo para repetir el mismo cálculo, pero avanzando en el siguiente caso. El código del método tendrá que distinguir SIEMPRE dos posibilidades:

- a) Se encuentra en el caso base → se da cuando ya NO tiene que llamarse a sí mismo. El caso base siempre tiene que existir, si no, la recursividad será infinita. Aquí, hará un return del resultado para ese caso base. Por ejemplo, si lo que queremos hacer es sumar todos los números de 1 a N, el caso base será cuando N sea igual a 1. En ese caso el resultado es 1 y haremos un return 1.
- b) Se encuentra en un caso normal (no base) → aquí es donde aplicaremos la recursividad llamando de nuevo al mismo método (llamándose a sí mismo), pero en el siguiente paso. Para el ejemplo de sumar todos los números de 1 a N, será el caso en el que N sea mayor a 1. Llamaremos de nuevo al mismo método pasándole N-1.

Ver el ejemplo que se muestra en el siguiente diagrama:



Otro diagrama para el cálculo del factorial:



¿Cómo creamos una clase en Java?

Basta con crear una nueva clase desde Eclipse y darle un nombre. En el código, el fichero debe empezar así:

```
1 package mipaquete;  
2  
3 public class NombreClase {  
4  
5 }  
6
```

El nombre de la clase tiene que estar escrito en camelcase y debe empezar por mayúsculas.

¿Cómo añadimos atributos a la clase?

Los atributos se pueden indicar en cualquier parte dentro de la clase (entre las llaves). Lo correcto por convención es hacerlo al principio de la clase. Se escribirían indicando el tipo y el nombre del atributo. Este nombre debe empezar en minúsculas.

```
1 package mipaquete;  
2  
3 public class NombreClase {  
4  
5     String nombreAtributo;  
6  
7 }
```

Antes del tipo se puede añadir un modificador (`private`, `protected`, `public`). El modificador indica desde dónde es visible ese atributo. Mira el siguiente cuadro que indica desde dónde sería visible o no en función del modificador que escribamos:

MODIFICADOR	CLASE	PACKAGE	SUBCLASE	TODOS
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
No especificado	Sí	Sí	No	No
private	Sí	No	No	No

Ejemplo de uso del modificador `private` en la clase anterior:

```
1 package mipaquete;  
2  
3 public class NombreClase {  
4  
5     protected String nombreAtributo;  
6  
7 }
```

¿Cómo podemos indicar si una clase hereda de otra?

Junto a la declaración de la clase, al final, podemos escribir `extends` y luego el nombre de la clase padre.

```
1 package mipaquete;
2
3 public class NombreClase extends NombreClasePadre{
4
5     protected String nombreAtributo;
6
7 }
```

Si no se indica nada, la clase no hereda de nada de forma explícita. En java, todas las clases heredan por defecto de la clase `Object`. Es como si fuera el padre de todos.

Recuerda que una clase hereda de su padre todos sus atributos, métodos y constructores (salvo lo que sea privado).

¿Cómo podemos crear métodos en la clase?

Los métodos se crean igual que los que ya hemos visto anteriormente. Pero no se indica la palabra `static`. Si se escribe `static`, significa que el método se invoca sin necesidad de crear un objeto.

Se deben escribir, por convención, al final de la clase.

En el método también podemos escribir un modificador de visibilidad (`public`, `protected`, `private`). Significan lo mismo que en el caso de los atributos.

Ejemplo de un método público que no devuelve nada y recibe un parámetro de tipo `String` llamado parametro1:

```
3 public class NombreClase extends NombreClasePadre{
4
5     protected String nombreAtributo;
6
7     public void nombreMetodo(String parametro1){
8
9     }
10 }
```

Ejemplo de un método privado que devuelve un `Integer` y no recibe nada:

```
3 public class NombreClase extends NombreClasePadre{
4
5     protected String nombreAtributo;
6
7     private Integer nombreMetodo(){
8
9     }
10 }
```

¿Cómo podemos crear constructores en la clase?

Un constructor se escribe igual que un método, pero con dos diferencias:

- No devuelve nada. Ni siquiera `void`. Es decir, no se escribe ningún tipo al principio.
- Se debe llamar exactamente igual que la clase, y también empezando con mayúsculas igual que ella.

Los constructores se escriben por convención después de los atributos de la clase. Puede haber tantos como queramos. Incluso ninguno.

Ejemplo de un constructor que recibe un parámetro de tipo `String`:

```
3 public class NombreClase extends NombreClasePadre{
4
5     protected String nombreAtributo;
6
7     • public NombreClase(String parametro1) {
8
9     }
10 }
```

Ejemplo de un constructor que no recibe ningún parámetro:

```
3 public class NombreClase extends NombreClasePadre{
4
5     protected String nombreAtributo;
6
7     • protected NombreClase() {
8
9     }
10 }
```

Si el constructor que no recibe ningún parámetro no hace nada (está vacío) se le conoce como constructor vacío.

Todas las clases, si no se escribe ningún constructor, tienen de manera implícita un constructor vacío para que se puedan crear objetos de ella. Si necesidad de escribir el constructor. No obstante, lo recomendable es crearlo siempre. Algunos *frameworks* podrían necesitarlo.

Los constructores, igual que los métodos, también llevan modificador de visibilidad. Pero, salvo excepciones donde no queramos que se puedan crear objetos de nuestra clase desde determinados lugares, lo normal es que siempre sean públicos.

¿Cómo podemos crear un objeto de una clase?

Las clases que tienen un origen en tipos primitivos (`Integer`, `String`, `Boolean`...) permiten crear objetos de una manera directa como ya sabemos. Pero para el resto de clases es necesario invocar a su constructor con la palabra `new`. Ejemplo para la clase que hemos escrito anteriormente con un constructor sin parámetros:

```
public static void main(String[] args) {
    NombreClase variable = new NombreClase();
}
```

Si tuviéramos un constructor con parámetros, lo invocaríamos igual, pero pasando los parámetros necesarios. Ejemplo:

```
public static void main(String[] args) {  
    NombreClase variable = new NombreClase("ejemplo");  
}
```

Para acceder a sus atributos y métodos, basta con usar la variable que está asociada al nuevo objeto con un punto a continuación.

¿Qué son los métodos get y set?

Normalmente, los atributos que se crean en la clase se desea que no se puedan leer ni escribir de forma directa desde fuera de la clase. Por ello se les indica un modificador privado. Para permitir que se puedan leer desde fuera, se crean métodos `get` públicos. Estos métodos se han llamado así por convención (se podrían llamar de cualquier otro modo). Y únicamente devuelven el valor del atributo. Por ejemplo:

```
public class NombreClase {  
    private String nombreAtributo;  
    public String getNombreAtributo() {  
        return nombreAtributo;  
    }  
}
```

Del mismo modo, si queremos que los atributos se puedan modificar desde fuera de la clase, se crean métodos `set`. También se llaman así por convención. Ejemplo:

```
public class NombreClase {  
    private String nombreAtributo;  
    public void setNombreAtributo(String nuevoValor) {  
        nombreAtributo = nuevoValor;  
    }  
}
```

Eclipse dispone de una ayuda para generar los métodos `get` y `set` de forma rápida. Está en `menú Source >> Generate getter and setter...`

¿Podemos modificar los métodos que heredamos de nuestra clase padre?

Sí, basta con volver a crearlos exactamente con el mismo nombre. Además, escribiremos la anotación `Override` encima, para indicar que está sobrescrito o sobrecargado.

Ejemplo de mi clase padre:

```
public class NombreClasePadre {  
    public void metodoClasePadre() {  
    }  
}
```

Método sobrecargado en la clase hija:

```
public class NombreClase extends NombreClasePadre {  
    @Override  
    public void metodoClasePadre() {  
    }  
}
```

¿Qué es this y super?

Cuando estoy dentro de una clase, es posible que necesite hacer referencia a mí mismo para algo. En ese caso se utiliza `this`.

Ejemplo en un método `set` donde el parámetro que se recibe es igual al nombre del atributo:

```
public class NombreClase {  
    private String nombreAtributo;  
    public void setNombreAtributo(String nombreAtributo) {  
        this.nombreAtributo = nombreAtributo;  
    }  
}
```

Cuando quiero hacer referencia a algo de la clase padre, utilizo `super`. Ejemplo cuando sobrescribo un método y quiero llamar al del padre después de imprimir hola:

```
public class NombreClase extends NombreClasePadre {  
    @Override  
    public void metodoClasePadre() {  
        System.out.println("hola");  
        super.metodoClasePadre();  
    }  
}
```

¿Qué es el método toString?

El método `toString` es un método especial de la clase `Object`. Por tanto, es un método que se hereda en todas las clases. Cada que queremos imprimir (convertir en cadena) un objeto, java llama al método `toString` de ese objeto.

Por defecto, el `toString` de `Object` lo que hace es imprimir el tipo (la clase de ese objeto) y el identificador. Nosotros podemos sobrescribir ese método para modificar la representación como cadena de nuestro objeto.

¿Cómo lo sobrescribimos? Como cualquier otro método: tenemos simplemente que implementarlo y añadir la anotación `@Override`:

```
@Override
public String toString() {
    return identificador + " - " + descripcion;
}
```

Eclipse tiene una opción para generar automáticamente métodos `toString()` a partir los atributos y métodos de la clase. Lo hacemos en el menú `Source > Generate toString()...`

¿Qué es hacer un “casting”?

Un `casting` sirve para forzar un cambio de tipo de un objeto. Ejemplo: Tenemos dos clases, `Gato` y `Felino`. `Gato` hereda de `Felino`.

1. Tengo un objeto de tipo `Gato` y únicamente una variable de tipo `Felino` que apunta al objeto anterior. Esto se puede hacer, porque todo `Gato` es también un `Felino`:

```
Felino felino = new Gato("A");
```

2. Imaginemos que ahora quiero tener una variable de tipo `Gato` apuntando al mismo objeto. Me daría un error. Porque no todos los `Felinos` son `Gatos`:

```
Felino felino = new Gato("A");
Gato gato = felino;
```

3. Sin embargo, yo sé y estoy seguro que el objeto al que apunta la variable “felino” es de tipo `Gato`. Como estoy seguro de ello, puedo hacer un `CASTING`, para forzar a que ese `Felino` sea un `Gato`:

```
Felino felino = new Gato("A");
Gato gato = (Gato) felino;
```

El `casting` se hace poniendo entre paréntesis el nombre de la clase a la que queremos forzar el cambio. Pero es **muy importante** que estemos seguros de que se puede hacer, si no, estaremos enmascarando un error que fallará cuando el programa se ejecute.

¿Cómo puedo saber de qué clase es un objeto?

A veces necesitamos saber de qué clase exacta es un objeto. En el ejemplo anterior, tenemos una variable Felino que puede que no estemos seguros de si apunta a un objeto Gato o no. Para ello utilizamos “instance of” que significa algo así como “es instancia de”.

Ejemplo:

```
Felino felino = new Gato("A");

if (felino instanceof Gato) {
    // se imprimiría esto
    System.out.println("El felino es un gato");
}
else {
    System.out.println("El felino no es un gato");
}
```

También podemos usar el método `getClass()`. Este es un método de la clase `Object`, por lo que se hereda en todas las clases y está disponible en todos los objetos. Este método devuelve la clase de un objeto. Si queremos comparar las clases de dos objetos, podemos utilizarlo así:

```
Felino felino = new Gato("A");
Gato gato = new Gato("B");

if (felino.getClass() == gato.getClass()) {
    System.out.println("la variable felino y la variable gato apuntan a Objetos de la misma Clase");
}
else {
    System.out.println("la variable felino y la variable gato apuntan a Objetos de distinta Clase");
}
```

¿Qué es el método equals?

El método `equals` es un método especial de la clase `Object`. Por tanto, es un método que se hereda en todas las clases. Nos sirve para comparar dos objetos y saber si son iguales o no.

Lo primero que tenemos que tener en cuenta:

- Dos objetos son iguales si el método `equals()` de la clase devuelve true.
- Dos objetos son el mismo objeto no cuando son iguales, sino cuando son el mismo (sería comparar un objeto consigo mismo). Para saber si dos objetos son el mismo objeto (si dos variables apuntan al mismo objeto) podemos utilizar la expresión `==`

```
Gato gato1 = new Gato("A");
Gato gato2 = new Gato("A");

if (gato1 == gato2) {
    System.out.println("gato1 y gato2 apuntan al mismo objeto");
}
else { // este caso imprimiría esto:
    System.out.println("gato1 y gato2 apuntan a objetos distintos aunque sean iguales");
}
```

```
Gato gato1 = new Gato("A");
Gato gato2 = gato1;

if (gato1 == gato2) { // este caso imprimiría esto:
    System.out.println("gato1 y gato2 apuntan al mismo objeto");
}
else {
    System.out.println("gato1 y gato2 apuntan a objetos distintos aunque sean iguales");
}
```

El método `equals()` de la clase `Object` hace lo mismo que `"=="`. Es decir, devuelve `true` si los dos objetos son el mismo objeto.

En muchas ocasiones queremos sobrescribir el método `equals()` de nuestras clases para que se comporte de otra forma. Por ejemplo: dos artículos son iguales si tienen el mismo código de barras, dos personas son iguales si tienen el mismo dni, dos coches son iguales si tienen la misma matrícula, dos películas son iguales si tienen todos sus atributos iguales, etc.

Un ejemplo sencillo para sobrescribir el método `equals()` en el caso de dos alumnos donde hemos decidido que sean iguales si tienen el mismo DNI sería este: (comparamos mi dni con el dni del objeto con el que me estoy comparando)

```
@Override
public boolean equals(Object obj) {
    Alumno other = (Alumno) obj;
    return dni.equals(other.dni);
}
```

Lo correcto en un método `equals()` es que evaluemos también si el objeto con el que me estoy comparando no es `null` y que sea de la misma clase que yo. Todas estas comprobaciones se pueden implementar de forma muy rápida con ayuda de Eclipse. Menú `Source > Generate hashCode() y equals()....`

Esto haría lo mismo que el ejemplo anterior, pero comprobando todo ese tipo de cosas:

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Alumno other = (Alumno) obj;
    return Objects.equals(dni, other.dni);
}
```

¿Qué es una interfaz?

Una interfaz es un lugar donde se definen un conjunto de métodos (sólo se declaran, no se implementan) que tendrán que cumplir o implementar las clases que quieran decir que tienen esa interfaz.

Un ejemplo de la vida real: tenemos la interfaz USB. En ella se describen una serie de requisitos que tienen que tener todos los dispositivos que quieran decir: “yo cumplo con la interfaz USB”. Pero la interfaz no hace ni implementa nada. Son los dispositivos los que tendrán que fabricar e implementar lo necesario para cumplir los requisitos que la interfaz exige.

Si nos llevamos esto al mundo de la programación y la orientación a objetos, tendríamos algo similar. Veamos como se implementa y se usa:

1. En primer lugar, la interfaz sería un fichero más, independiente. Se crea desde Eclipse con **New > Interface...** Deben nombrarse igual que las clases, siempre en mayúscula.
2. La declaración de la cabecera es igual a las clases, sólo cambia la palabra **class** por **interface**. En este ejemplo hemos creado la interfaz “SerMortal”:

```
public interface SerMortal {  
  
}
```

3. Ahora podemos escribir los métodos que la interfaz va a exigir. Lo hacemos sin incorporar las llaves del cuerpo del método. En su lugar, ponemos punto y coma. Por ejemplo:

```
public interface SerMortal {  
  
    public String nacer();  
  
    public void morir(String lugar);  
  
}
```

Aquí hemos definido que la interfaz va a exigir que existan dos métodos:

- nacer → tiene que devolver un **String**
 - morir → recibe un **String**
4. Si tenemos una clase, por ejemplo, Gato, que queremos que cumpla la interfaz SerMortal, tendremos que:
 - a. Añadir arriba la expresión “implements” y el nombre de la interfaz:

```
public class Gato implements SerMortal {
```

Nos dará error al principio porque nos obliga a hacer el siguiente paso.

- b. Tendremos que implementar los métodos que nos obliga la interfaz. Eclipse nos da la opción de añadirlos automáticamente:

```
public class Gato implements SerMortal {  
  
    @Override  
    public String nacer() {  
        return null;  
    }  
  
    @Override  
    public void morir(String lugar) {  
    }  
}
```

Los métodos que son implementado por exigencia de una interfaz, también llevan la anotación `@Override`

A tener en cuenta:

- Recuerda que una interfaz nunca tiene nada de código. Sólo la declaración de métodos. También podemos incluir en la interfaz CONSTANTES. Nada más.
- Una clase puede implementar tantas interfaces como se quiera a la vez. Por ejemplo, una impresora podría a la vez implementar la interfaz USB y la interfaz Puerto Serie. Basta con cumplir los requisitos de cada una. Se escribiría así:

```
public class Gato implements SerMortal, Mamifero {
```

La clase Gato implementa la interfaz SerMortal y la interfaz Mamifero

- Las interfaces son un tipo de dato más, igual que las clases. Por tanto, puedo crear variables del tipo de una interfaz, o un array del tipo de una interfaz. Ejemplo:

```
SerMortal animalMortal = new Gato("A");  
Gato gato = new Gato("B");  
  
SerMortal[] animalesMortales = new SerMortal[3];  
animalesMortales[0] = animalMortal;  
animalesMortales[1] = gato;  
animalesMortales[2] = new Gato("C");
```

- Cuando trabajamos con interfaces, a veces tenemos que hacer uso de CASTING. En el ejemplo anterior, si yo quiero guardar el gato "C" en una variable de tipo Gato:

```
Gato gatoC = (Gato) animalesMortales[2];
```

Para JAVA, lo que hay dentro del array, son objetos que cumplen la interfaz "SerMortal", pero podrían ser objetos de tipos diferentes (Gato, Perro, etc.). Cualquier clase que implemente la interfaz. Si yo estoy seguro de que el objeto que está en la posición 2 del array es un Gato, entonces sí puedo hacer el casting anterior.

¿Qué significa el modificador static?

Vemos que a veces hemos creado métodos que llevan la palabra **static** delante y otros que no. Esa palabra significa “estático” en contraposición a “dinámico”. Ahora que sabemos qué es una clase y qué es un objeto, es fácil entender qué es algo estático: es aquello que no requiere que exista un objeto para ser utilizado.

En otras palabras, si yo tengo un método estático, puedo invocarlo escribiendo el nombre de la clase directamente. Pero si no es estático, para invocarlo tendría antes que crear un objeto de esa clase, e invocarlo a través del objeto. Veamos un ejemplo:

Tengo esta clase con dos métodos, uno estático y otro no:

```
public class Persona {  
  
    public void metodoNoEstatico() {  
        System.out.println("Esto es un método NO estático");  
    }  
  
    public static void metodoEstatico() {  
        System.out.println("Esto es un método estático");  
    }  
  
}
```

Para invocar a estos métodos de cualquier otro lugar, tendría que hacer esto:

```
// Puedo llamar a un método estático solo poniendo el nombre de la clase  
Persona.metodoEstatico();  
  
// No puedo hacer lo mismo con el NO estático, porque es un método de los objetos de la clase  
Persona.metodoNoEstatico(); // Aparece un error, no se puede hacer  
  
// Tendría que crearme un objeto  
Persona persona = new Persona();  
persona.metodoNoEstatico();  
  
// También puedo invocar a los métodos estáticos a través del objeto, aunque no es habitual  
persona.metodoEstatico(); // Aparece un warning porque es algo raro..
```

Al igual que los métodos, los atributos también pueden ser estáticos. Y significaría más o menos lo mismo: Los atributos NO estáticos sólo son accesibles a través de un objeto de la clase. Los atributos estáticos pueden usarse sin crear ningún objeto.

```
public class Persona {  
  
    public static Integer esperanzaDeVida;  
    public Integer edad;  
  
}
```

IMPORTANTE: Un atributo estático es algo de la clase, no de los objetos. Eso significa que todos los objetos de la clase tendrán siempre el mismo valor en sus atributos estáticos. Si un objeto cambia el valor, se cambia para todos.

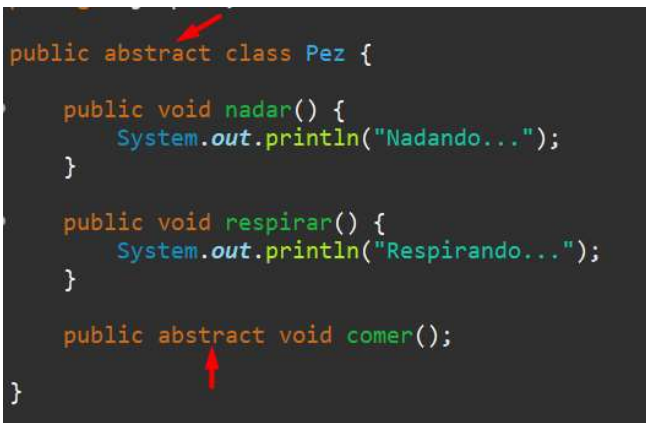
¿Qué es una clase ABSTRACTA?

A veces queremos crear una clase con determinada funcionalidad. Pero queremos dejar un método definido, pero no implementado, con el objetivo de que las clases que hereden de ella lo implementen.

Por ejemplo, podemos tener la clase `Pez` con los métodos `nadar()` y `respirar()` implementados, porque todos los peces nadan y respiran del mismo modo. Pero queremos que las clases hijas implementen el método `comer()`, porque cada tipo de `Pez` come de modo diferente. Para ello, podemos declarar el método `comer()` en la clase `Pez` de forma abstracta.

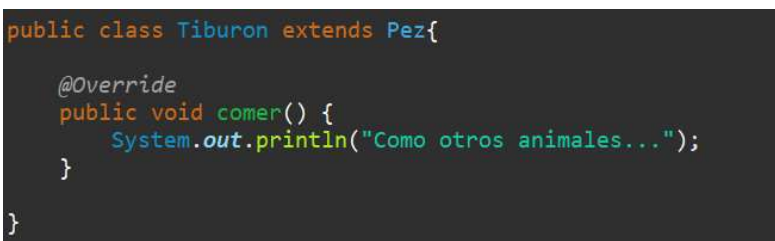
Declarar un método de forma abstracta significa que sólo ponemos su definición, pero no su cuerpo con la implementación. Igual que si fuera una interfaz. Si lo hacemos así, nos obligará a indicar el modificador `abstract` en el método y en la clase.

Veamos ejemplo:



```
public abstract class Pez {  
    public void nadar() {  
        System.out.println("Nadando...");  
    }  
    public void respirar() {  
        System.out.println("Respirando...");  
    }  
    public abstract void comer();  
}
```

Si ahora nos creamos una clase que herede de `Pez`, será obligatorio sobrescribir el método `comer()` para indicar cómo lo vamos a implementar:



```
public class Tiburon extends Pez {  
    @Override  
    public void comer() {  
        System.out.println("Como otros animales...");  
    }  
}
```

IMPORTANTE: No se puede crear un objeto de una clase Abstracta, salvo que al crearlo implementemos sus métodos abstractos.