Material descargable

egg

Teoría JAVA I

¿Qué son los Métodos?

Los métodos son segmentos de código que representan funcionalidades más pequeñas y específicas dentro de un programa. Hasta este punto, hemos desarrollado nuestros programas en un solo bloque de código dentro del método "main". Sin embargo, al utilizar iteraciones, es posible que aún encontremos partes del código que se repiten.

Los métodos nos permiten abordar esta repetición y mejorar la legibilidad de nuestro código. Al utilizar métodos, podemos encapsular secciones de código repetitivas en bloques separados y reutilizables. Esto simplifica nuestro código y facilita su comprensión.

Ahora, exploremos cómo los métodos en Java pueden optimizar nuestro código y hacerlo más legible.

La importancia de los métodos

Reutilización de Código: Los métodos nos permiten reutilizar código sin necesidad de repetirlo, lo que hace que nuestros programas sean más cortos, más legibles y más fáciles de mantener.

Abstracción: Al encapsular la complejidad en métodos, podemos trabajar a un nivel más alto de abstracción, preocupándonos menos de los detalles de implementación, es decir, podemos dividir nuestro proceso en pasos con un nombre, y cada uno de esos pasos podemos traducirlos en métodos.

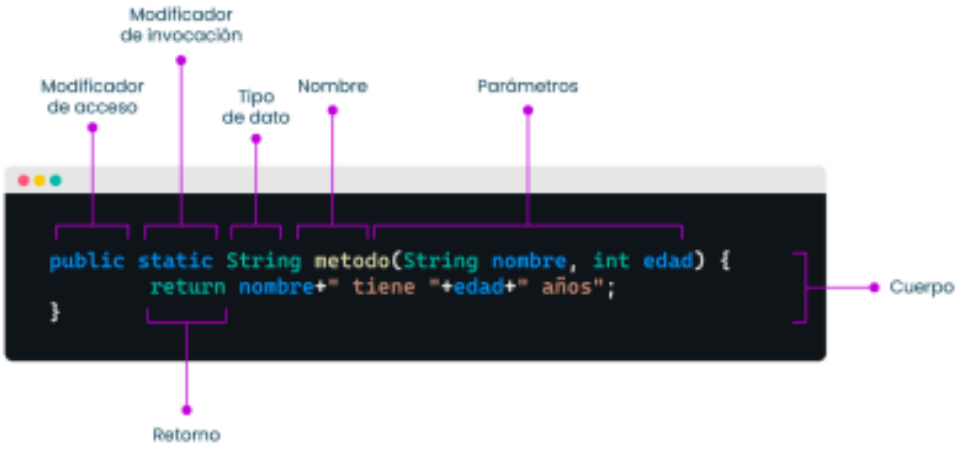
Organización: Los métodos ayudan a organizar el código en unidades lógicas, lo que facilita la depuración y el testing del código.

Declaración de un Método

Aquí presentamos 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". En el segundo caso, el método devuelve una cadena de texto y utiliza el tipo de dato "String".

Vamos a explicar los elementos de la declaración del método:

- **Modificador de Acceso:** En este caso, el modificador es "public". Sin embargo, discutiremos los diferentes modificadores de acceso en profundidad cuando nuestros programas sean más complejos.
- **Modificador de Invocación**: En este caso, el modificador es "static" para permitir la invocación del método desde el método "main". Discutiremos esto con más detalle más adelante.
- **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.
- **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".

Uso de un método

Veamos cómo utilizar un nuevo método para imprimir un arreglo:

```
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 seguir la misma sintaxis que con el método "main". Al utilizar un método, se hace más claro el propósito de la línea de código, mejorando la comprensión del programa.

Ejemplo de Organización de Código utilizando Métodos

Comparando dos versiones del mismo código, es evidente cómo la organización en métodos facilita la comprensión:

```java
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] + "]");
        }
    }
}
```

```java
public class Metodos {
    public static void main(String[] args) {
        int[] array = crearArrayAleatorio();
        imprimirArray(array);
        ordernarDeFormaAscendente(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 ordernarDeFormaAscendente(int[] array) {
        Arrays.sort(array);
    }
}
```

El segundo ejemplo es más fácil de leer y comprender debido a la organización en métodos.

Sobrecarga de Métodos

La sobrecarga de métodos nos permite utilizar el mismo nombre de método con diferentes conjuntos de parámetros, lo que brinda flexibilidad al diseñar métodos con funcionalidades similares pero con diferentes formas de uso.

Por ejemplo, podemos sobrecargar el método "crearArrayAleatorio()" para permitir que se le pase el tamaño deseado para el nuevo array como parámetro

```
public class Metodos {
        public static void main(String[] args) {
            int[] array = crearArrayAleatorio(10);
            imprimirArray(array);
            ordernarDeFormaAscendente(array);
            imprimirArray(array);
        }

        public static int[] crearArrayAleatorio(int tamanho) {
            int[] array = new int[tamanho];
            for (int i = 0; i < array.length; i++) {
                array[i] = (int) (Math.random()*11+1);
            }
            return 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 ordernarDeFormaAscendente(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 vs Variables por Parámetros

Ya hemos aprendido sobre el ámbito de las variables al explorar los bloques de código en las estructuras de control. Si declaramos una variable dentro de un bloque, esa variable es accesible sólo dentro de ese bloque. Sin embargo, al utilizar más de un método, esta lógica de ámbito también se aplica.

Existen dos formas comunes de compartir el contenido de las variables entre métodos: **pasar las variables por parámetro** o **declararlas como variables globales de la clase**. Las variables globales pueden ser utilizadas en todos los métodos de la clase sin necesidad de pasarlas como parámetros.

Es fundamental tener en cuenta la prioridad de las variables locales sobre las globales con el mismo nombre dentro de un método.

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:

```java
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 como 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();
     }
 }
```

Parámetros en Java: Por Valor o Por Referencia

Al hablar de parámetros por valor o por referencia en Java, nos referimos a cómo se manejan nuestras variables cuando se pasan como argumentos a otros métodos. Es esencial comprender que este comportamiento depende del tipo de dato:

- **Por Valor:** Los datos de tipo primitivo, los Wrappers y la clase String se pasan por valor. Esto implica 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.
- **Por Referencia:** Los tipos de datos restantes, como objetos y tipos de datos no primitivos (arrays, clases personalizadas, etc.), se pasan por referencia.

Aquí, 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.

Entender la distinción entre parámetros por valor y por referencia es crucial, ya que afecta cómo se manipulan y modifican los datos dentro de los métodos. Esto nos brinda un mayor control sobre el comportamiento de nuestras variables al pasarlas como argumentos.

Una analogía útil es imaginar un equipo de trabajo con una lista de tareas pendientes. Cada miembro del equipo representa un método en el programa. Cuando la lista de tareas se pasa por valor, cada miembro recibe una copia de la lista y puede realizar modificaciones en ella sin afectar la lista original. En contraste, cuando la lista se pasa por referencia, cada miembro recibe una referencia a la ubicación de la lista original y puede modificarla directamente.

En el contexto de los métodos, el "papel con la lista" 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. Por otro lado, cuando el tipo de dato corresponde al paso por valor, cada método accede a una copia del valor original y realiza modificaciones en esa copia sin afectar la variable original pasada como parámetro.

💡**Para comprender mejor este concepto y ver cómo funciona en el código, te invitamos a ver el siguiente video** 🎥**Parámetros por valor y por referencia | JAVA | Egg**