

# Depuración

Entornos de desarrollo

Diapositivas realizadas por Aitor Ventura Edo

IES El Caminás  
Curso 2025–2026

# Depuración



La depuración es un proceso clave en el desarrollo de software que permite identificar y corregir errores mediante la ejecución controlada del programa.

Esta tarea se realiza con el depurador, una herramienta esencial del entorno de desarrollo. Su uso se vuelve indispensable para crear programas libres de fallos, ya que facilita la identificación de problemas en el código.

# Ejemplo

```
public class Main {  
    public static void main(String[] args) {  
        int[] numbers = {1, 2, 3, 4, 5};  
        int sum = 0;  
  
        for (int i = 0; i <= numbers.length; i++) {  
            sum += numbers[i];  
        }  
  
        System.out.println("La suma de los números es: " + sum);  
    }  
}
```

# Ejemplo

Si ejecuto el ejemplo anterior aparece:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:  
Index 5 out of bounds for length 5

at activities.Main.main(Main.java:9)

¡SPOILER! En este caso, el bucle tiene una condición incorrecta:

`i <= numbers.length` debería ser `i < numbers.length`.

Esto provoca un `ArrayIndexOutOfBoundsException` porque el índice `i` llega a un valor igual al tamaño del array, lo cual no es válido.

# Depurador

El depurador permite analizar un programa **paso a paso**, observando los valores de métodos, variables y objetos en tiempo real. Además, permite establecer **puntos de interrupción** (breakpoints) para detener la ejecución en puntos específicos del código y examinar su estado en ese momento, optimizando la resolución de errores.



# Iniciar el depurador

Un **depurador** no solo corrige errores, sino que **permite analizar y comprender cómo funciona el código**, e incluso **modificar su comportamiento sin cambiar la fuente**. Para usarlo en una aplicación de consola, **se puede iniciar desde**:

- El **área de números de línea** (haciendo clic en la zona gris) seleccionando Debug 'Main'.
- El **menú Run → Debug 'Main'**.
- El **atajo Shift + F9**.

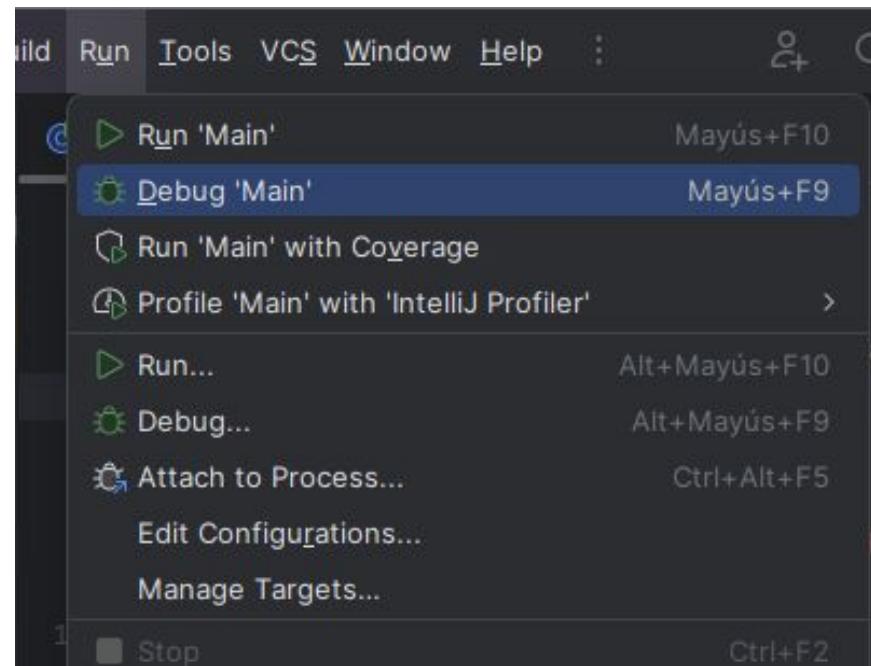
## IMPORTANTE:

Si **no se definen puntos de interrupción**, el programa **se ejecuta normalmente**. Para depurar, **es necesario establecer al menos un punto de interrupción**.

# Iniciar el depurador

A screenshot of the IntelliJ IDEA interface showing the code editor for a Java file named Main.java. The code defines a Main class with a main method that prints the sum of numbers from 1 to 5. A context menu is open over the main method, with the 'Debug 'Main.main()'' option highlighted in blue. Other options in the menu include 'Run 'Main.main()'', 'Run 'Main.main()' with Coverage', and 'Profile 'Main.main()' with 'IntelliJ Profiler''. The menu also contains a 'Modify Run Configuration...' option.

```
1 package activities;
2
3 public class Main {
4     public static void main(String[] args) {
5         int i = 1;
6         int sum = 0;
7         while (i <= 5) {
8             sum += i;
9             i++;
10        }
11        System.out.println("La suma de los números es: " + sum);
12    }
13 }
14 }
```



# Ejecución sin puntos de interrupción

Connected to the target VM, address: '127.0.0.1:61055', transport: 'socket'

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index  
5 out of bounds for length 5  
at activities.Main.main(Main.java:9)

Disconnected from the target VM, address: '127.0.0.1:61055', transport:  
'socket'

# Puntos de interrupción

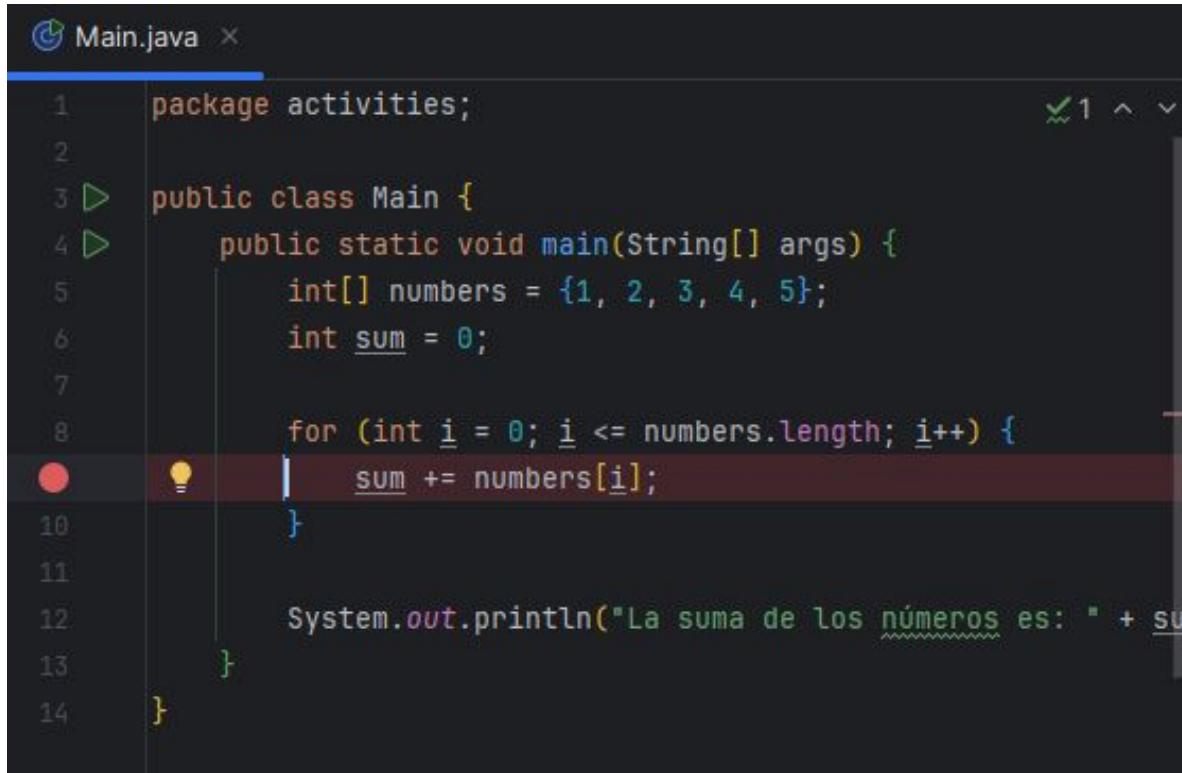
Un punto de interrupción (breakpoint) es un punto de control en el código donde el depurador detiene la ejecución del programa. Es **esencial definirlos antes de depurar**.

Cómo establecer puntos de interrupción en IntelliJ:

- Haz clic con el botón izquierdo del ratón en el área izquierda junto a la línea deseada. Aparecerá una bola roja, y la línea se marcará en rojo.
- Alternativamente, usa el atajo Ctrl + F8.

Se pueden agregar múltiples puntos de interrupción y eliminarlos haciendo clic nuevamente sobre ellos.

# Breakpoint añadido



The screenshot shows a Java code editor with a dark theme. The file is named Main.java. A red circular breakpoint icon is visible on the left margin at line 9, indicating that the debugger will stop execution at that point. The code itself is a simple program that calculates the sum of the first five integers:

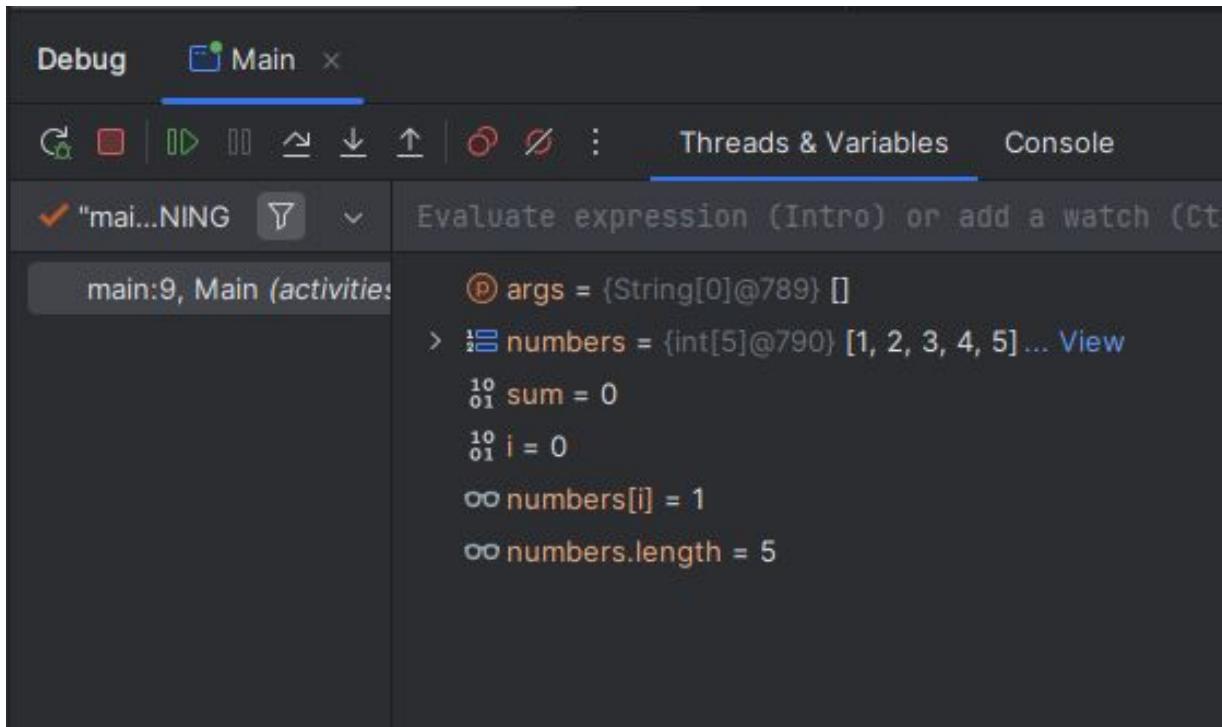
```
1 package activities;
2
3 public class Main {
4     public static void main(String[] args) {
5         int[] numbers = {1, 2, 3, 4, 5};
6         int sum = 0;
7
8         for (int i = 0; i <= numbers.length; i++) {
9             sum += numbers[i];
10        }
11
12        System.out.println("La suma de los números es: " + sum);
13    }
14 }
```

# Vista de depuración

Al **iniciar el depurador** en el IDE, aparece la ventana Debug en la parte inferior. El programa se ejecuta normalmente hasta llegar a un **breakpoint**, donde **se detiene** para permitir tareas de depuración como:

- Visualizar los valores de las **variables**.
- Ejecutar el programa **paso a paso**.
- Analizar quién llama a quién, entre otras opciones.

# Vista de depuración



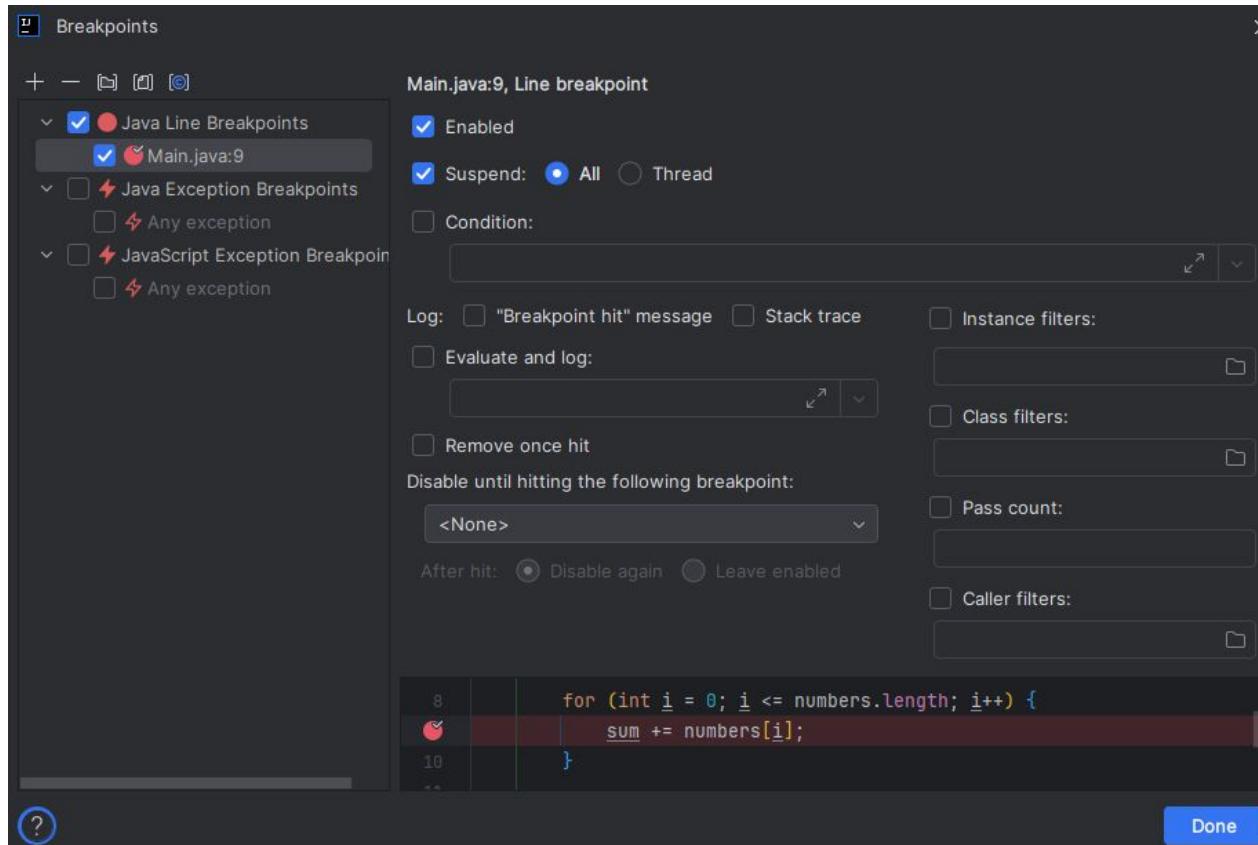
# View Breakpoints

Si se quieren ver **todos los puntos de interrupción** y su ubicación, podemos ejecutar **Run → View Breakpoints** o con el atajo **Ctrl + Shift + F8** y nos aparecerá una ventana como la de la siguiente diapositiva.

Desde esa ventana se pueden **ver, agregar, suspender o eliminar puntos de interrupción**. Además, se puede **agregar información adicional a cada punto de interrupción, añadirle condiciones, registrarla y otras acciones**.

A continuación se muestran ejemplos de unas de las más usadas.

# View Breakpoints

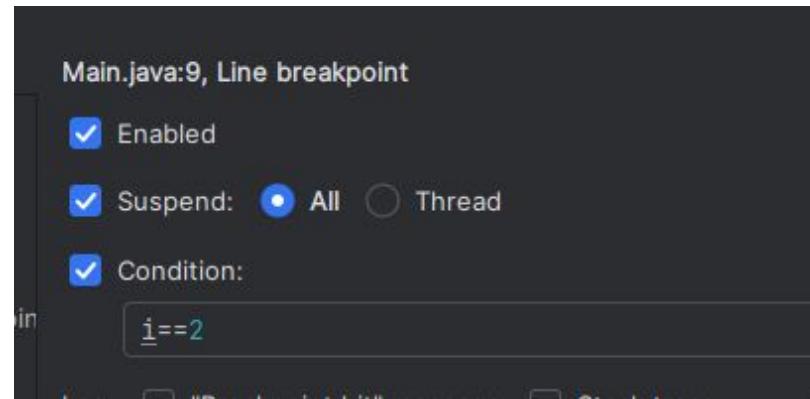


# Condition

## Agregar un punto de interrupción condicional

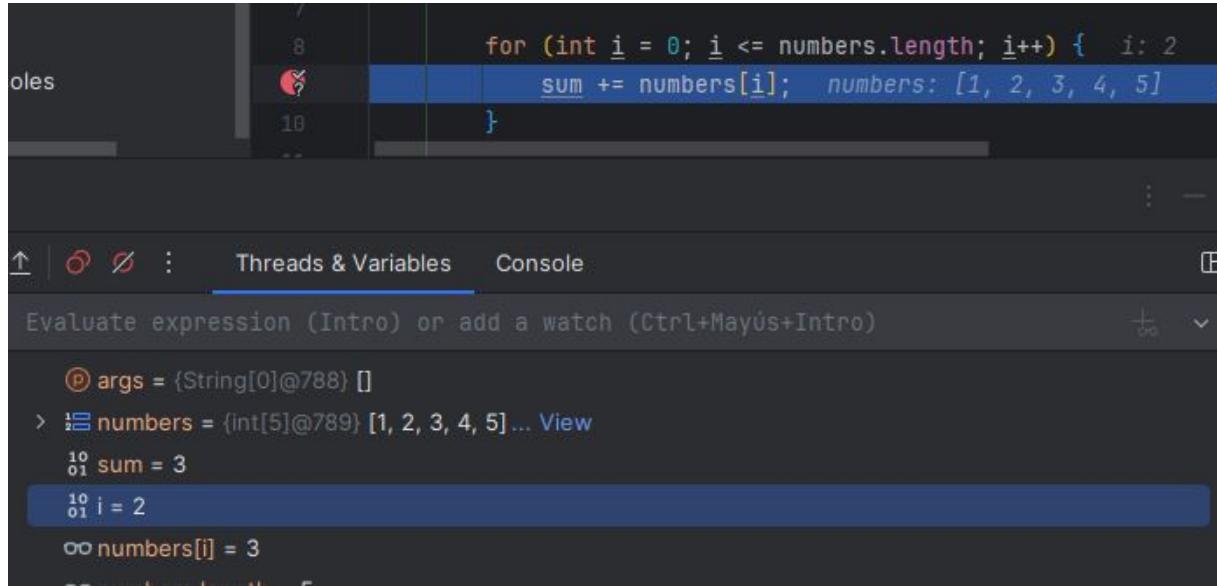
Desde la ventana de breakpoints:

- Selecciona el punto de interrupción en la línea del bucle: `sum += numbers[i];`
- Añade una condición como `i == 2` para detener la ejecución solo cuando el índice sea igual a 2.



# Condition

Resultado: El programa **solo** se detendrá cuando `i` valga 2, permitiendo analizar el estado del programa en esa iteración específica.



The screenshot shows a debugger interface with a code editor and a watch window. The code editor displays a Java-like snippet:for (int i = 0; i <= numbers.length; i++) { i: 2 sum += numbers[i]; numbers: [1, 2, 3, 4, 5] s}The line number 8 is highlighted, and the variable `i` is set to 2, indicated by a red dot in the gutter and the value 2 in the code editor. The watch window below shows variables `args`, `numbers`, `sum`, and `i`. The variable `i` is currently selected, showing its value as 2.

Threads & Variables      Console

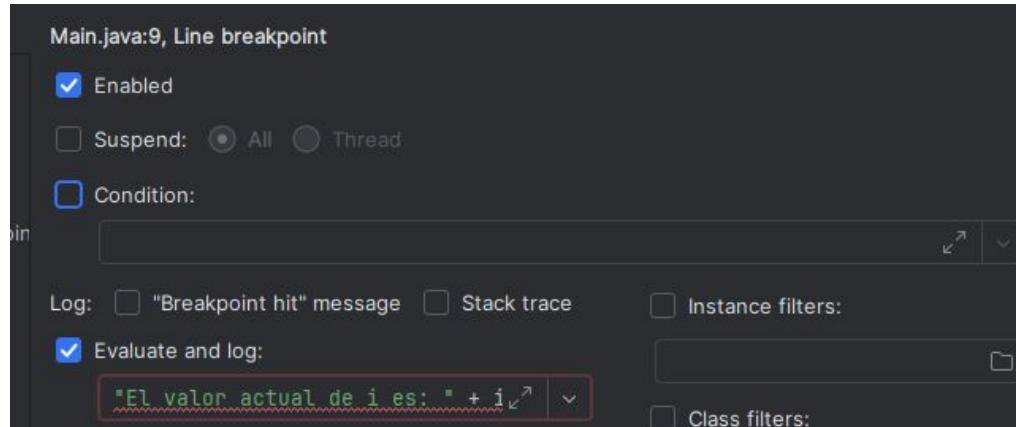
Evaluate expression (Intro) or add a watch (Ctrl+Mayús+Intro)

args = {String[0]@788} []  
numbers = {int[5]@789} [1, 2, 3, 4, 5] ... View  
sum = 3  
i = 2  
numbers[i] = 3

# Evaluate and log

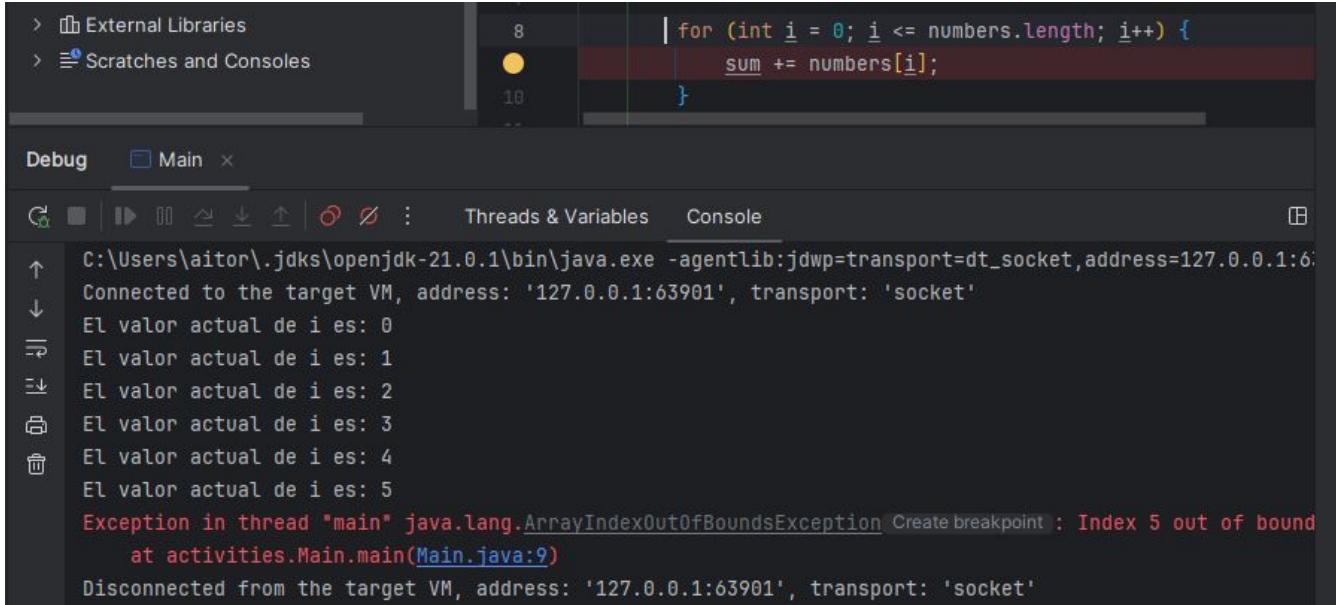
## Agregar acciones adicionales a un breakpoint

Puedes configurar breakpoints para realizar otras tareas en lugar de detener el programa. Configura el breakpoint en la línea `sum += numbers[i];` para registrar un mensaje como: "El valor actual de i es: " + i. Además, desactiva suspend para que no se paralice al llegar al breakpoint.



# Evaluate and log

Resultado: Cada vez que el programa pase por ese punto, aparecerá el mensaje en la consola sin detener la ejecución.



The screenshot shows a Java debugger interface with the following details:

- Breakpoint:** A yellow circle at line 8 of the code indicates a breakpoint.
- Code:** The code is a simple loop:

```
for (int i = 0; i <= numbers.length; i++) {  
    sum += numbers[i];  
}
```
- Console Output:** The console shows the value of the variable `i` being printed each time it reaches the breakpoint:

```
El valor actual de i es: 0  
El valor actual de i es: 1  
El valor actual de i es: 2  
El valor actual de i es: 3  
El valor actual de i es: 4  
El valor actual de i es: 5
```
- Error Message:** An exception is caught at the end of the loop:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds  
        at activities.Main.main(Main.java:9)
```
- Final Message:** The connection is disconnected from the target VM.

```
Disconnected from the target VM, address: '127.0.0.1:63901', transport: 'socket'
```

# Remove once hit

**Función:** El breakpoint se elimina automáticamente después de que se active una vez.

**Uso típico:** Útil cuando quieres depurar un evento que ocurre solo una vez, como verificar una condición específica o inspeccionar una única iteración.

**Ejemplo:** En el bucle:

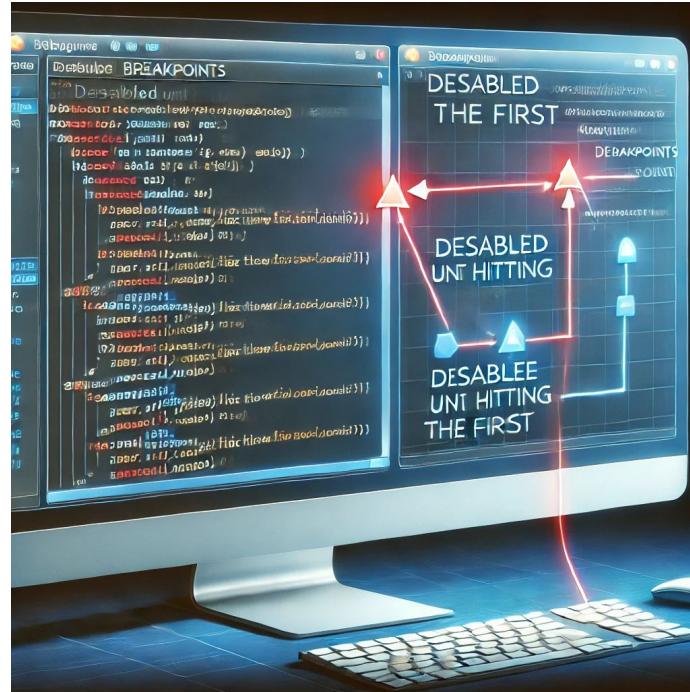
```
for (int i = 0; i <= numbers.length; i++) {  
    sum += numbers[i];  
}
```

Si quieras detenerte únicamente la primera vez que *i* es igual a 2, configuras esta opción.

# Disable until hitting the following breakpoint

**Función:** El breakpoint permanece inactivo hasta que otro breakpoint especificado sea alcanzado.

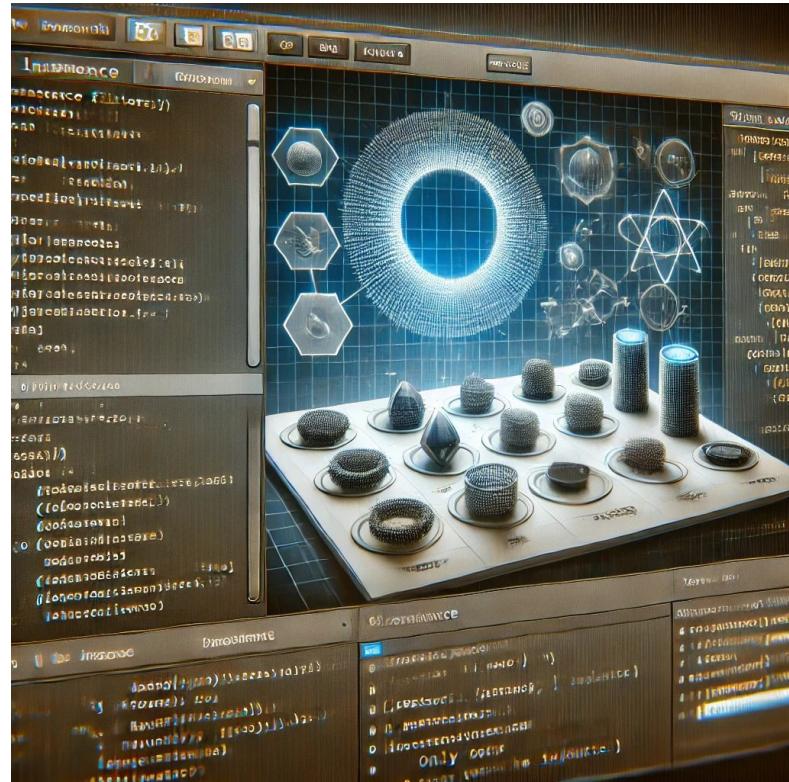
**Uso típico:** Para coordinar la ejecución entre múltiples puntos de interrupción y evitar detenerte innecesariamente antes de un contexto relevante.



# Instance filters

**Función:** Restringe la activación del breakpoint solo a instancias específicas de una clase.

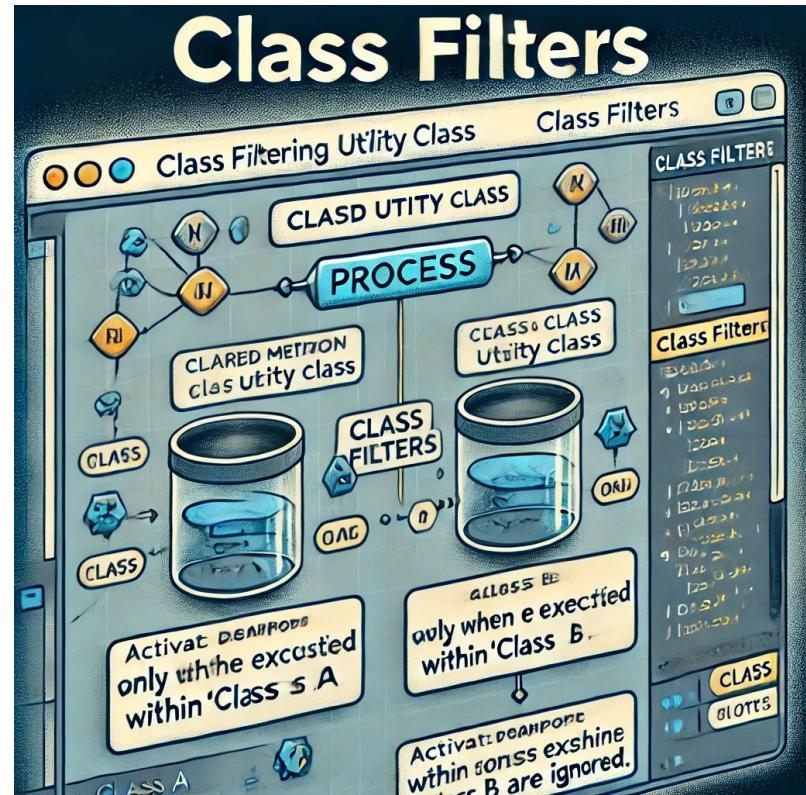
**Uso típico:** Ideal cuando tienes múltiples objetos de una misma clase y solo necesitas depurar uno en particular.



# Class filters

**Función:** Permite restringir el breakpoint para que se active solo cuando el código se ejecuta en una clase específica.

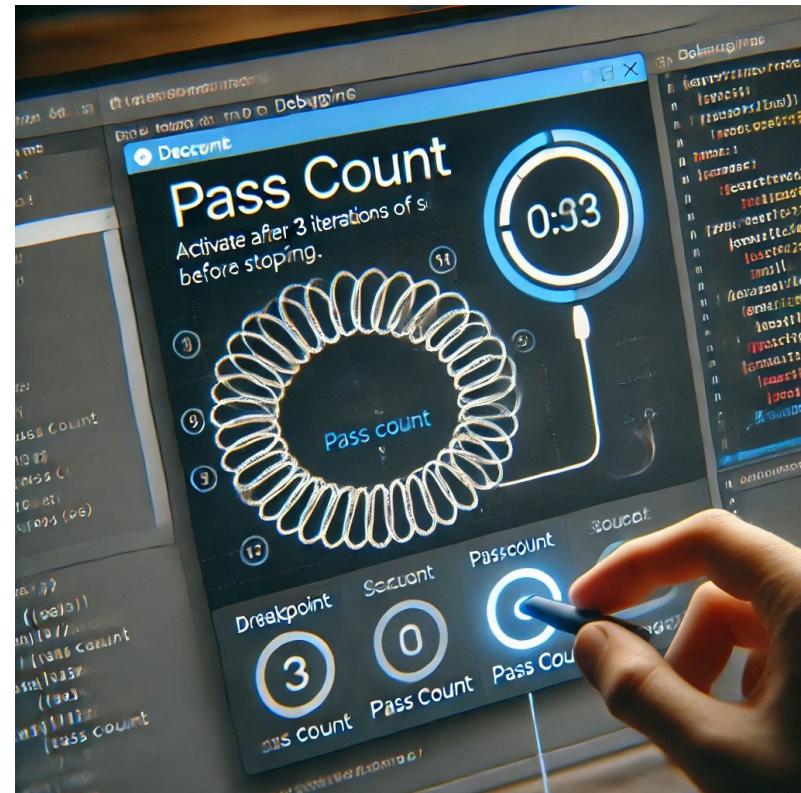
**Uso típico:** Útil en programas grandes con muchas clases, para concentrarte en la ejecución dentro de una clase particular.



# Pass count

**Función:** Configura el breakpoint para que se active solo después de que se haya alcanzado un número específico de veces.

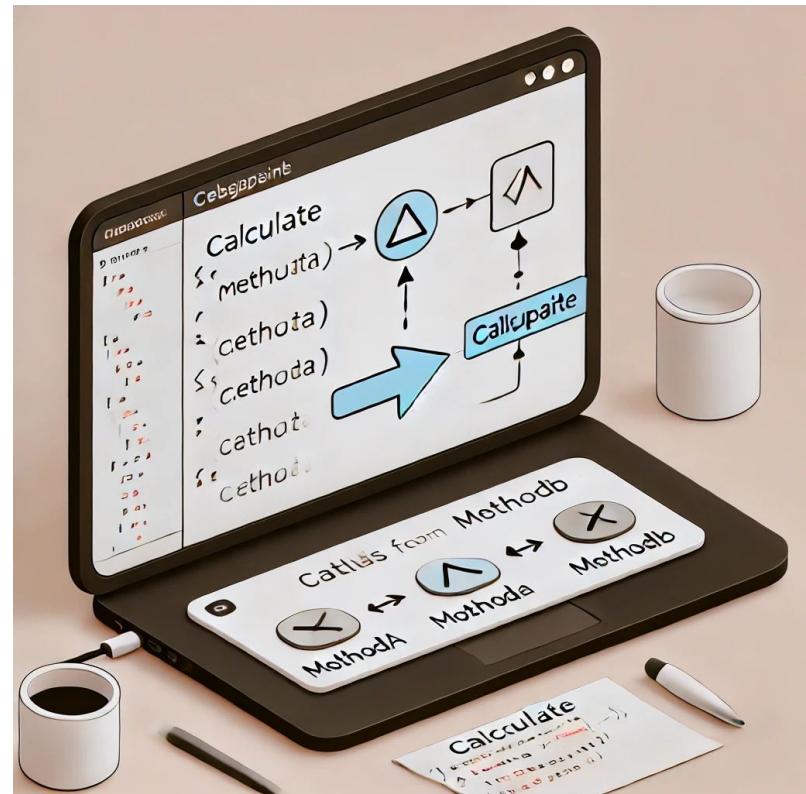
**Uso típico:** Cuando un error ocurre solo después de varias iteraciones o repeticiones.



# Caller filters

**Función:** Restringe la activación del breakpoint dependiendo de qué método o función haya llamado el código.

**Uso típico:** Para analizar el flujo de llamadas y detectar problemas que ocurren solo cuando una función específica invoca el código.



# El proceso de depuración

- Partiendo del ejemplo visto anteriormente, cuando ejecutamos el código en modo depuración, colocamos un breakpoint en la línea:

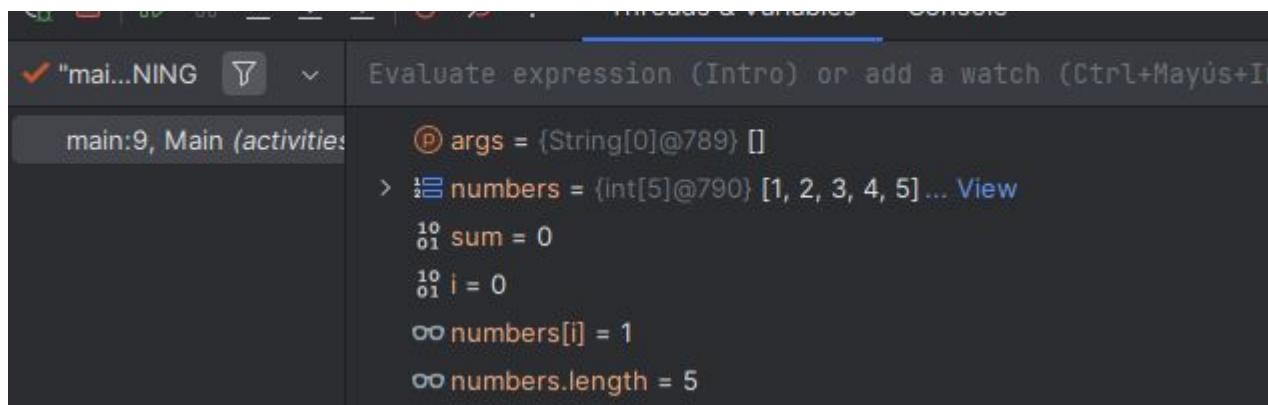
`sum += numbers[i];`

- Esto permitirá detener la ejecución del programa cada vez que el bucle alcance esta línea. Desde este punto, podemos observar y manipular el estado de las variables.

# Visualización de variables

Al detenerse en el breakpoint, la pestaña Variables mostrará:

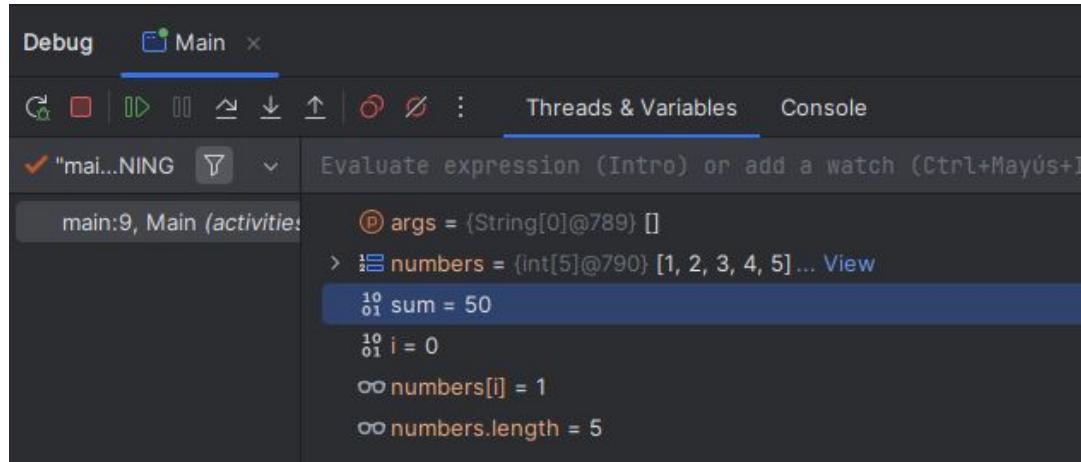
- `i`: El índice actual del bucle.
- `numbers[i]`: El valor del elemento actual del array.
- `sum`: El acumulado de la suma hasta esa iteración.



# Modificación de valores

Supongamos que en la primera iteración `sum` tiene un valor de 0. Podemos cambiar su valor a 50 haciendo clic derecho sobre `sum` en la pestaña Variables, seleccionando Set Value... y asignando 50.

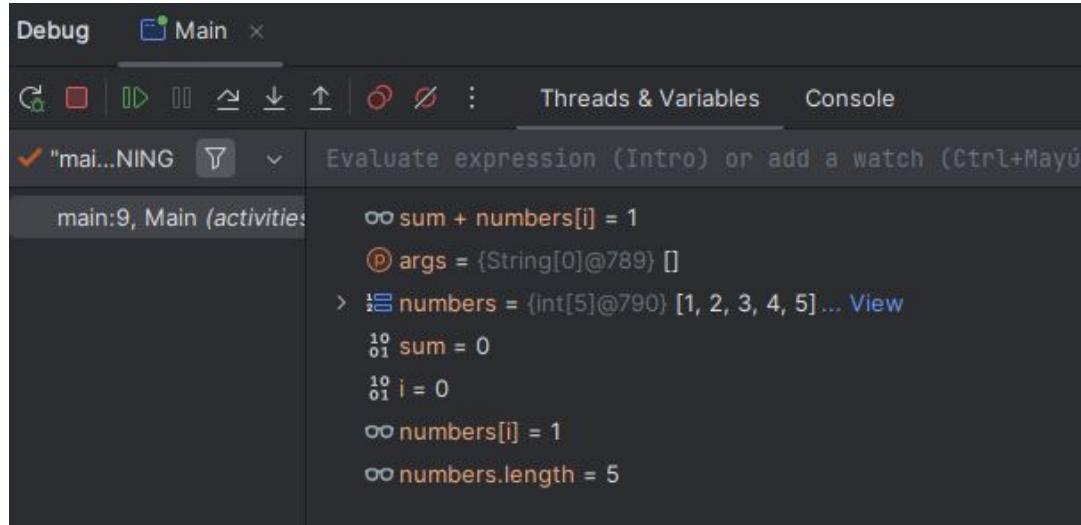
Esto permitirá observar cómo afecta este cambio al cálculo final de la suma.



# Seguimiento de variables (Watch)

Podemos añadir expresiones dinámicas como `sum + numbers[i]` para verificar cómo evolucionan los valores.

Esto se hace presionando `+ New Watch...` en la ventana de depuración.



# Funciones de depuración paso a paso



## Step Over (F8):

- Avanzamos **línea por línea**.
- En nuestro ejemplo, en **cada iteración del bucle**, el depurador **pasará a la siguiente línea**, permitiendo observar cómo cambia **sum**.
- No entra en los métodos.



## Step Into (F7):

- Igual que Step Over pero **si el programa tuviera un método específico para realizar la suma, por ejemplo: `sum = add(sum, numbers[i]);`**
- Con F7, **podríamos entrar en el método add para depurar su contenido.**

# Funciones de depuración paso a paso



## Step Out (Shift F8):

Step Out se utiliza para **salir de un método en el que te encuentras durante la depuración** y volver al punto desde donde se llamó ese método. Es una forma de "abandonar" el método actual y regresar al contexto superior sin tener que ejecutar manualmente todas las líneas restantes dentro del método.



## Resume Program (F9):

Si queremos que el programa **siga ejecutándose sin detenerse en los próximos pasos**, usamos esta función para que corra hasta el siguiente breakpoint.

# Funciones de depuración paso a paso



Evaluate Expression (Alt + F8):

Esta función permite **evaluar expresiones dinámicas en tiempo real**. Por ejemplo:

- Evaluar `sum + numbers[i]` para predecir el próximo valor de `sum`.
- Verificar si `i == 3` para condicionar futuras acciones.
- Si necesitas **cambiar el valor de una variable**, escribe la nueva asignación. Por ejemplo, **para cambiar suma 200, escribe `sum = 200`**. El nuevo valor se aplicará a la variable en el contexto de la ejecución actual.

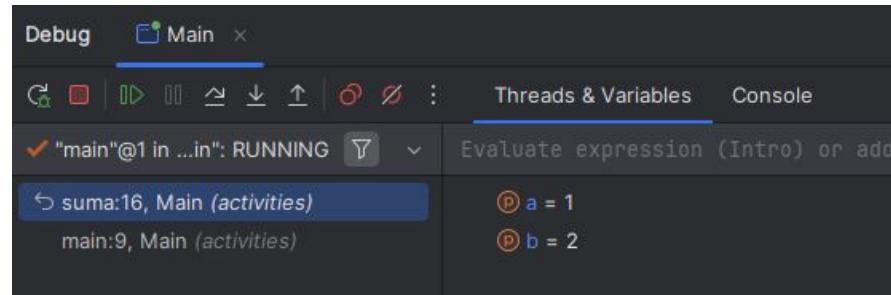
# Pila de llamadas

La pila de llamadas muestra las partes del programa que se están ejecutando actualmente y cómo están relacionadas. Se encuentra en la pestaña **Frames** dentro de la ventana Debugger.

- Contenido de la pila:
  - Lista los hilos en ejecución (como el hilo main en aplicaciones simples).
  - Indica el archivo, método y línea donde se encuentra detenida la ejecución.
  - Muestra las llamadas previas que llevaron al estado actual, permitiendo entender el flujo del programa.
- Interactividad:
  - Al hacer clic en cualquier línea de la pila, el IDE te llevará directamente al archivo y a la línea de código correspondiente.

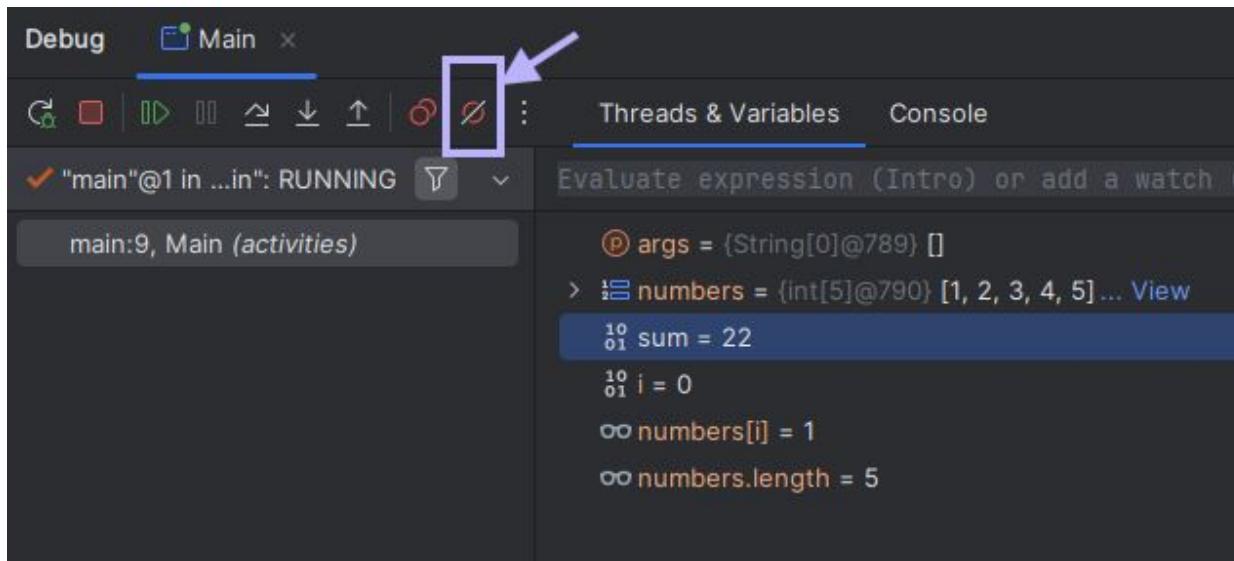
# Ejemplo de pila de llamadas

```
public class Main {  
  
    public static void main(String[] args) {  
        int[] numbers = {1, 2, 3, 4, 5};  
        int sum = 0;  
  
        for (int i = 0; i <= numbers.length; i++) {  
            sum += suma(sum, numbers[i]);  
        }  
  
        System.out.println("La suma de los números es: " + sum);  
    }  
  
    public static int suma(int a, int b) {  
        return a + b;  
    }  
}
```



# Desactivar todos los puntos de interrupción

Puedes desactivar temporalmente todos los breakpoints pulsando el botón **Mute Breakpoints** en la ventana Debugger. Esto permite ejecutar el programa sin interrupciones.



# Tipos de puntos de interrupción

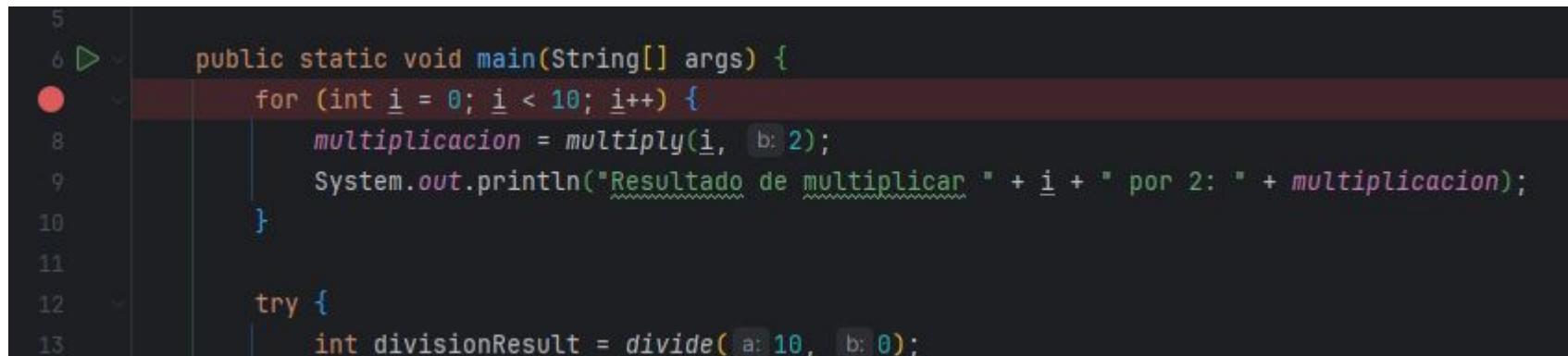
- Puntos de **interrupción de línea**
  - Suspenden el programa **al llegar a la línea de código donde se establecieron.**
- Puntos de **interrupción del método**
  - Suspenden el programa **al entrar o salir de un método especificado.**
- Puntos de **observación de campo** (Field Watchpoints)
  - Suspenden el programa **cuando se lee o modifica un atributo de instancia.**
- Puntos de **interrupción de excepción**
  - Suspenden el programa **cuando se lanza una excepción**, incluso si no está directamente referenciada en el código.

# Ejemplo

```
public class SimpleDebugExample {  
    private static int multiplicacion;  
  
    public static void main(String[] args) {  
        for (int i = 0; i < 10; i++) {  
            multiplicacion = multiply(i, 2);  
            System.out.println("Resultado de multiplicar " + i + " por 2: " +  
multiplicacion);  
        }  
  
        try {  
            int divisionResult = divide(10, 0);  
            System.out.println("Resultado de la división: " + divisionResult);  
        } catch (ArithmeticException e) {  
            System.out.println("Excepción capturada: " + e.getMessage());  
        }  
    }  
  
    public static int multiply(int a, int b) {  
        return a * b;  
    }  
  
    public static int divide(int a, int b) {  
        return a / b;  
    }  
}
```

# Puntos de interrupción de línea

- Se configuran en cualquier línea de código ejecutable y son los más comunes.

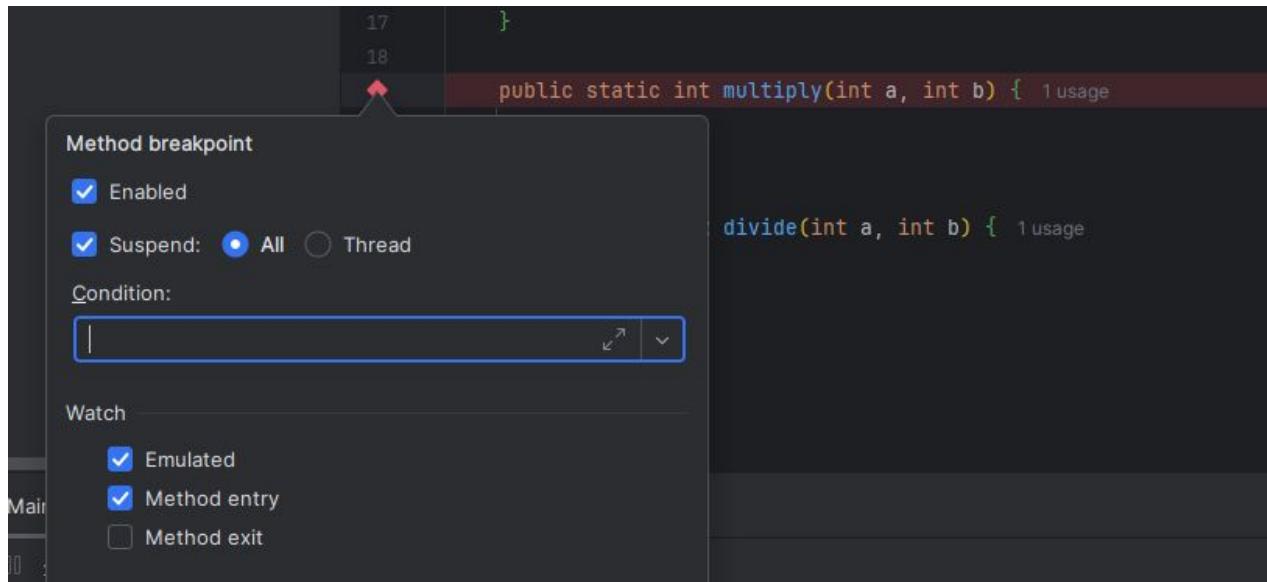


A screenshot of a Java code editor showing a line breakpoint. The code is a simple program that prints the result of multiplying each number from 0 to 10 by 2. A red circular breakpoint icon is positioned at the start of the for loop on line 7. The code is as follows:

```
5  
6 D  
7 ● public static void main(String[] args) {  
8     for (int i = 0; i < 10; i++) {  
9         multiplicacion = multiply(i, b: 2);  
10        System.out.println("Resultado de multiplicar " + i + " por 2: " + multiplicacion);  
11    }  
12    try {  
13        int divisionResult = divide(a: 10, b: 0);  
14    }  
15 }
```

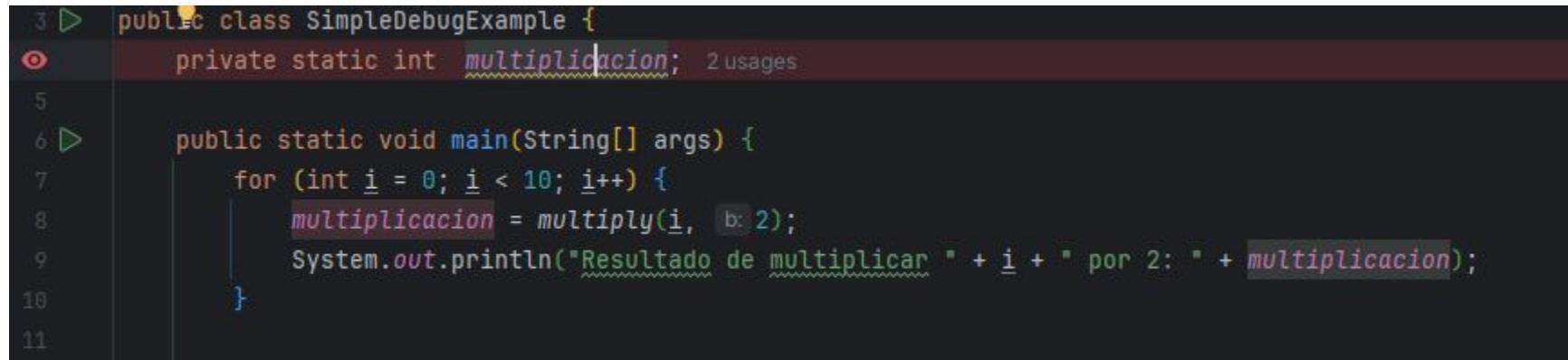
# Puntos de interrupción del método

- Útil para verificar condiciones de entrada/salida en métodos.



# Puntos de observación de campo

- Útil para detectar interacciones con variables de instancia específicas.



```
3 > public class SimpleDebugExample {
4   private static int multiplicacion;  2 usages
5
6 >   public static void main(String[] args) {
7     for (int i = 0; i < 10; i++) {
8       multiplicacion = multiply(i, b: 2);
9       System.out.println("Resultado de multiplicar " + i + " por 2: " + multiplicacion);
10    }
11 }
```

The screenshot shows a Java code editor with the following code:

```
public class SimpleDebugExample {
    private static int multiplicacion;  2 usages

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            multiplicacion = multiply(i, b: 2);
            System.out.println("Resultado de multiplicar " + i + " por 2: " + multiplicacion);
        }
    }
}
```

The variable `multiplicacion` is highlighted in red, indicating it has two usages. The first usage is in the declaration at line 4, and the second is in the assignment statement at line 8.

# Puntos de interrupción de excepción

- Se aplican globalmente a cualquier subclase de Throwable.
- Útil para detectar el origen de excepciones inesperadas.
- Se tiene que activar desde **Run → View Breakpoints** o con el atajo **Ctrl + Shift + F8**.

