**Introducción**

El debugging es una habilidad esencial para cualquier desarrollador. Permite identificar y corregir errores en el código, asegurando que funcione como se espera. IntelliJ IDEA ofrece una poderosa herramienta de debugging que facilita este proceso. En esta práctica, aprenderás a usar las funcionalidades básicas de debugging en IntelliJ IDEA.

**Objetivos**

Configurar un entorno de debugging.

Utilizar breakpoints.

Inspeccionar variables y el flujo del programa.

Utilizar herramientas avanzadas de debugging como el Step Into, Step Over y Step Out.

**Requisitos**

IntelliJ IDEA instalado.

Conocimientos básicos de Java.

**Parte 1: Configuración del entorno de debugging y crear un proyecto en IntelliJ IDEA con Gradle**

**Paso 2:** Crear una clase Java y nombra la clase DebugExample.

Código de ejemplo

Copia el siguiente código en tu clase DebugExample:

public class DebugExample {

public static void main(String[] args) {

int a = 5;

int b = 10;

int result = add(a, b);

System.out.println("Result: " + result);

}

public static int add(int x, int y) {

int sum = x + y;

return sum;

}

}

**Parte 2: Utilizando breakpoints**

**Paso 1: Añadir un breakpoint**

Haz clic en el margen izquierdo junto al número de línea donde quieres añadir un breakpoint (por ejemplo, en la línea int result = add(a, b);).

Aparecerá un punto rojo indicando que se ha añadido un breakpoint.

**Paso 2: Iniciar el modo de debugging**

Haz clic derecho en el archivo DebugExample.java.

Selecciona "Debug 'DebugExample.main()'".

El programa se ejecutará y se detendrá en el breakpoint que has añadido.

**Parte 3: Inspección de variables y flujo del programa**

**Paso 1: Inspeccionar variables**

Una vez detenido el programa en el breakpoint, pasa el cursor sobre la variable a, b y result.

Verás los valores actuales de estas variables en una ventana emergente.

**Paso 2: Utilizar la ventana de debugging**

En la parte inferior de IntelliJ IDEA, verás la ventana de debugging.

Usa la pestaña "Variables" para ver y monitorear todas las variables locales.

Usa la pestaña "Frames" para ver la pila de llamadas.

**Parte 4: Herramientas avanzadas de debugging**

**Paso 1: Step Over**

Haz clic en el botón "Step Over" (representado por un icono de flecha curva) o presiona F8.

Esto ejecutará la línea actual y se detendrá en la siguiente línea del método actual.

**Paso 2: Step Into**

Haz clic en el botón "Step Into" (representado por un icono de flecha hacia abajo) o presiona F7.

Esto te permitirá entrar en el método add y detenerte en su primera línea.

**Paso 3: Step Out**

Haz clic en el botón "Step Out" (representado por un icono de flecha hacia arriba) o presiona Shift+F8.

Esto ejecutará el resto del método actual y se detendrá en la línea que sigue después de la llamada al método.

**Paso 4: Continuar la ejecución**

Haz clic en el botón "Resume Program" (representado por un icono de flecha recta) o presiona F9.

Esto reanudará la ejecución del programa hasta el siguiente breakpoint o hasta el final del programa.

**Parte 5: Ejercicio práctico**

Desafío

Modifica el código para incluir un nuevo método multiply que multiplique dos números. Añade breakpoints y utiliza las herramientas de debugging para asegurarte de que los métodos add y multiply funcionan correctamente.

**Parte 6: Ejercicio práctico**

Encuentra y clasifica los errores en el código presentado a continuación

***Ejercicio a***

public class ClasificacionErrores {

public static void main(String[] args) {

int[] numeros = {1, 2, 3, 4, 5};

System.out.println("El tercer elemento es: " + numeros[3]); // Error lógico

String mensaje = null;

System.out.println(mensaje.length()); // Error de ejecución

for(int i = 0; i <= numeros.length; i++) { // Error de ejecución

System.out.println("Número: " + numeros[i]);

}

int resultado = dividir(10, 0);

System.out.println("Resultado de la división: " + resultado); // Error de ejecución

System.out.println(saludo("Juan")); // Error de compilación

}

public static int dividir(int a, int b) {

return a / b; // Error de ejecución

}

public static String saludo(String nombre) {

return "Hola, " + nombre;

}

}

***Ejercicio b***

import java.util.HashMap;

import java.util.Map;

public class GestionEstudiantes {

public static void main(String[] args) {

Map<String, Integer> estudiantes = new HashMap<>();

estudiantes.put("Ana", 85);

estudiantes.put("Juan", 90);

estudiantes.put("Maria", 75);

estudiantes.put("Pedro", 60);

System.out.println("Nota media: " + calcularNotaMedia(estudiantes));

}

public static double calcularNotaMedia(Map<String, Integer> estudiantes) {

int suma = 0;

for (String nombre : estudiantes.keySet()) {

suma += estudiantes.get(nombre);

}

return suma / estudiantes.size(); // Error semántico: División entera

}

}

***Ejercicio c***

package com.example.demo.model;

public class Product {

private Long id;

private String name;

private double price;

// Constructor, getters y setters

public Product() {}

public Product(Long id, String name, double price) {

this.id = id;

this.name = name;

this.price = price;

}

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public double getPrice() {

return price;

}

public void setPrice(double price) {

this.price = price;

}

}

package com.example.demo.controller;

import com.example.demo.model.Product;

import org.springframework.web.bind.annotation.\*;

import java.util.ArrayList;

import java.util.List;

@RestController

@RequestMapping("/products")

public class ProductController {

private List<Product> products = new ArrayList<>();

@GetMapping

public List<Product> getAllProducts() {

return products;

}

@PostMapping

public String addProduct(@RequestBody Product product) {

products.add(product);

return "Product added successfully";

}

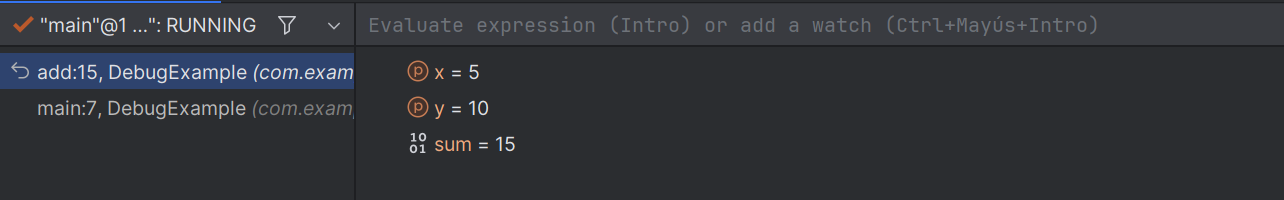
}

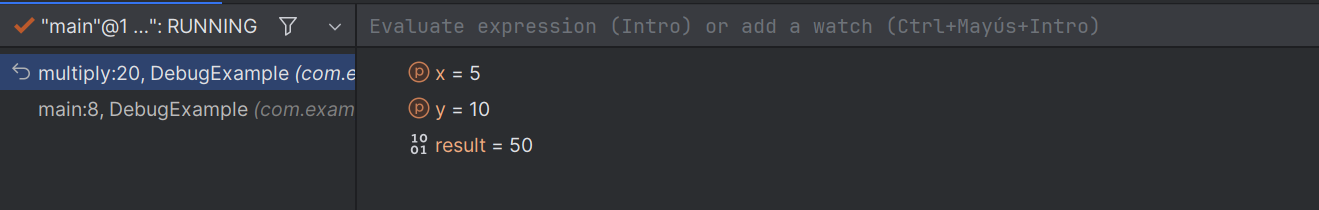
**Tareas para entregar**

1. Código fuente actualizado con modificaciones de los ejercicios

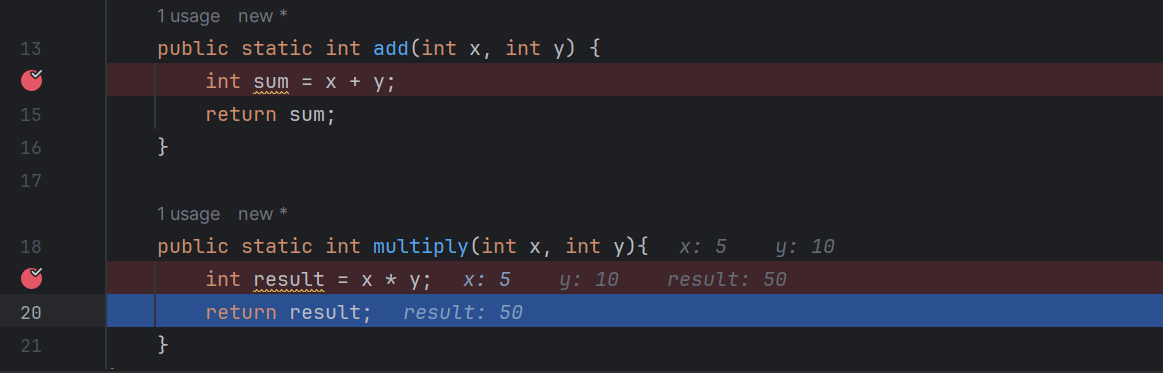


1. Capturas de pantalla del entorno de debugging

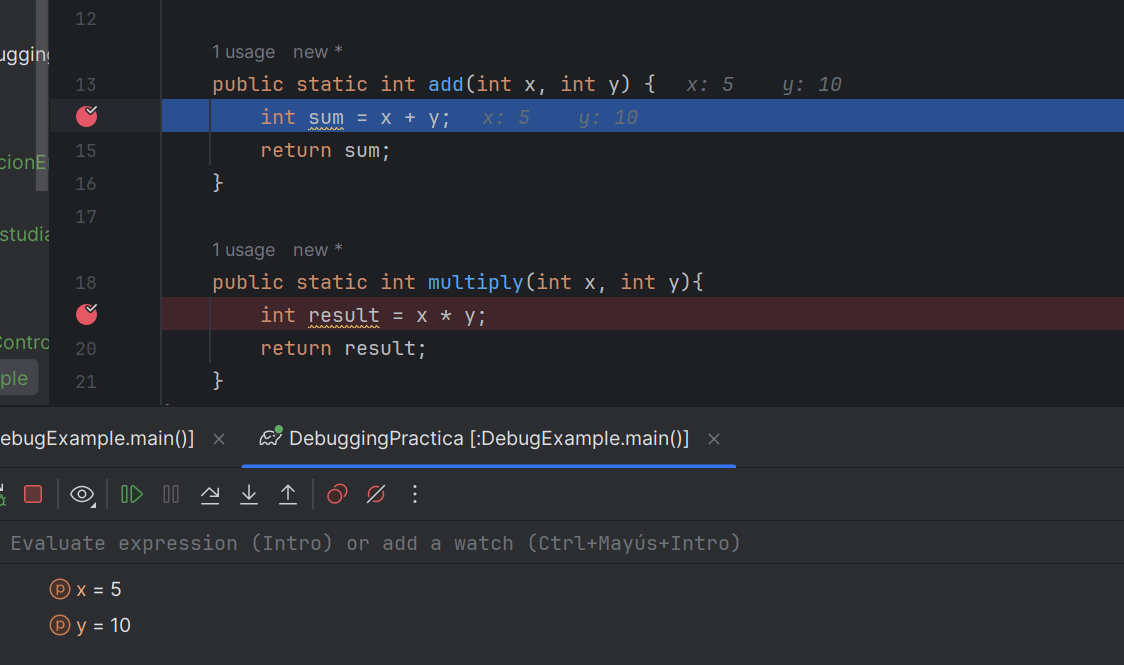


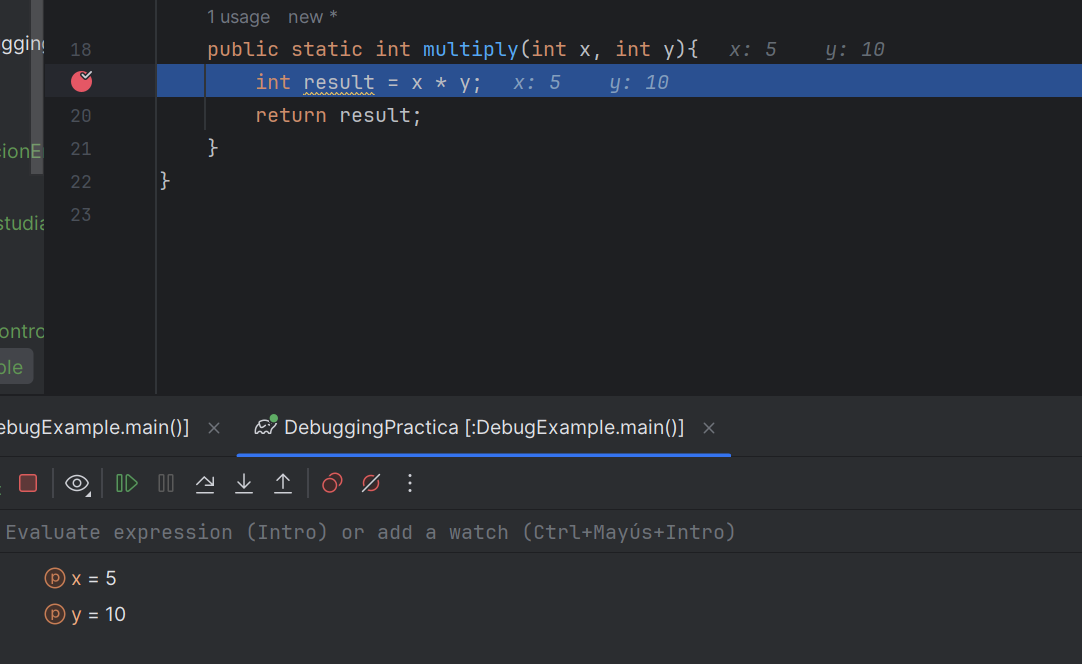


1. El breakpoint añadido en el método main.

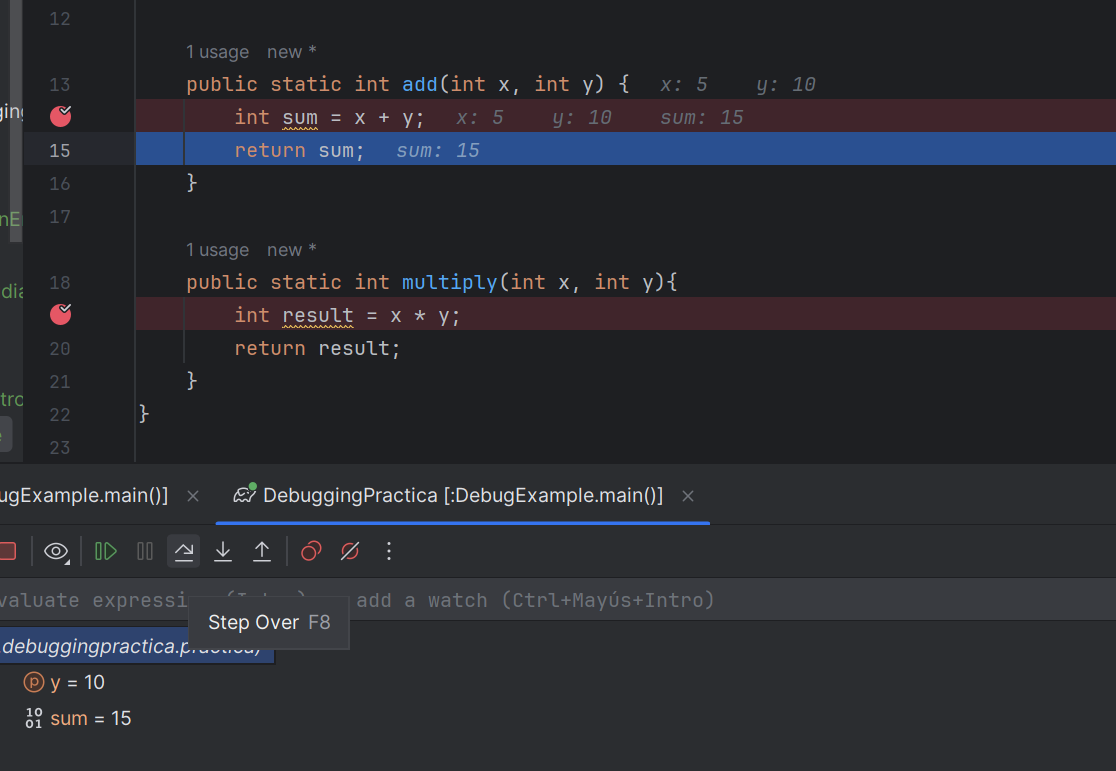


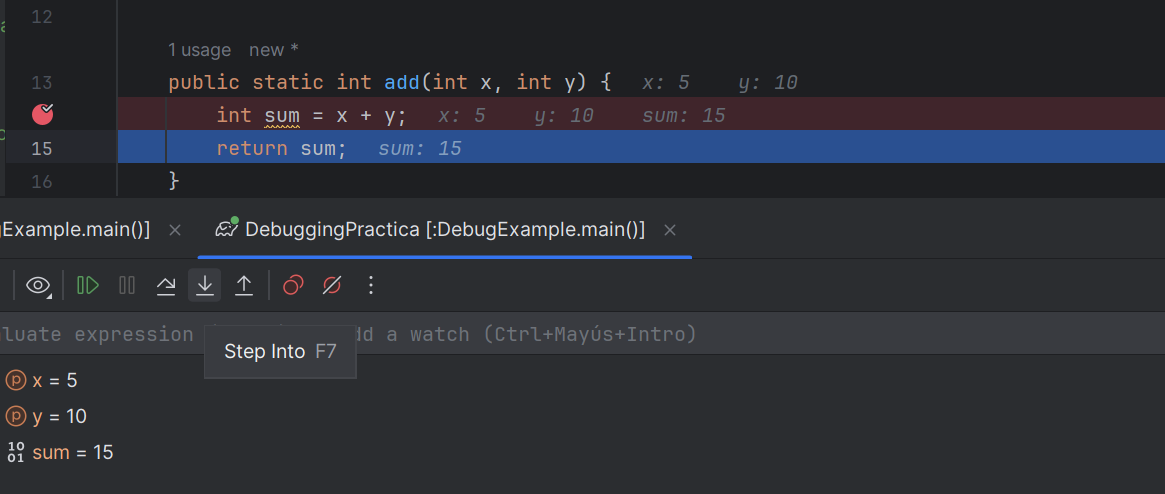
1. El programa detenido en el breakpoint, mostrando las variables en la ventana de debugging.

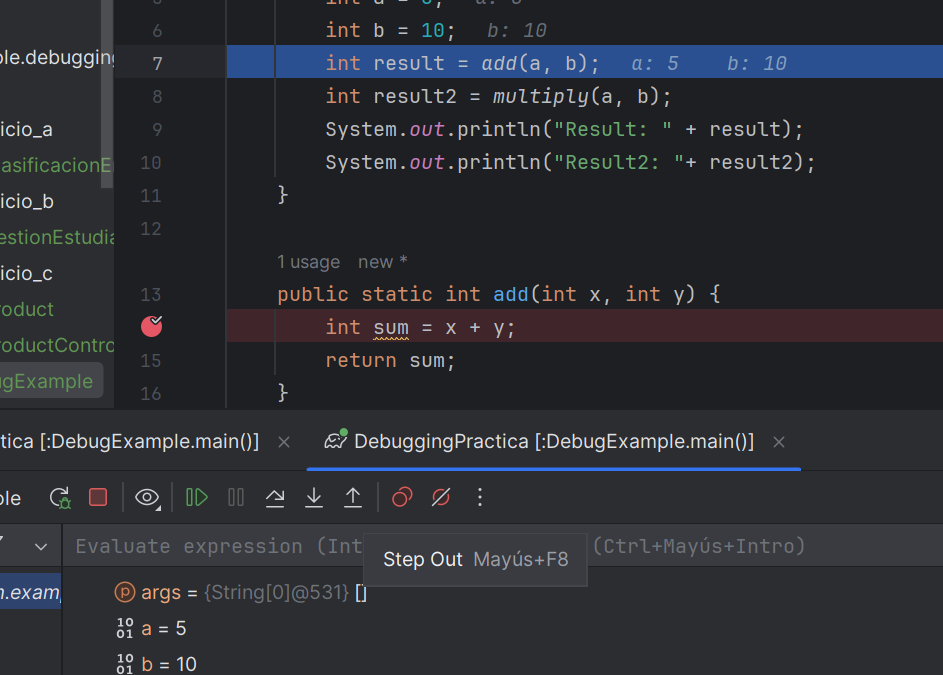


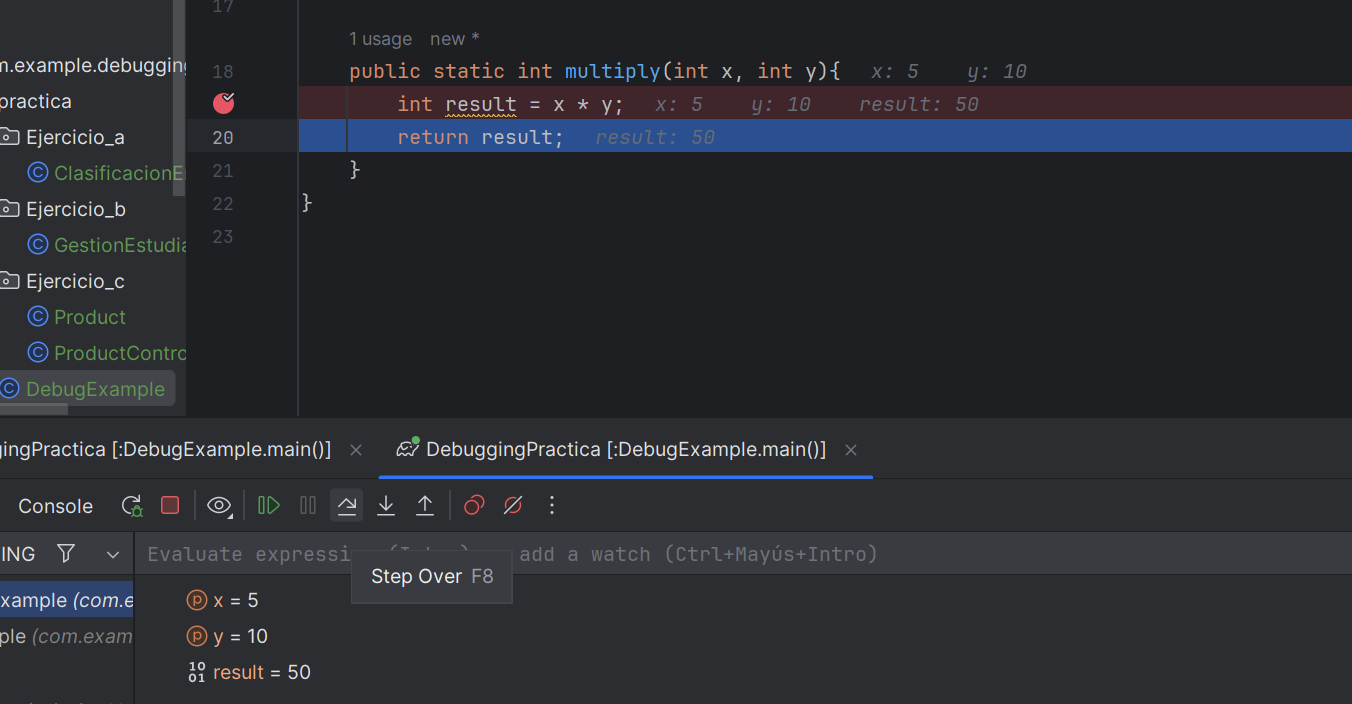


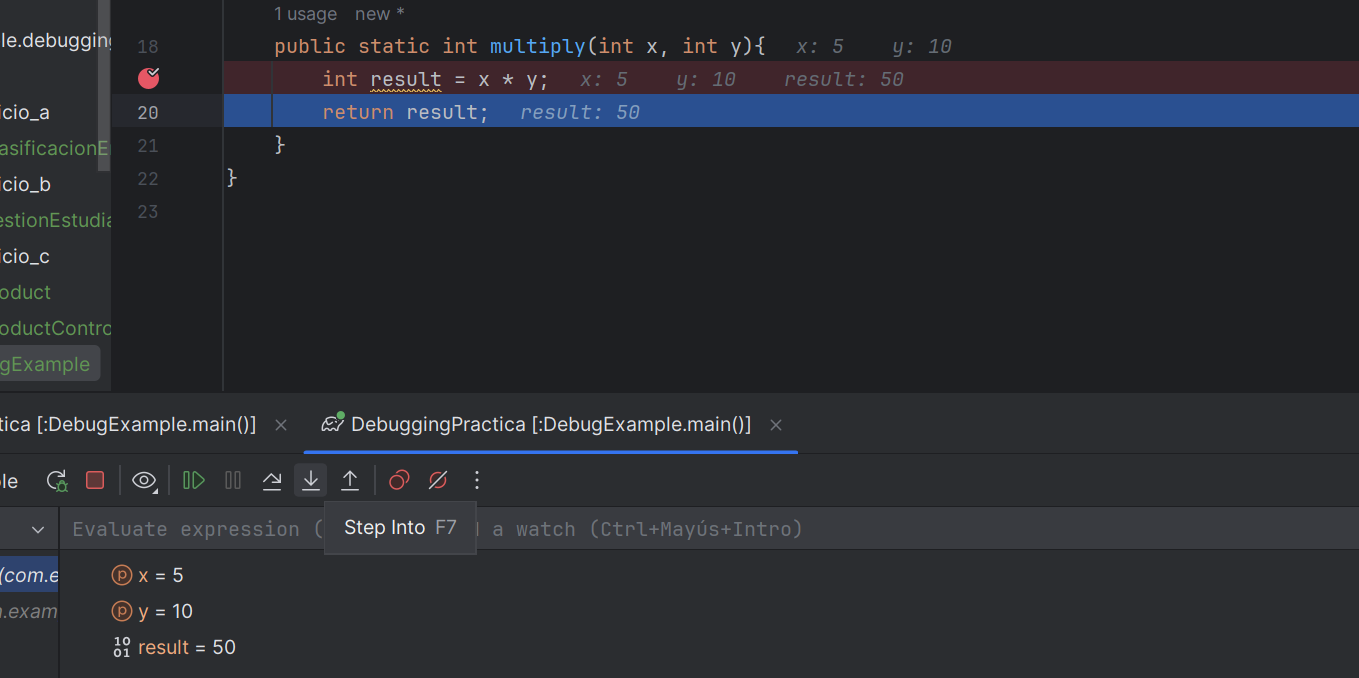
1. La ejecución paso a paso (Step Over, Step Into, Step Out) de ambos métodos (add y multiply).

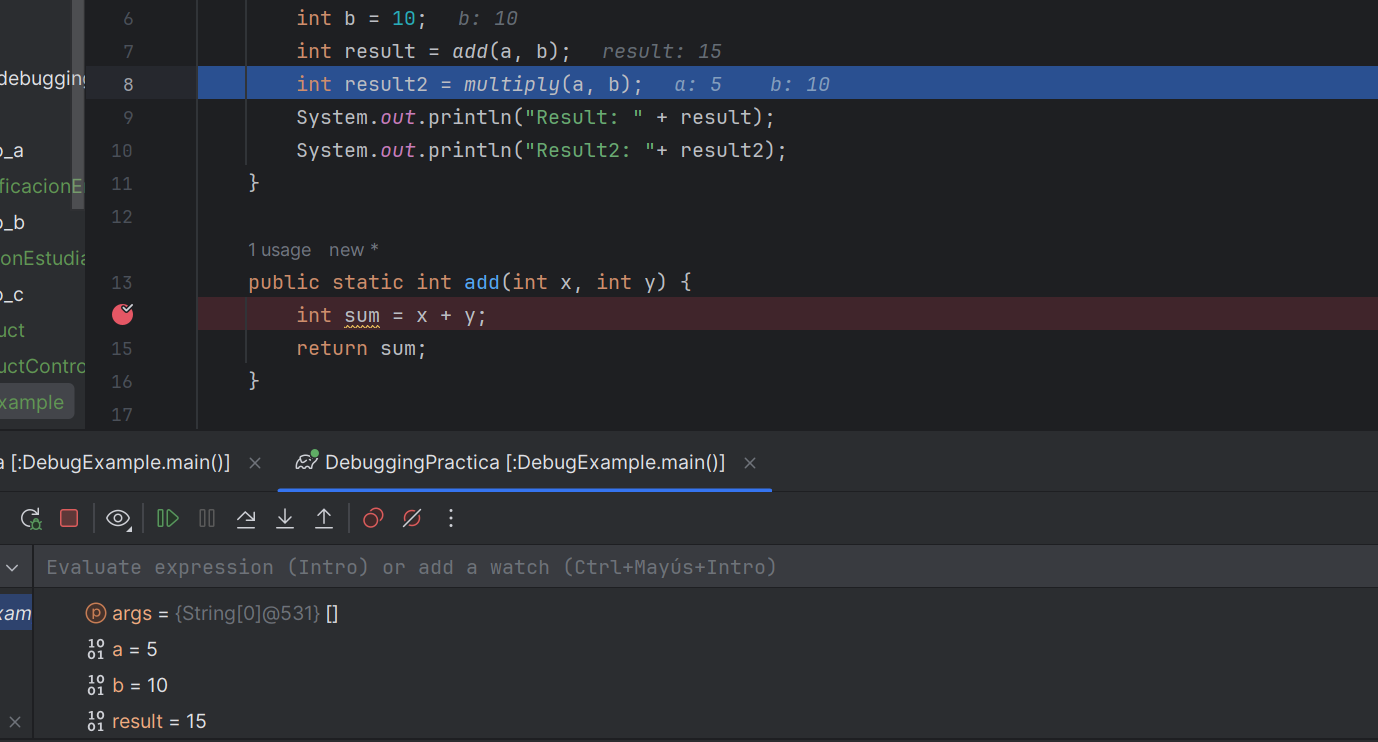












1. Resultados impresos en consola

Copiar y pegar en el informe los resultados impresos en la consola.

