

Métodos, funciones y procedimientos en JAVA



Los métodos, funciones y procedimientos son un conjunto de líneas de código separadas en un bloque que cumplen una tarea en específico.

¿Por qué son tan importantes?

La segmentación de código nos permite generar bloques reutilizables y automatizar ciertas tareas que requerimos con frecuencia. Por ejemplo: Nuestro proyecto podría trabajar frecuentemente con archivos .txt a lo largo de la ejecución del programa y en distintos apartados donde se harán tareas muy distintas con el archivo en cuestión, sin embargo al trabajar con archivos se realizarán repetidamente tareas para abrir, modificar, cerrar, eliminar o crear archivos. Sin la existencia de los métodos, funciones o procedimientos, las instrucciones para realizar estas tareas se repetirían a lo largo de todo nuestro proyecto.

Al generar un método que se encargue exclusivamente de abrir un archivo de texto, podremos llamarlo desde cualquier punto de nuestro programa y utilizarlo sin necesidad de repetir las instrucciones que contiene el bloque.

¿En que nos benefician los métodos, funciones y procedimientos en JAVA?

 **Atomicidad:** reducen la cantidad de líneas de código que tendrá el proyecto finalizado.

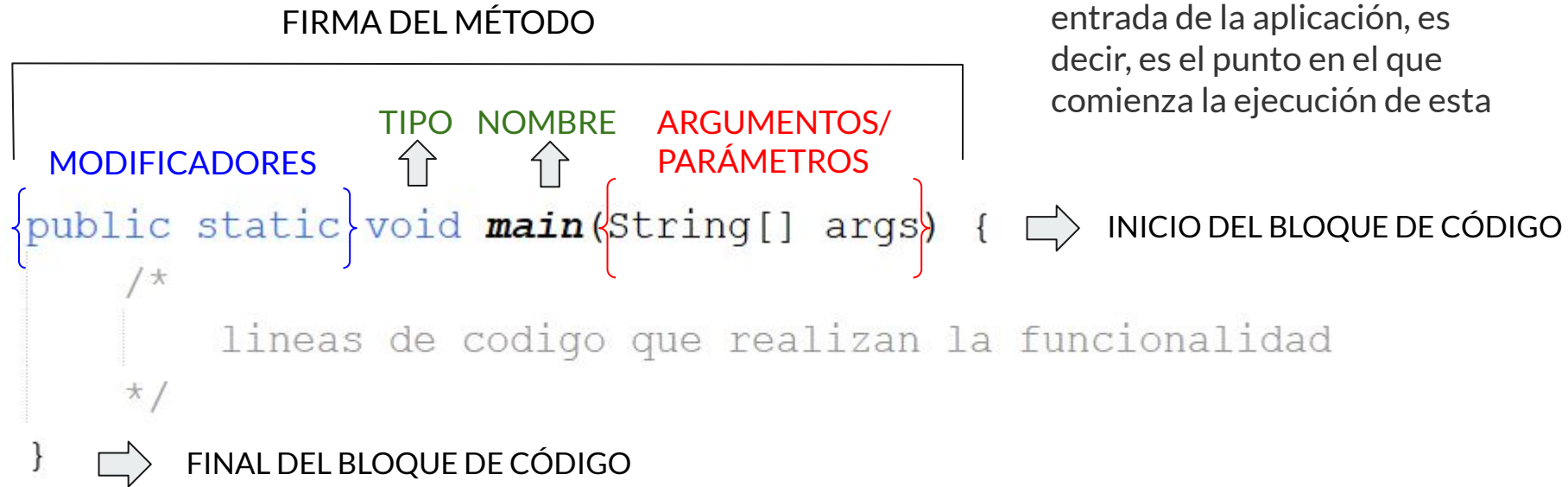
Reutilizabilidad: hoy podríamos definir un método para realizar una operación matemática sencilla para una tarea en específica de nuestro proyecto. A medida que el proyecto crezca tal vez el cliente necesitará que la misma operación se realice en otra sección del programa. Al tener el método ya definido bastará con añadir una llamada al método en la sección correspondiente para agregar la nueva funcionalidad, ahorrando tiempo de desarrollo.

Mantenimiento: nuestra aplicación podría ahora dejar de utilizar .txt para pasar a utilizar .pdf, para adaptarla a los nuevos requerimientos bastará con modificar los métodos de abrir, cerrar, crear, modificar y eliminar archivos, sin necesidad de modificar el resto del código en todo el proyecto.

Escalabilidad: al programar en bloques, se nos facilitara aumentar el alcance de nuestro proyecto. Podríamos tener dos métodos de sumar y restar en nuestra aplicación, si el cliente decide que ahora también multiplicará y dividirá, bastará con definir dos nuevos métodos que lleven a cabo dichas operaciones. Esto también permitirá que varios programadores trabajen al mismo tiempo en las nuevas funcionalidades, asignando la creación de un solo método a cada uno, reduciendo el tiempo de programación a la mitad.

Legibilidad: nombrar y delimitar la responsabilidad que tiene el método hará que para nuevos integrantes del proyecto sea más entendible que es lo que hace la aplicación en general, lo que también favorece el mantenimiento. Si el cliente decide que las sumas ahora se harán con números decimales y no con números enteros, el nuevo programador irá directamente a modificar el método **SUMAR()** y no tendrá que leer todo el código para encontrar la porción donde se suman los valores. O podrá renombrar el método **SUMAR()** a **SUMAR_ENTEROS()** y luego definir uno nuevo llamado **SUMAR_DECIMALES()**, donde no hará falta revisar el código para comprender su funcionalidad, ya que el nombre del método es explícito.

Firma de un método y sus partes



El método *main()* es el punto de entrada de la aplicación, es decir, es el punto en el que comienza la ejecución de esta

Parámetros de un método

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

```
1 imprimir(); ← Llama al método imprimir sin parámetros
```

```
2 imprimir(mensaje: "Usando parametros"); ← Llama al método imprimir con parámetros con una literal como parámetro
```

```
String variable = "Valor por parametro desde variable";
```

```
3 imprimir(mensaje: variable); ← Llama al método imprimir con parámetros con una variable como parámetro
```

```
private static void imprimir() { Método imprimir sin parámetros 1 Imprime "Mi mensaje desde un método"
```

```
System.out.println(x: "Mi mensaje desde un metodo");
```

```
private static void imprimir(String mensaje) {
```

```
System.out.println(x: mensaje);
```

2 Imprime la literal pasada por parámetro "Usando parámetros"

3 Imprime la variable pasada por parámetro "Valor por parámetro desde variable"

Métodos con retorno

En los métodos que deban devolver un valor como retorno, usaremos la palabra reservada `return` y cambiamos `void` por el tipo de dato que deseamos devolver

```
private static int sumar(int a, int b) {  
    return a + b;  
}
```

Retorna el valor entero
resultante de la suma

```
private static boolean sonIguales(int a, int b) {  
    return a == b;  
}
```

Retorna el valor booleano
resultante de la operación
lógica (true o false)

```
private static void imprimir() {  
    System.out.println(x: "Mi mensaje desde un metodo");  
}
```

Los métodos de tipo
`void` no tienen retorno

Consumiendo el valor retornado de un método

```
public static void main(String[] args) {  
1  int resultado = sumar(a: 2, b: 3);  
    System.out.println("Suma: " + resultado);  
  
3  System.out.println("Suma: " + sumar(a: 3, b: 3));  
  
4  int a = 5; int b = 7;  
    System.out.println("Suma: " + sumar(a, b));  
  
2  int resultadoSuma = sumar(a, b);  
    System.out.println("SUMA: " + resultadoSuma);  
}
```

```
private static int sumar(int a, int b) {  
    return a + b;  
}
```


Caso 1 y 2:

El retorno de la operación aritmética del método `sumar` puede ser almacenado en una variable (`resultado/ resultadoSuma`)

Caso 3 y 4:

El retorno de la operación aritmética del método `sumar` puede ser consumido en el momento sin ser almacenado, como parámetro de otra función o método (`System.out.println()`)

Observen que los parámetros recibidos por el método pueden ser literales, variables o retornos de otros métodos, siempre y cuando coincidan los tipos esperados en la firma.




Observen que los valores retornados por los métodos también pueden ser consumidos en expresiones lógicas de los condicionales de las distintas estructuras de control que existen en JAVA

Ya sean IF, IF ELSE, FOR, WHILE, DO WHILE, SWITCH u el operador TERNARIO

```
public static void main(String[] args) {  
    int a = 5; int b = 7;  
    if (sonIguales(a, b)) {  
        System.out.println(x: "Son iguales");  
    } else {  
        System.out.println(x: "Son distintos");  
    }  
  
    if (sumar(a, b) > 0) {  
        System.out.println(x: "El resultado es positivo");  
    } else {  
        System.out.println(x: "El resultado es negativo");  
    }  
}  
  
private static boolean sonIguales(int a, int b) {  
    return a == b;  
}  
  
private static int sumar(int a, int b) {  
    return a + b;  
}
```

Sobrecarga de un método (Utilizaremos este concepto en Clases y Objetos)



```
private static void imprimir() {  
    System.out.println(x: "Mi mensaje desde un metodo");  
}  
  
private static void imprimir(String mensaje) {  
    System.out.println(x: mensaje);  
}  
  
private static void imprimir(String mensaje, String msg) {  
    System.out.println("Primer mensaje: " + mensaje);  
    System.out.println("Segundo mensaje: " + msg);  
}
```

En JAVA los nombres de los métodos al igual que las variables no pueden repetirse a lo largo de la clase, ya que el compilador es incapaz de distinguir a qué método se refiere el programador. Para ello existe la sobrecarga de métodos, que consiste en declarar métodos con el mismo nombre pero cambian la cantidad de parámetros que recibe cada uno. Puede mantener la cantidad pero deberá cambiar el tipo de parámetro.

Consideraciones a tener en cuenta sobre PARÁMETROS o ARGUMENTOS



- Una función, un método o un procedimiento pueden tener una cantidad cualquier de parámetros, es decir pueden tener cero, uno, tres, diez, cien o más parámetros. Aunque habitualmente no suelen tener más de 4 o 5.
- Si una función tiene más de un parámetro cada uno de ellos debe ir separado por una coma.
- Los argumentos de un método también tienen un tipo y un nombre que los identifica. El tipo del argumento puede ser cualquiera y no tiene relación con el tipo del método.
- Al recibir un argumento nada nos obliga a hacer uso de éste al interior del método, sin embargo para qué recibirlo si no lo vamos a usar.
- En Java los parámetros que podemos recibir pueden ser por valor por referencia, esto implica que si modificamos los valores recibidos al interior del método, estos pueden mantener sus cambios o no después de ejecutada el método (esto lo explicaré con más detalle en las próximas clases).

Consideraciones a tener en cuenta sobre RETURN



- El tipo del valor que se retorna en una función debe coincidir con el del tipo declarado a la función, es decir si se declara int, el valor retornado debe ser un número entero.
- Cualquier instrucción que se encuentre después de la ejecución de return NO será ejecutada. Es común encontrar funciones con múltiples sentencias return al interior de condicionales, pero una vez que el código ejecuta una sentencia return lo que haya de allí hacia abajo no se ejecutará.
- En el caso de los procedimientos (void) podemos usar la sentencia `return` pero sin ningún tipo de valor, sólo la usamos como una manera de terminar la ejecución del procedimiento.

Estos últimos dos casos los veremos más adelante

Consideraciones a tener en cuenta sobre la invocación o llamada a métodos



- No importa si se trata de un método o de una función o de un procedimiento, sólo debes ocuparte de enviar los parámetros de la forma correcta para invocarlos.
- El nombre debe coincidir exactamente al momento de invocar, pues es la única forma de identificarlo.
- El orden de los parámetros y el tipo debe coincidir. Hay que ser cuidadosos al momento de enviar los parámetros, debemos hacerlo en el mismo orden en el que fueron declarados y deben ser del mismo tipo (entero, cadena, etc).
- Cada parámetro enviado también va separado por comas.
- Si una función no recibe parámetros, simplemente no ponemos nada al interior de los paréntesis, pero SIEMPRE debemos poner los paréntesis.
- Invocar una función sigue siendo una sentencia común y corriente en Java, así que ésta debe finalizar con ';' como siempre.
- El valor retornado por un método o función puede ser asignado a una variable del mismo tipo, pero no podemos hacer esto con un procedimiento, pues no retornan valor alguno.
- Una función puede llamar a otra dentro de sí misma o incluso puede enviar su retorno como parámetro a otra (mira el siguiente ejemplo).

Métodos anidados

1. Se ejecuta el método main()



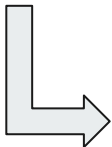
2. Llama al método leerEntero()



3. El método leerEntero()
llama al método imprimir()



4. El método imprimir() imprime
y regresa a leerEntero()



5. El método leerEntero()
lee el entero y retorna a main()



6. El método main() imprime el número
leído desde consola por el método
leerEntero()

```
1 public static void main(String[] args) {  
2     int enteroLeido = leerEntero();  
6     System.out.println("El numero ingresado es: " + enteroLeido);  
}
```

```
private static int leerEntero() {  
    Scanner consola = new Scanner(source: System.in);  
3     imprimir(mensaje: "Ingresa un numero:");  
    return consola.nextInt(); 5  
}
```

```
private static void imprimir(String mensaje) {  
4     System.out.println(x: mensaje);  
}
```

Diferencias entre métodos, funciones y procedimientos



Generalmente en JAVA todos son métodos o por lo menos así los nombra la gente comúnmente. Pero podemos distinguir a los tres dependiendo del retorno o si están asociados a la clase u objeto.

Los procedimientos son los “métodos” de tipo **VOID**, es decir aquellos que realizan un proceso y no retornan ninguna salida.

Las funciones siempre serán “métodos” que tendrán el modificador **static** (ya aprenderemos más adelante que significa esta palabra reservada), están asociados estrechamente a la clase.

Los métodos en cambio siempre estarán asociados al objeto y será necesario **instanciar** el objeto primero para utilizar sus métodos. También veremos este tema más adelante.

Sintácticamente y semánticamente no hay mucha diferencia en cómo se escribe uno y otro, y aunque no este 100% bien llamar a los tres tipos “métodos” de manera generalizada, es mas una cuestion teorica de **PROGRAMACIÓN ORIENTADA A OBJETOS**