Material descargable

Teoría **JAVA VII**

¡Hola nuevamente! 👋

En esta ocasión, nos adentraremos en un fascinante tema de programación: **los métodos en Java**.

Comenzaremos por comprender la **declaración de un método** y cómo podemos utilizarlo para encapsular una serie de instrucciones y ejecutar tareas específicas. Además, exploraremos la **sobrecarga de métodos**, una poderosa técnica que nos permitirá definir múltiples versiones de un método con diferentes parámetros.

También abordaremos el concepto de **variables globales** y cómo pueden ser utilizadas en diferentes métodos de una clase. Analizaremos las diferencias entre los **parámetros pasados por valor y por referencia**, lo que nos ayudará a comprender cómo se manejan los datos en las llamadas a métodos.

Dominar los métodos en Java es esencial para construir programas robustos y eficientes. A medida que avancemos en este tema, desarrollaremos habilidades prácticas que nos permitirán abordar proyectos más complejos y mejorar nuestras capacidades como programadores.

¡Prepárate para expandir tu dominio de Java y llevar tus habilidades de programación al siguiente nivel! 🚀

Métodos

Los **métodos** son **bloques de código que representan programas más pequeños**.

Hasta el momento, hemos estado construyendo nuestros programas en un único bloque de código en el método *"main"*. Sin embargo, es posible que hayas notado que, a pesar de utilizar iteradores, todavía hay partes del código que se repiten.

Los métodos nos permiten resolver este problema y mejorar la legibilidad de nuestro código. **Al utilizar métodos, podemos encapsular secciones repetitivas**

de código en bloques separados y reutilizables. Esto simplifica nuestra programación y facilita la comprensión del código.

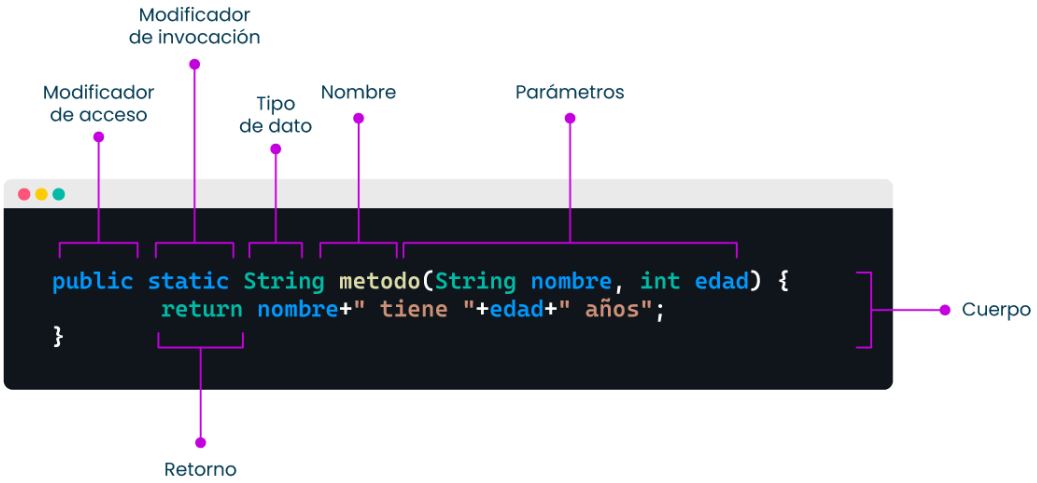
¡Prepárate para explorar cómo los métodos en Java pueden optimizar y hacer que nuestro código sea más legible!

Declaración de un método

En las siguientes imágenes se muestra el formato de *declaración de un método*:

Método sin retorno

Método con retorno



En el primer caso, el método no tiene un valor de retorno, ya que se declara con el tipo de dato "void". Mientras que en el segundo caso, el método devuelve una cadena de texto y se utiliza el tipo de dato "String".

Aquí te proporcionamos una explicación de los elementos de la declaración del método:

- **Modificador de acceso:** En este caso, el modificador es *"public"*. Sin embargo, se discutirá en detalle el funcionamiento de los diferentes modificadores de acceso cuando nuestros programas tengan más de un archivo y/o paquete.
- **Modificador de invocación:** En este caso, el modificador es "static" para permitir la invocación del método desde el método *"main"*. Proporcionaremos una explicación más detallada sobre esto cuando exploremos los modificadores de acceso en profundidad.
- **Tipo de dato:** Este es el tipo de dato que el método devuelve.
    - En el primer caso, al ser *"void"*, el método no devuelve ningún valor.
    - En el segundo caso, al ser *"String"*, el método retorna una cadena de texto.
- **Nombre**: Es el nombre del método, siguiendo la convención *camelCase* al igual que las variables.
- **Parámetros**: Se colocan entre paréntesis y representan las variables que el método espera recibir cuando se invoca. Se declara el tipo de dato seguido del nombre de la variable. Si el método no requiere parámetros, se utilizan paréntesis vacíos. Puede haber múltiples parámetros separados por comas.
- **Cuerpo:** Aquí se encuentran las líneas de código que se ejecutarán cuando se invoque el método. El cuerpo está delimitado por las llaves.
- **Retorno:** Es el valor que el método devuelve y debe coincidir con el tipo de dato declarado en el método. Se utiliza la palabra clave *"return"* seguida del valor a retornar. En el caso de un método declarado con "void", no se utiliza la palabra clave "return" ya que no se espera ningún valor de retorno.

Uso de un método

En el código a continuación, estamos utilizando solo el método "main":

```java
public class Metodos {
    public static void main(String[] args) {
        int[] array = {1,5,2,3};
        System.out.println();
        for (int i = 0; i < array.length; i++) {
            System.out.print("["+array[i]+"]");
        }
        System.out.println();
    }
}
```

Ahora, veamos cómo declarar un nuevo método para imprimir el arreglo:

```java
public class Metodos {
    public static void main(String[] args) {
        int[] array = {1,5,2,3};
        imprimirArray(array);
    }

    public static void imprimirArray(int[] array) {
        System.out.println();
        for (int i = 0; i < array.length; i++) {
            System.out.print("["+array[i]+"]");
        }
        System.out.println();
    }
}
```

Como puedes ver, *debemos utilizar la misma sintaxis que con el método "main".* Si observamos el primer método "main" al principio del programa, es posible que no entiendas claramente qué hace ese programa. Sin embargo, si observamos el segundo programa donde utilizamos un método, es más fácil comprender que esa línea de código imprimirá algo.

Esa es la idea de crear métodos: *generar pequeños programas que nos permitan reutilizar código y hacer que nuestros algoritmos sean más legibles y comprensibles*.

Veamos el siguiente ejemplo de código con el método "main":

```
public class Metodos {
    public static void main(String[] args) {
        int[] array = new int[10];
        for (int i = 0; i < array.length; i++) {
            array[i] = (int) (Math.random()*11+1);
        }
        System.out.println();
        for (int i = 0; i < array.length; i++) {
            System.out.print("["+array[i]+"]");
        }
        System.out.println();
        Arrays.sort(array);
        System.out.println();
        for (int i = 0; i < array.length; i++) {
            System.out.print("["+array[i]+"]");
        }
    }
}
```

Ahora vamos a compararlo con el siguiente código, que utiliza el mismo código pero lo organiza en métodos separados:

```
public class Metodos {
    public static void main(String[] args) {
        int[] array = crearArrayAleatorio();
        imprimirArray(array);
        ordernarAscendentemente(array);
        imprimirArray(array);
    }

    public static int[] crearArrayAleatorio() {
        int[] array = new int[10];
        for (int i = 0; i < array.length; i++) {
            array[i] = (int) (Math.random()*11+1);
        }
        return array;
    }

    public static void imprimirArray(int[] array) {
```

```
        System.out.println();
        for (int i = 0; i < array.length; i++) {
            System.out.print("["+array[i]+"]");
        }
        System.out.println();
    }

    public static void ordernarAscendentemente(int[] array) {
        Arrays.sort(array);
    }
}
```

Seguramente notaste cómo en el segundo ejemplo resultó mucho más fácil leer el código y comprender qué hace el programa.

Sobrecarga de métodos

La **sobrecarga de métodos** nos **permite reutilizar un nombre de método con diferentes conjuntos de parámetros**. Esto nos brinda flexibilidad al crear métodos con funcionalidades similares pero con diferentes formas de uso.

Por ejemplo, consideremos el método *"crearArrayAleatorio()"*. Podemos sobrecargar este método para permitir que se le pase por parámetro el tamaño deseado para el nuevo array:

```
public class Metodos {
    public static void main(String[] args) {
        int[] array = crearArrayAleatorio();
        imprimirArray(array);
        ordernarAscendentemente(array);
        imprimirArray(array);
    }

    public static int[] crearArrayAleatorio() {
        int[] array = new int[10];
        for (int i = 0; i < array.length; i++) {
            array[i] = (int) (Math.random()*11+1);
        }
```

```
        return array;
    }

    public static void imprimirArray(int[] array) {
        System.out.println();
        for (int i = 0; i < array.length; i++) {
            System.out.print("["+array[i]+"]");
        }
        System.out.println();
    }

    public static void ordernarAscendentemente(int[] array) {
        Arrays.sort(array);
    }
}
```

💡*La sobrecarga de métodos nos permite tener métodos con nombres intuitivos y claros, adaptados a diferentes situaciones y necesidades en nuestro programa.*

Variables globales

Ya hemos aprendido sobre el ámbito de una variable cuando exploramos los bloques de código en las estructuras de control.

💡*Recuerda que un bloque de código es aquel que está encerrado entre llaves.*

Si declaramos una variable dentro de una *estructura de control*, esa variable es accesible únicamente dentro de su bloque y no puede ser utilizada fuera del mismo. Por lo tanto, **si deseamos utilizar una variable en varios bloques, debemos declararla en un bloque de código que englobe a los demás**. En nuestro caso, esto sería el método "main".

Sin embargo, al utilizar más de un método, debemos entender que esta lógica de ámbito de las variables se aplica de la misma manera. Es decir, **las variables declaradas en un método no son accesibles desde otros métodos**.

Existen dos formas comunes de compartir el contenido de las variables entre métodos:

1. Pasar las variables por parámetro: De esta manera, el valor de una variable declarada en un método se copia en otra variable declarada como parámetro en el método objetivo. Por ejemplo:

```
public class VariableLocal {

    public static void main(String[] args) {
        Scanner pepe = new Scanner(System.in);
        String palabra = metodo1(pepe);
        int numero = metodo2(pepe);
    }

    public static String metodo1(Scanner sc) {
        return sc.nextLine();
    }

    public static int metodo2(Scanner scanner) {
        return scanner.nextInt();
    }
}
```

2. Declarar variables globales: Podemos declarar variables en un *bloque que englobe a los bloques de los métodos*, es decir, en el bloque de la clase donde se encuentran los métodos. Por ejemplo:

```
public class VariableGlobal {

    public static Scanner pepe = new Scanner(System.in);

    public static void main(String[] args) {
        String palabra = metodo1();
        int numero = metodo2();
    }

    public static String metodo1() {
        return pepe.nextLine();
    }

    public static int metodo2() {
```

```
        return pepe.nextInt();
    }
}
```

Estas variables se conocen como *variables globales de la clase* y pueden ser utilizadas en todos los métodos sin necesidad de declararlas como parámetros, y a las de los métodos se conocen como ***variables locales del método***.

Es importante tener en cuenta que, al declarar una variable global, **es posible crear una variable local con el mismo nombre en un método particular**.

En ese caso, la variable local tendrá prioridad o precedencia sobre la variable global dentro del ámbito del método en el que se declara. Esto significa que al hacer referencia al nombre de la variable dentro del método, se hará referencia a la variable local en lugar de la variable global. *La variable local "toma prioridad" sobre la variable global con el mismo nombre*.

Es fundamental tener en cuenta esta prioridad al trabajar con variables locales y globales en el mismo método. En futuras clases, cuando comencemos a trabajar con el contexto de instancia dentro de una clase, exploraremos cómo manejar variables globales y locales con el mismo nombre dentro de un mismo método.

Parámetros por valor o por referencia

Cuando hablamos de p**arámetros por valor o por referencia**, nos referimos a **lo que sucede cuando pasamos nuestras variables como argumentos a otros métodos**.

En Java, es importante entender que este comportamiento depende del tipo de dato:

- **Se pasan por valor:** Los datos de tipo primitivo, los Wrappers y la clase String se pasan por valor. Los tipos de datos primitivos, como int, boolean, char, etc., junto con sus respectivos Wrappers, como Integer, Boolean, Character, se comportan de esta manera. Esto significa que se crea una copia de la variable y se pasa esa copia al método. Cualquier cambio realizado en la copia no afectará a la variable original.

- **Se pasan por referencia:** El resto de los tipos de datos, como objetos y tipos de datos no primitivos (arrays, clases personalizadas, etc.), se pasan por referencia. En este caso, no se crea una copia de la variable, sino que se pasa una referencia a la ubicación de memoria donde se encuentra el objeto. Cualquier modificación realizada en el objeto dentro del método afectará a la variable original.

La distinción entre parámetros por valor y por referencia es importante porque afecta cómo se manipulan y modifican los datos dentro de los métodos. Comprender esto nos permite tener un mayor control sobre el comportamiento de nuestras variables al pasarlas como argumentos a otros métodos.

Podemos entenderlo mejor con la siguiente analogía:

Imagina que tienes un equipo de trabajo y una lista de tareas pendientes. La lista de tareas representa una variable, y cada miembro del equipo representa un método diferente en el programa.

Cuando la lista de tareas se pasa por valor, es como si se entregara un papel a cada miembro con una copia de la lista. Cada miembro puede tomar su copia de la lista y realizar modificaciones en ella sin afectar la lista original. Cada miembro del equipo terminará con una versión modificada de la lista de tareas, pero estas modificaciones no se reflejarán en la lista original.

En cambio, cuando la lista de tareas se pasa por referencia, se les proporciona a los miembros del equipo un papel con la "ubicación" de la lista de tareas original. Cada miembro del equipo sabe dónde encontrar la lista y puede realizar modificaciones directamente en ella.

En el contexto de los métodos, el papel que tiene cada miembro representa la *variable local* recibida como *parámetro*.

- Cuando el *tipo de dato corresponde al paso por referencia*, cada método accede al objeto original y realiza modificaciones directamente sobre él.
- En cambio, cuando el *tipo de dato corresponde al paso por valor*, cada método accede a una copia del valor original y realiza modificaciones en la copia sin afectar el contenido de la variable original que se pasó por parámetro.

Para comprender esto y ver cómo funciona en el código, te invitamos a ver el siguiente video:

🎥**[Java] Parámetros por valor y por referencia**