«Talento Tech»

Back-End

Java

Clase 04

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase 4: Funciones y modularización

Índice

Objetivos de la clase

- Entender qué son las funciones (métodos) en Java y cómo se diferencian de otras partes del código.
- Aprender a declarar y llamar métodos, comprender los parámetros y argumentos.
- Entender el concepto de paso por valor y su impacto en el comportamiento de nuestros métodos.
- Conocer la sobrecarga de métodos y cuándo es útil.
- Aplicar buenas prácticas de modularización para mejorar la mantenibilidad y escalabilidad del código.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Introducción:

¡Les damos la bienvenida de nuevo! Hasta ahora hemos aprendido a **manejar variables, operadores, estructuras de control y a interactuar con usuarias y usuarios**. Hoy daremos un paso fundamental hacia una programación más organizada: el uso de **funciones** (métodos) y la modularización del código. Estas herramientas permitirán dividir nuestro programa en piezas más pequeñas, reusables y fáciles de mantener.

Concepto de funciones (métodos) en Java

Una función (en Java, típicamente llamadas "métodos") es un bloque de código que se ejecuta cuando se invoca por su nombre. Los métodos permiten:

- Reutilizar código: Si una misma operación se repite en varias partes del programa, un método evita duplicarla.
- Dividir el programa en partes lógicas: Cada método puede encargarse de una tarea concreta, facilitando la lectura y el mantenimiento del código.
- Encapsular la lógica: La complejidad puede ocultarse dentro de un método, presentando una interfaz simple (la invocación del método) al resto del programa.

Un método en Java se define con una firma que incluye el tipo de retorno, el nombre y una lista de parámetros (opcional). Por ejemplo:

```
public int sumar(int a, int b) {
    int resultado = a + b;
    return resultado;
}
```

Aquí, `sumar` es un método que recibe dos enteros (a y b) y devuelve su suma.

Problema: Cálculo de Descuentos Reutilizable

Contexto: Hasta ahora, TechLab calculaba el descuento directamente en el `main`, mezclando la lógica de entrada de datos, cálculo y presentación del resultado. Silvia pide una forma de aplicar el descuento a distintos productos sin reescribir la lógica.

Solución con un método:

Podemos crear un método `aplicarDescuento` que reciba el precio original y el porcentaje de descuento, y retorne el precio final.

```
public static double aplicarDescuento(double precio, double porcentaje)
{
   double descuento = precio * (porcentaje / 100.0);
   double precioFinal = precio - descuento;
   return precioFinal;
}
```

Ahora este método se puede invocar tantas veces como sea necesario, con diferentes precios y descuentos.

Declaración y llamada de Funciones

Para usar un método, primero se declara con una firma (visibilidad, si es `static` o no, tipo de retorno, nombre y parámetros). Luego, se lo invoca escribiendo su nombre seguido de paréntesis y pasando los argumentos necesarios.

- **Declaración:** Dentro de la clase, fuera de otro método.
- **Llamada:** Puede hacerse desde `main` u otros métodos. Ejemplo:

```
double precioOriginal = 1000.0;
double precioConDescuento = aplicarDescuento(precioOriginal, 15.0);
System.out.println("Precio final: " + precioConDescuento);
```

Problema: separación de lógica en múltiples métodos

Contexto: Silvia ahora pide no sólo calcular el descuento, sino también imprimir un recibo detallado. Actualmente, todo está en `main`. Matías y Sabrina deciden crear un método para calcular el descuento y otro para imprimir el recibo, manteniendo el `main` más limpio. Veamos cómo quedaría el código.

Solución:

```
public static void imprimirRecibo(String producto, double
precioOriginal, double precioFinal) {
    System.out.println("Recibo de Compra");
    System.out.println("Producto: " + producto);
    System.out.println("Precio Original: $" + precioOriginal);
    System.out.println("Precio con Descuento: $" + precioFinal);
}
```

Así, `main` puede enfocarse en la orquestación y no en los detalles.

Parámetros y argumentos

- **Parámetros:** Son las variables que se listan en la declaración del método, dentro de los paréntesis. Definen qué datos espera el método.
- **Argumentos:** Son los valores concretos que se pasan al método cuando se llama, reemplazando a los parámetros.

Ejemplo: En `aplicarDescuento(double precio, double porcentaje)`, `precio` y `porcentaje` son parámetros. Cuando llamamos `aplicarDescuento(1000.0, 15.0)`, `1000.0` y `15.0` son argumentos.

Problema: ajustar precios según parámetros entrantes

Contexto: Silvia propone que si la cantidad de productos supera cierto umbral el descuento sea mayor. Esto implica un método que reciba también la cantidad como parámetro.

Solución:

```
public static double aplicarDescuentoConCantidad(double precio, double
porcentajeBase, int cantidad) {
    if (cantidad > 50) {
        porcentajeBase += 5; // Aumentamos el descuento si compra más de
50 unidades
    }
    return aplicarDescuento(precio, porcentajeBase);
}
```

Aquí, `cantidad` es un parámetro que afecta el cálculo final.

Paso por valor y referencia

En Java, los argumentos se pasan siempre por valor, lo que significa que el método recibe una copia del valor y no puede cambiar la variable original fuera del método. Sin embargo, para tipos complejos (como objetos o arrays), la referencia a ese objeto se copia, pero no el objeto en sí. Esto significa que el método puede modificar el estado del objeto apuntado por esa referencia, aunque no puede cambiar a qué referencia apunta la variable original en el contexto que la llama.

Problema: ajuste de inventario mediante métodos

Contexto: Tenemos un array que representa el stock de varios productos. Al venderse productos, necesitamos actualizar las cantidades. Matías y Sabrina temen que al pasar el array a un método no se reflejen los cambios en el `main`. Pero recuerdan que Java pasa referencias por valor, y si modifican el contenido del array en el método, el cambio se reflejará afuera.

Solución:

```
public static void actualizarStock(int[] inventario, int
indiceProducto, int unidadesVendidas) {
    // Modificamos el contenido del array
    inventario[indiceProducto] -= unidadesVendidas;
}
```

Llamando a `actualizarStock(inventario, 2, 10)`, si `inventario[2]` era 50, ahora será 40, y este cambio se verá reflejado en el `main`.

Sobrecarga de métodos

La sobrecarga permite tener varios métodos con el mismo nombre pero distintos parámetros (en número o tipo). Esto resulta útil cuando queremos que una misma acción se aplique a diferentes datos sin cambiar el nombre del método.

Problema: cálculo de descuentos con distintas firmas de método

Contexto: Silvia solicita un método `aplicarDescuento` para clientes regulares y otro para clientes premium, que reciben un descuento adicional. En lugar de crear métodos con nombres distintos, podemos sobrecargar el mismo nombre `aplicarDescuento`.

Solución:

```
// Versión básica
public static double aplicarDescuento(double precio, double porcentaje)
{
    return precio - (precio * (porcentaje / 100.0));
}
// Versión sobrecargada para clientes premium, agregando un bonus
public static double aplicarDescuento(double precio, double porcentaje,
double bonusAdicional) {
    double precioDescontado = aplicarDescuento(precio, porcentaje);
    return aplicarDescuento(precioDescontado, bonusAdicional);
}
```

De esta manera, podemos llamar `aplicarDescuento(1000.0, 10.0)` o `aplicarDescuento(1000.0, 10.0, 5.0)` dependiendo del tipo de cliente.

Buenas prácticas de modularización

La modularización implica dividir el código en métodos lógicos, evitando métodos demasiado largos o que hagan muchas cosas a la vez. Algunas buenas prácticas:

- **Un método, una responsabilidad:** Cada método debe realizar una sola tarea concreta.
- **Nombres descriptivos:** Los nombres de los métodos deben reflejar su función.
- **Evitar métodos demasiado largos:** Si un método supera unas pocas decenas de líneas, considera dividirlo.
- **Reutilización y mantenibilidad:** Código modular es más fácil de mantener, probar y extender.

Problema: reestructuración del código para mejor mantenibilidad

Contexto: Actualmente, el `main` en TechLab hace de todo: lee datos, calcula descuentos, actualiza el inventario, imprime resultados. Silvia reclama mayor claridad. Matías y Sabrina deciden modularizar.

Solución:

- Un método para leer datos del usuario.
- Un método para calcular el precio final con descuento.
- Un método para imprimir el resultado.
- Un método para actualizar el inventario.

Esto crea un flujo más claro y fácil de mantener.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Materiales y Recursos Adicionales

- Documentación Oficial de Java
- Guía sobre Métodos en Java (Oracle)
- Videos Recomendados:
    - "Organización del Código con Métodos" en YouTube.

Preguntas para reflexionar

- ¿De qué manera las funciones facilitan la lectura y mantenimiento del código en un proyecto grande como el de TechLab?
- ¿Cómo la sobrecarga de métodos te permite enfrentar diferentes escenarios sin cambiar el nombre de las funciones?
- ¿Qué beneficios trae la modularización al agregar nuevas funcionalidades o modificar las existentes?

Situación actual en TalentoLab

Talento Lab

Silvia, la Product Owner, acaba de asignar una nueva tarea al equipo de desarrollo: el sistema debe aplicar distintos tipos de descuentos a los productos según la categoría, la cantidad comprada y el tipo de cliente. Matías y Sabrina se dan cuenta de que el código está creciendo y volviéndose complejo. Necesitan una forma de organizar el código en bloques más pequeños y reusables: las funciones.

Además, habrá escenarios donde será necesario crear métodos con el mismo nombre pero diferentes parámetros, gestionar el impacto de las variables dentro de los métodos y mantener un código claro y fácilmente ampliable.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Ejercicios prácticos

Matias
Desarrollador

Sabrina
Desarrolladora

Ayudemos a Martin y Sabrina a desarrollar un código modular y organizado de manera que sea fácil de entender y mantener, aplicando conceptos trabajados vistos en la documentación. Para hacerlo necesitaremos:

1. **Crear de Funciones:**
   - Escribí un método `calcularImpuesto` que reciba un precio y un porcentaje de impuesto, y devuelva el precio final.
   - Llamá al método desde `main` con distintos valores.
2. **Parámetros y Argumentos:**
   - Creá un método `calcularPrecioFinal` que reciba el precio, el descuento y la cantidad. Si la cantidad es mayor a 50, aumentá el descuento. Imprimí el resultado.
3. **Paso por Valor:**
   - Definí un array con el stock de 3 productos.
   - Escribí un método `reponerStock` que sume unidades a un índice específico del array.
   - Mostrá el stock antes y después de la reposición.
4. **Sobrecarga de Métodos:**
   - Creá dos métodos `mostrarMensaje`: uno recibe una `String`, otro recibe una `String` y un `int`.
   - Usá ambos para imprimir mensajes en función de si el cliente es nuevo o recurrente.
5. **Modularización:**
   - Tomá un código previo (como el de la clase anterior) y extraé partes en métodos.
   - Comentá por qué ahora es más fácil de leer y mantener.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Próximos pasos

En la siguiente clase exploraremos aún más las estructuras de datos y cómo manipularlas. Mientras tanto, practicá creando y llamando métodos para distintas tareas, aplicá sobrecarga cuando sea pertinente y buscá mejorar la modularización de tu código. ¡Hasta la próxima!

Buenos Aires Ciudad