

# Diseño de Pruebas, Documentación y Optimización

---

## 1º Desarrollo de Aplicaciones Web

Módulo Entornos de Desarrollo

Docente: Victor Manuel Navarro Camino

Autor: Antonio Cordero Molina

Mayo 2025

## Índice

### · Objetivo

#### A. Diseño y realización de pruebas:

1.
  - 1.1. Identificación de los diferentes tipos de pruebas.
  - 1.2. Definición de casos de prueba.
  - 1.3. Identificación de las herramientas de depuración y prueba de aplicaciones ofrecidas por el entorno de desarrollo.
2.
  - 2.1. Utilización de herramientas de depuración para definir puntos de ruptura y seguimiento.
  - 2.2. Utilización de las herramientas de depuración para examinar y modificar el comportamiento de un programa en tiempo de ejecución.
3. Documentar el plan de pruebas
4.
  - 4.1. Elaboración de pruebas unitarias de clases y funciones.
  - 4.2. Elaboración de pruebas de integración, de sistema y de aceptación.
5.
  - 5.1. Implementación de pruebas automáticas.
  - 5.2. Documentación de las incidencias detectadas.
6.
  - 6.1. Aplicación de normas de calidad a los procedimientos de desarrollo de software.
  - 6.2. Realización de medidas de calidad sobre el software de desarrollo.

#### B. Optimización y documentación:

1.
  - 1.1. Identificar los patrones de refactorización más usuales.
  - 1.2. Elaboración de las pruebas asociadas a la refactorización
2. Revisión del código fuente usando un analizador de código.
3. Identificación de las posibilidades de configuración de un analizador de código.

4. Aplicar patrones de refactorización con las herramientas que proporciona el entorno de desarrollo.
5. Realizar el control de versiones integrado en el entorno de desarrollo.
6. Documentar el código fuente mediante comentarios.
7. Utilizar herramientas del entorno de desarrollo para documentar los procesos, datos y eventos.
8. Utilizar herramientas del entorno de desarrollo para documentar las clases.

**C. Anéxo:**

**D. Webgrafía:**

## Objetivo:

El objetivo principal de esta memoria es profundizar en el diseño y aplicación de pruebas en código, utilizando como caso de estudio una calculadora simple implementada en Java (*Anexo 1 - Código Calculadora*). A través de esta memoria, se busca:

1. Identificar y describir los diferentes tipos de pruebas que se pueden aplicar en el desarrollo de software, como pruebas unitarias, de integración, funcionales y no funcionales.
2. Definir casos de prueba relevantes para la calculadora, considerando escenarios normales y de error, como la división por cero o la entrada de datos no numéricos por ejemplo.
3. Explorar herramientas de depuración disponibles en entornos de desarrollo populares como Eclipse, IntelliJ IDEA y Visual Studio Code, destacando sus características y utilidades para el seguimiento y modificación del comportamiento del programa en tiempo de ejecución.
4. Demostrar de forma práctica cómo utilizar estas herramientas para establecer puntos de ruptura, inspeccionar variables y modificar el comportamiento del programa durante la ejecución, lo que facilita la identificación y resolución de errores.
5. Documentar y comparar las herramientas y técnicas de depuración en diferentes entornos de desarrollo, resaltando sus ventajas y limitaciones.

Con esta memoria, se busca proporcionar un enfoque estructurado y práctico para la implementación de pruebas y depuración en el desarrollo de software, facilitando la identificación y corrección de errores para mejorar la calidad del código.

## A. Diseño y Realización de Pruebas

### 1.

#### 1.1. Identificación de los diferentes tipos de pruebas.

En el desarrollo de software, las pruebas se utilizan para garantizar que el código funcione como se espera. Existen diferentes tipos de pruebas que pueden aplicarse a una aplicación, entre ellas:

- **Pruebas Unitarias.** Estas pruebas se centran en verificar el funcionamiento correcto de unidades individuales de código, como funciones o métodos. Su objetivo es asegurar de que cada componente individual opera según lo esperado sin depender de otros módulos. En el caso de la calculadora se deben de realizar pruebas unitarias para comprobar que los métodos de suma, resta, multiplicación y división devuelven los resultados correctos para diferentes entradas. Por ejemplo, se incluirán casos normales (sumar 3+5, dividir 10/2) y casos especiales, como dividir entre cero, para comprobar si el código maneja errores adecuadamente.

*(Ejemplo: Anexo 1.1.1: Prueba Unitaria)*

- **Pruebas de Integración.** Las pruebas de integración verifican que diferentes módulos o componentes del sistema funcionen correctamente cuando se combinan. Se centran en la interacción entre distintas partes del código. Aplicándolo a la calculadora, se prueba cómo la entrada de datos, en este caso, números ingresados por el usuario en la terminal, se transmiten a los métodos que se elijan de forma correcta. Se verificaría finalmente que la integración entre la interfaz de usuario y las funciones de la calculadora no genere errores, para ello se han de probar combinaciones de operaciones.

*(Ejemplo: Anexo 1.1.2: Prueba de Integración)*

- **Pruebas Funcionales.** Las pruebas funcionales evalúan si el software cumple con los requisitos definidos. Se centran en la funcionalidad del sistema sin preocuparse por la implementación interna. Llevándolo al caso de la calculadora, se comprobará que la calculadora realiza correctamente las operaciones matemáticas básicas. Es por esto que se probarán diferentes escenarios de entrada para validar que el programa se comporta de manera adecuada ante distintos valores. Se verificará que, cuando el usuario introduce datos no válidos, el programa responde con un mensaje de error y no se bloquea.

*(Ejemplo: Anexo 1.1.3: Prueba Funcional)*

- **Pruebas de Usabilidad.** Se enfocan en la experiencia del usuario, es decir, evalúan la facilidad o comodidad de uso y la experiencia de uso del usuario con la aplicación. En este caso, se analizaría si la interacción con el usuario es clara y sencilla., comprobar si los mensajes son comprensibles, directos y concisos. Finalmente, se

debe evaluar la gestión de errores, en caso que tenga una entrada incorrecta, cómo lo indica el programa.

*(Ejemplo: Anexo 1.1.4: Prueba Usabilidad)*

- **Pruebas de Rendimiento.** Las pruebas de rendimiento evalúan la velocidad, estabilidad y eficiencia del sistema bajo diferentes condiciones de carga. Son más relevantes en aplicaciones de gran escala, pero pueden aplicarse a cualquier software. En la calculadora, se medirá el tiempo que tarda la calculadora en procesar múltiples operaciones seguidas, el cual lo indica la terminal del IDE, *IntelliJ Idea* en este caso. Debe probarse con una cantidad masiva de cálculos para detectar posibles ralentizaciones. Finalmente, analizar el consumo de recursos del programa en ejecución.

*(Ejemplo: Anexo 1.1.5: Prueba Rendimiento)*

## 1.2. Definición de casos de prueba.

Los casos de prueba son escenarios diseñados para verificar si una aplicación funciona correctamente en diversas situaciones. En el caso de la calculadora creada, estos casos aseguran que los cálculos se realizan correctamente y que el programa maneja errores adecuadamente.

Para la aplicación actual de la calculadora se pueden utilizar varios:

**1.2.1.** Suma de dos números positivos. El objetivo es verificar que la calculadora suma correctamente dos números positivos. Con una entrada de  $2 + 3$  y se debe esperar como salida el resultado 5.

*(Ejemplo: Anexo 1.2.1: Caso Prueba 1: suma)*

**1.2.2.** Resta de dos números con el objetivo de obtener un resultado negativo. El objetivo es comprobar que la calculadora realmente maneja la operación de resta cuando se espera un resultado negativo. Se prueba la entrada de dos números enteros  $3 - 5$ , esperando como salida el -2 negativo.

*(Ejemplo: Anexo 1.2.2: Caso Prueba 2: resta con salida negativa)*

**1.2.3.** Multiplicación por cero, el objetivo es ver que cualquier número multiplicado por cero se obtiene una salida de cero

*(Ejemplo: Anexo 1.2.3: Caso Prueba 3: multiplicación por cero)*

**1.2.4.** División de cualquier número por cero, aquí se de ha de tener en cuenta que cualquier número entre cero no se puede operar. Se obtiene un resultado de *Infinity* pero es más correcto controlar esto como error. En este caso se debería añadir al código inicial de

la calculadora un control de excepciones del tipo *ArithmeticException* al método de la división en caso de que el divisor introducido sea cero.

(Ejemplo: Anexo 1.2.4: Caso Prueba 4: división por cero - error)

**1.2.5.** Con una entrada por teclado de datos no numéricos, se debe validar que el sistema maneja correctamente la entrada de caracteres no numéricos y evitar errores derivados de ello. Por ejemplo con una entrada de cualquier carácter tipo char, por ejemplo "a" + 3. La salida debe ser un mensaje de error controlando dicho error con una excepción del tipo *InputMismatchException*

(Ejemplo: Anexo 1.2.5: Caso Prueba 5: entrada de datos no numéricos)

### 1.3. Identificación de las herramientas de depuración y prueba de aplicaciones ofrecidas por el entorno de desarrollo.

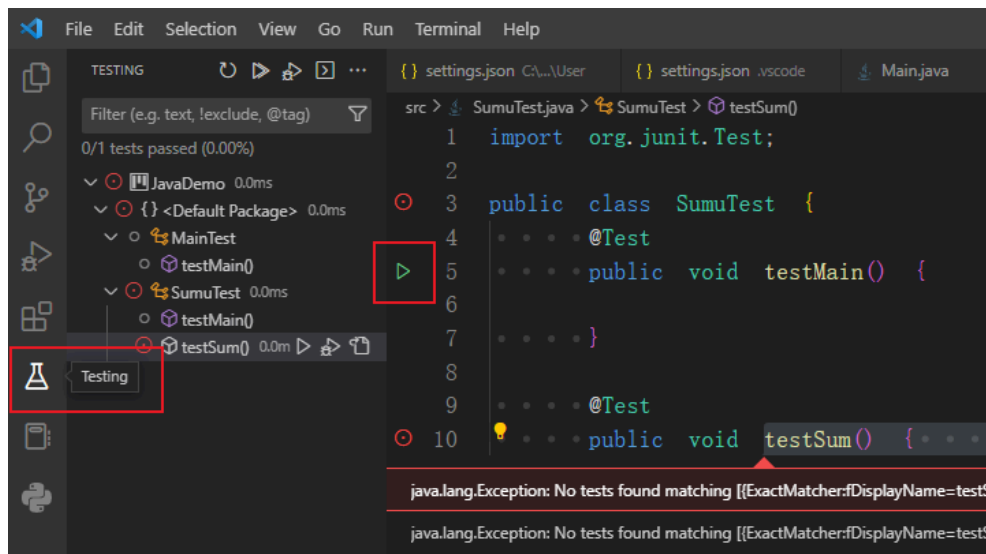
Los entornos de desarrollo integrados (IDE) ofrecen diversas herramientas para la depuración y prueba de aplicaciones, lo que facilita la identificación y corrección de errores en el código. En los entornos de desarrollo que utilizamos se presentan varias herramientas para la depuración de código, entre ellas:

- **IntelliJ IDEA:** IntelliJ IDEA es un IDE popular para el desarrollo en Java y otros lenguajes, conocido por su potente sistema de análisis de código y herramientas de depuración, cuenta con:
  - **Debugger avanzado:** Ofrece funcionalidades como evaluación de expresiones en tiempo de ejecución, visualización de estructuras de datos y opciones avanzadas de inspección.
  - **Integración con JUnit y TestNG:** Facilita la creación y ejecución de pruebas unitarias y pruebas funcionales.
  - **Soporte para pruebas de rendimiento:** Mediante herramientas como YourKit, permite analizar el consumo de recursos y la eficiencia del código.

Call Tree	Time (ms)	Avg. Time (ms)	Count
<All threads>	198,671	100 %	
java.lang.Thread.run()	172,946	87 %	3,603
Thread.java:748 org.apache.catalina.core.ContainerBase\$ContainerBackgroundProcessor.run()	122,380	62 %	40,793
ContainerBase.java:1637 java.lang.Thread.sleep(long)	122,376	62 %	13,597
ContainerBase.java:1648 org.apache.catalina.core.ContainerBase\$ContainerBackgroundProcess	3	0 %	0.4
ContainerBase.java:1644 java.lang.Thread.getContextClassLoader()	0.4	0 %	0.2
ContainerBase.java:1645 org.apache.catalina.core.ContainerBase.getLoader()	0.2	0 %	< 0.1
ContainerBase.java:1646 org.apache.catalina.core.ContainerBase.getLoader()	0	0 %	0
Thread.java:748 org.glassfish.grizzly.threadpool.AbstractThreadPool\$Worker.run()	25,748	13 %	1,355
AbstractThreadPool.java:549 org.glassfish.grizzly.threadpool.AbstractThreadPool\$Worker.do	25,748	13 %	1,355
WorkerThreadIOStrategy.java:114 org.glassfish.grizzly.threadpool.AbstractThreadPool\$Worker.do	25,605	13 %	68
WorkerThreadIOStrategy.java:33 org.glassfish.grizzly.threadpool.AbstractThreadPool\$Worker.do	25,605	13 %	68
WorkerThreadIOStrategy.java:94 org.glassfish.grizzly.threadpool.AbstractThreadPool\$Worker.do	25,605	13 %	68
AbstractIOStrategy.java:89 org.glassfish.grizzly.threadpool.AbstractThreadPool\$Worker.do	25,605	13 %	68
TCPNIOTransport.java:515 org.glassfish.grizzly.threadpool.AbstractThreadPool\$Worker.do	25,598	13 %	68
TCPNIOTransport.java:516 org.glassfish.grizzly.threadpool.AbstractThreadPool\$Worker.do	6	0 %	< 0.1
TCPNIOTransport.java:518 org.glassfish.grizzly.threadpool.AbstractThreadPool\$Worker.do	0.7	0 %	< 0.1

(CPU Profiling para recursos de cpu usando YourKit en IntelliJ IDEA)

- **Visual Studio Code:** Visual Studio Code es un editor de código ligero pero altamente extendido mediante extensiones, lo que lo convierte en una opción flexible para la depuración y prueba de aplicaciones.
  - Extensión Java Debugger: Permite depurar aplicaciones Java directamente en VS Code, estableciendo breakpoints y viendo valores de variables.
  - Soporte para frameworks de pruebas como Jest, Mocha y JUnit: Permite probar aplicaciones en distintos lenguajes con facilidad.
  - Extensión Java Test Runner: Facilita la ejecución de pruebas unitarias para proyectos en Java.



(Uso de la extensión Java Test Runner en VSCode)

La depuración y las pruebas son procesos esenciales en el desarrollo de software para garantizar la calidad del código, es por esto que cada IDE ofrece herramientas específicas en general o muy específicas para lenguajes concretos que permiten mejorar la eficiencia del desarrollo, detectar errores con mayor facilidad y asegurar que el código funcione correctamente antes de ser implementado en producción.



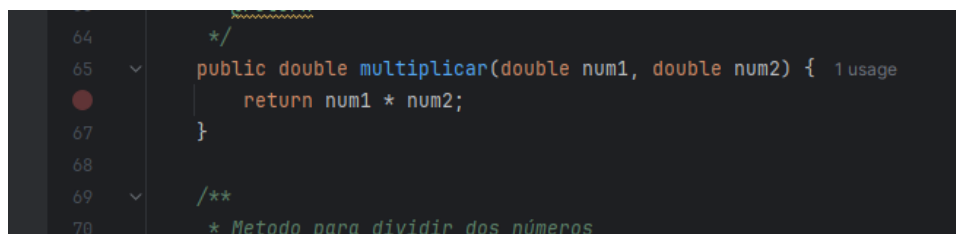
## 2.

### 2.1. Utilización de herramientas de depuración para definir puntos de ruptura y seguimiento.

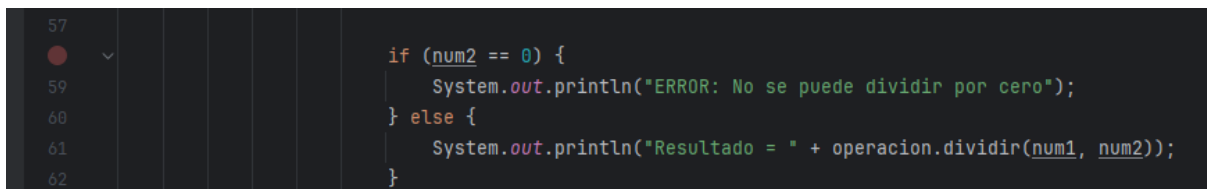
Los puntos de ruptura (breakpoints) permiten detener la ejecución del programa en una línea específica del código para inspeccionar su estado y comportamiento en tiempo de ejecución. Esto es esencial para analizar el flujo del programa, identificar errores y verificar el estado de las variables. Tanto en IntelliJ IDEA como en VSCode se puede acceder a ellos y usarlos en el código haciendo click en el margen izquierdo de la numeración de líneas del código.

Tipos de breakpoints:

1. **Breakpoints simples:** Detienen la ejecución en una línea específica.



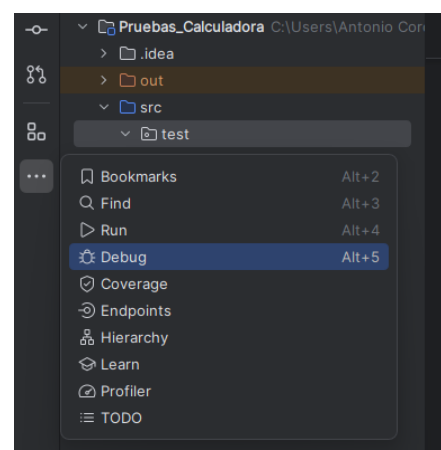
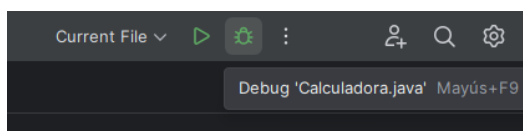
2. **Breakpoints condicionales:** Solo detienen la ejecución si se cumple una condición definida por el usuario (ejemplo: `num1 == 0`).



3. **Breakpoints de excepciones:** Se activan cuando se lanza una excepción en el programa.
4. **Breakpoints de método:** Se activan cuando se llama a un método específico.

En cambio, dentro del seguimiento del programa:

- **Ejecutar en modo depuración:** Iniciar la aplicación en modo *Debug* para que los breakpoints funcionen.



- **Ejecución paso a paso** (seleccionando previamente puntos de ruptura donde sea crean necesarios):
  - *Step Into (F5)*: Entra en el método llamado en la línea actual.
  - *Step Over (F6)*: Ejecuta la línea actual y avanza sin entrar en métodos.
  - *Step Out (F7)*: Sale del método actual y regresa al llamador.

*(Anexo 2.1 - Ejemplo de depuración en ejecución paso a paso)*

Usando el método de división por cero inicial en el código:

- Marcar los puntos de ruptura donde se crean necesario para ver los datos cómo cambian en tiempo real durante la ejecución paso a paso de forma manual del código: *(Anexo 2.1 - imagen 1)*
- Pulsando en el icono de debug anteriormente mencionado, se puede ver cómo nos da los valores en tiempo real a medida que avanza la ejecución del código: *(Anexo 2.1 - imagen 2)*
- Esto nos muestra una nueva ventana donde se encuentran los botones anteriores y además, los valores seleccionados en los puntos de ruptura donde se quiere controlar el paso del código: *(Anexo 2.1 - imagen 3)* y *(Anexo 2.1 - imagen 4)*
- Por otra parte, cuando el código llega a la línea de código donde pide al usuario introducir datos por teclado, deja disponible la consola: *(Anexo 2.1 - imagen 5)*
- En la ventana anterior de inspección de valores de las variables, recoger el nuevo dato introducido, en este caso cero, por lo que en la condición de la estructura condicional del if llega al mensaje de error controlado, terminando así en el siguiente step over, el procedimiento de la operación tras mostrar por pantalla el mensaje de error al dividir por cero. *(Anexo 2.1 - imagen 6)*, *(Anexo 2.1 - imagen 7)* y *(Anexo 2.1 - imagen 8)*.

## 2.2. Utilización de las herramientas de depuración para examinar y modificar el comportamiento de un programa en tiempo de ejecución.

Las herramientas de depuración permiten analizar el comportamiento de un programa en ejecución y modificar su estado sin detener la aplicación. Algunas funcionalidades clave incluyen:

1. **Inspección de variables:** Permite visualizar los valores almacenados en memoria.
2. **Modificación de variables:** Cambiar valores en tiempo real para probar distintos escenarios.
3. **Watch Expressions:** Permiten monitorear valores específicos sin necesidad de breakpoints.
4. **Depuración condicional:** Ejecución pausada basada en condiciones lógicas.
5. **HotSwap (en IntelliJ IDEA):** Permite modificar código sin reiniciar la aplicación.

### Ejemplo 1: Seguimiento del flujo de entrada de datos

#### Código a depurar:

(Anexo: 2.2 - Ejemplo 1 - Imagen 1)

#### Pasos:

1. Establecer un breakpoint en la línea `num2 = sc.nextDouble();`.  
a. (Anexo 2.2 - Ejemplo 1 - Imagen 1)
2. Configurar la condición del breakpoint: `num1 == 0`.  
a. (Anexo 2.2 - Ejemplo 1 - Imagen 2)
3. Ejecutar el programa en modo Debug.  
a. (Anexo 2.2 - Ejemplo 1 - Imagen 3)
4. Cuando `num1` sea 0, el programa se detendrá automáticamente.  
a. (Anexo 2.2 - Ejemplo 1 - Imagen 4)
5. Usar `Alt + F8` para evaluar expresiones y realizar pruebas.  
a. a (Anexo: 2.2 - Ejemplo 1 - Imagen 5 y 6):

### Ejemplo 2: Modificación de variables durante la ejecución

#### Pasos:

1. Establecer un breakpoint en el método `multiplicar`.  
a. (Anexo 2.2 - Ejemplo 2 - Imagen 1)
2. Ejecutar el programa en modo Debug.
3. Cuando el programa se detenga, ir a la pestaña Run and Debug.
4. Modificar el valor de `a` en la sección Variables.  
a. (Anexo 2.2 - Ejemplo 2 - Imagen 2 y 3):
5. Continuar la ejecución para ver los efectos del cambio.  
a. (Anexo 2.2 - Ejemplo 2 - Imagen 4):

Normalmente, los breakpoint condicionales se usan mayormente para errores intermitentes, por ejemplo `num2 == 0`, las watch expressions para monitorear variables críticas y el modo “hot swap” para recargar cambios de código sin reiniciar la depuración.

### 3. Anexo:

- **Anexo 1:** Código utilizado para la creación de la Calculadora sencilla:

La clase main con el funcionamiento principal, pidiendo por pantalla, primero la operación que desea realizar, luego pidiendo los dos números a introducir por el usuario y finalmente dando el resultado. De forma inicial este código de prueba contempla solamente que el usuario opere con dos números, así como también controla que el usuario no introduzca por error una opción errónea.

```
import java.util.Scanner;

public class Calculadora {
    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        Operaciones operacion = new Operaciones();
        int opcion;
        double num1;
        double num2;

        do {
            System.out.println("\nSeleccione una opción:");
            System.out.println("1 - Sumar\n2 - Restar\n3 - Multiplicar\n4 - Dividir\n5 - Salir");
            System.out.print("\t-> ");
            opcion = sc.nextInt();

            if (opcion < 1 || opcion > 5) {
                System.out.println("\nOpcion no valida - Seleccione una opcion correcta");
            } else {

                switch (opcion) {
                    //Suma
                    case 1:
                        System.out.print("Numerol: ");
                        num1 = sc.nextDouble();
                        System.out.print("Numero2: ");
                        num2 = sc.nextDouble();
                        System.out.println("Resultado = " + operacion.sumar(num1, num2));

                        break;

                    //Resta
                    case 2:
                        System.out.print("Numerol: ");
                        num1 = sc.nextDouble();
                        System.out.print("Numero2: ");
                        num2 = sc.nextDouble();
                        System.out.println("Resultado = " + operacion.restar(num1, num2));

                        break;

                    //Multiplicación
```

```

        case 3:
            System.out.print("Numerol: ");
            num1 = sc.nextDouble();
            System.out.print("Numero2: ");
            num2 = sc.nextDouble();
            System.out.println("Resultado = " + operacion.multiplicar(num1,
num2));

            break;

        //División
        case 4:
            System.out.print("Numerol: ");
            num1 = sc.nextDouble();
            System.out.print("Numero2: ");
            num2 = sc.nextDouble();

            if (num2 == 0) {
                System.out.println("ERROR: No se puede dividir por cero");
            } else {
                System.out.println("Resultado = " + operacion.dividir(num1,
num2));
            }
            break;

        //Fin
        case 5:
            System.out.println("\nSaliendo...\n");
            break;
    }
}
} while (opcion != 5);

sc.close();
}
}

```

La clase Operaciones, la cual incluye los métodos que utiliza la clase main para no saturar en exceso el código principal de la aplicación. Cada método recoge por parámetros los números introducidos por el usuario desde la clase main:

```

public class Operaciones {

    private double num1;
    private double num2;

    //Constructor
    public Operaciones() {

    }

    //Getters - Setters
    public double getNum1() {
        return num1;
    }

    public void setNum1(double num1) {
        this.num1 = num1;
    }
}

```

```
}

public double getNum2() {
    return num2;
}

public void setNum2(double num2) {
    this.num2 = num2;
}

//ToString
@Override
public String toString() {
    return "Operaciones{" +
        "num1=" + num1 +
        ", num2=" + num2 +
        '}';
}

//Métodos - Operaciones

/**
 * Metodo para sumar dos números
 * @param num1
 * @param num2
 * @return
 */
public double sumar(double num1, double num2) {
    return num1 + num2;
}

/**
 * Metodo para restar dos números
 * @param num1
 * @param num2
 * @return
 */
public double restar(double num1, double num2) {
    return num1 - num2;
}

/**
 * Metodo para multiplicar dos números
 * @param num1
 * @param num2
 * @return
 */
public double multiplicar(double num1, double num2) {
    return num1 * num2;
}

/**
 * Metodo para dividir dos números
 * @param num1
 * @param num2
 * @return
 */
public double dividir(double num1, double num2) {
    return num1 / num2;
}
```

```
}}
```

- **Anexo 1.1.1: Prueba Unitaria:**

```
@Test
public void testSuma() {
    Calculadora calc = new Calculadora();
    assertEquals(8, calc.sumar(5, 3));
}
```

- **Anexo 1.1.2: Prueba de Integración:**

```
@Test
public void testEntradaUsuario() {
    String input = "5\n3\nsuma";
    System.setIn(new ByteArrayInputStream(input.getBytes()));
    Scanner scanner = new Scanner(System.in);
    int num1 = scanner.nextInt();
    int num2 = scanner.nextInt();
    String operacion = scanner.next();

    assertEquals(5, num1);
    assertEquals(3, num2);
    assertEquals("suma", operacion);
}
```

- **Anexo 1.1.3: Prueba Funcional**

```
@Test
public void testOperacionInvalida() {
    Calculadora calc = new Calculadora();
    assertThrows(IllegalArgumentException.class, () -> {
        calc.operar(4, 2, "potencia");
    });
}
```

- **Anexo 1.1.4: Prueba de Usabilidad**

```
Seleccione una opción:
1 - Sumar
2 - Restar
3 - Multiplicar
4 - Dividir
5 - Salir
```



```
-> 4
Numero1: 10
Numero2: 0
ERROR: No se puede dividir por cero

Seleccione una opción:
1 - Sumar
2 - Restar
3 - Multiplicar
4 - Dividir
5 - Salir
    -> 5

Saliendo...

Process finished
```

#### - Anexo 1.1.5: Prueba de Rendimiento

```
@Test
public void testTiempoEjecucion() {
    Calculadora calc = new Calculadora();
    long inicio = System.nanoTime();
    for (int i = 0; i < 1000000; i++) {
        calc.sumar(i, i+1);
    }
    long fin = System.nanoTime();
    long tiempoEjecucion = fin - inicio;
    assertTrue(tiempoEjecucion < 500000000); //0.5 segundos
}
```

#### - Anexo 1.2.1: Caso Prueba 1: suma

```
@Test
void testSumaPositivos() {
    assertEquals(5, operaciones.sumar(2, 3), "La suma de 2 + 3
debe ser 5");
}
```

#### - Anexo 1.2.2: Caso Prueba 2: resta con salida negativa

```
@Test
void testRestaNegativa() {
```

```
    assertEquals(-2, operaciones.restar(3, 5), "La resta de 3 - 5  
debe ser -2");  
}
```

- **Anexo 1.2.3:** Caso Prueba 3: multiplicación por cero

```
@Test  
void testMultiplicacionPorCero() {  
    assertEquals(0, operaciones.multiplicar(7, 0), "La  
multiplicación de 7 * 0 debe ser 0");  
}
```

- **Anexo 1.2.4:** Caso Prueba 4: división por cero - error

```
@Test  
void testDivisionPorCero() {  
    assertThrows(ArithmeticException.class, () ->  
operaciones.dividir(5, 0), "Dividir por cero debe lanzar una  
excepción");  
}  
  
/**  
 * El método debería quedar así  
 * controlando el error con una excepción  
 */  
  
public double dividir(double num1, double num2) {  
    if (num2 == 0) {  
        throw new ArithmeticException("No se puede dividir por  
cero");  
    }  
    return num1 / num2;  
}
```

- **Anexo 1.2.5:** Caso Prueba 5: entrada de datos no numéricos

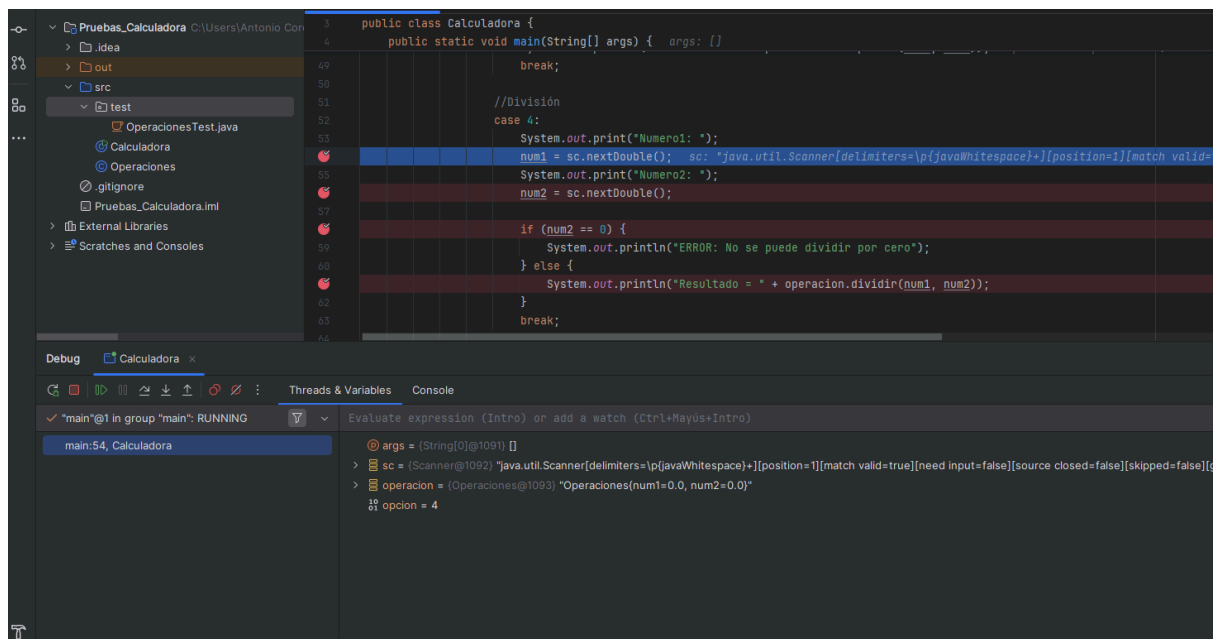
```
try {  
    System.out.print("Numerol: ");  
    num1 = sc.nextDouble();  
} catch (InputMismatchException e) {  
    System.out.println("Error: Entrada no válida. Debe ingresar un  
número.");  
    sc.next(); // Limpia el buffer del Scanner  
    continue; // Reinicia el bucle  
}
```

- **Anexo 2.1:** Ejemplo de depuración en ejecución paso a paso

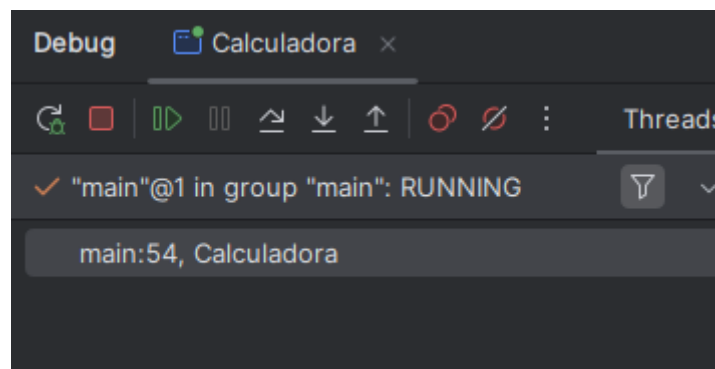
- (Anexo 2.1 - imagen 1):

```
50
51 //División
52 case 4:
53     System.out.print("Numero1: ");
54     num1 = sc.nextDouble();
55     System.out.print("Numero2: ");
56     num2 = sc.nextDouble();
57
58     if (num2 == 0) {
59         System.out.println("ERROR: No se puede dividir por cero");
60     } else {
61         System.out.println("Resultado = " + operacion.dividir(num1, num2));
62     }
63     break;
64
```

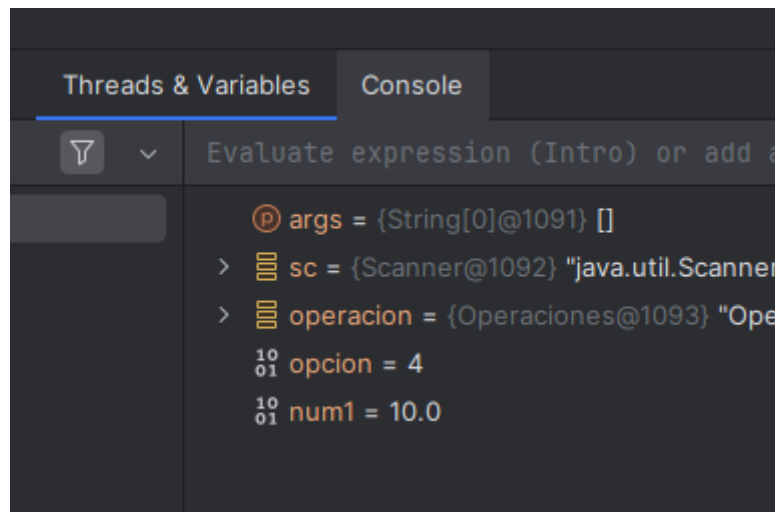
- (Anexo 2.1 - imagen 2):



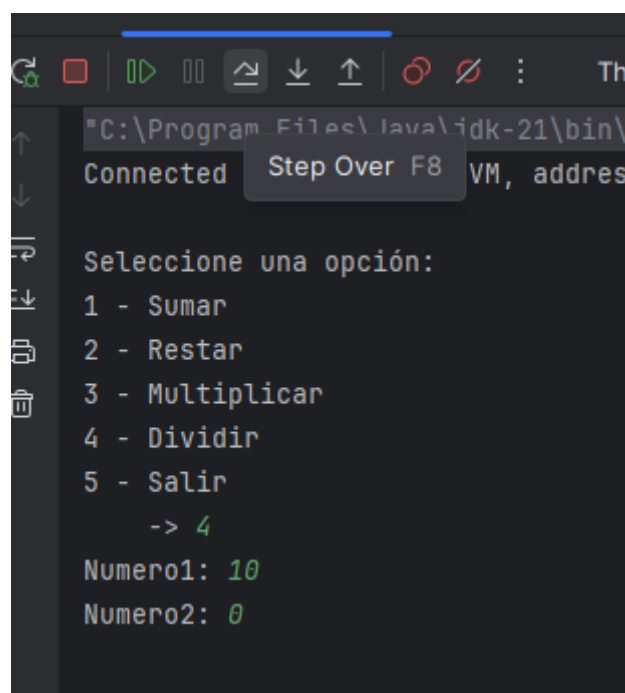
- (Anexo 2.1- imagen 3):



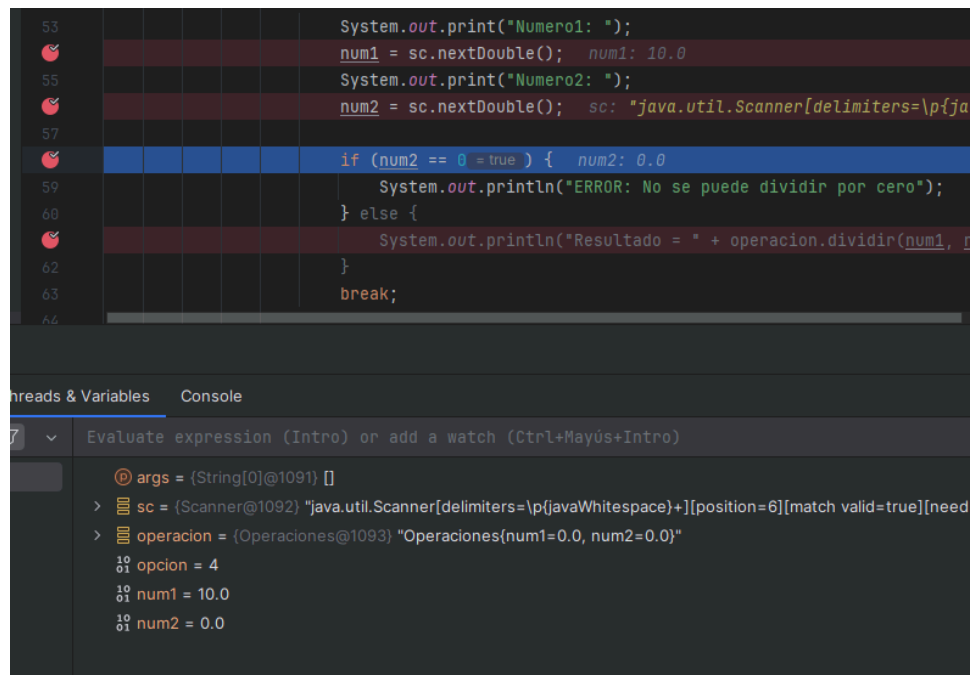
- (Anexo 2.1 - imagen 4):



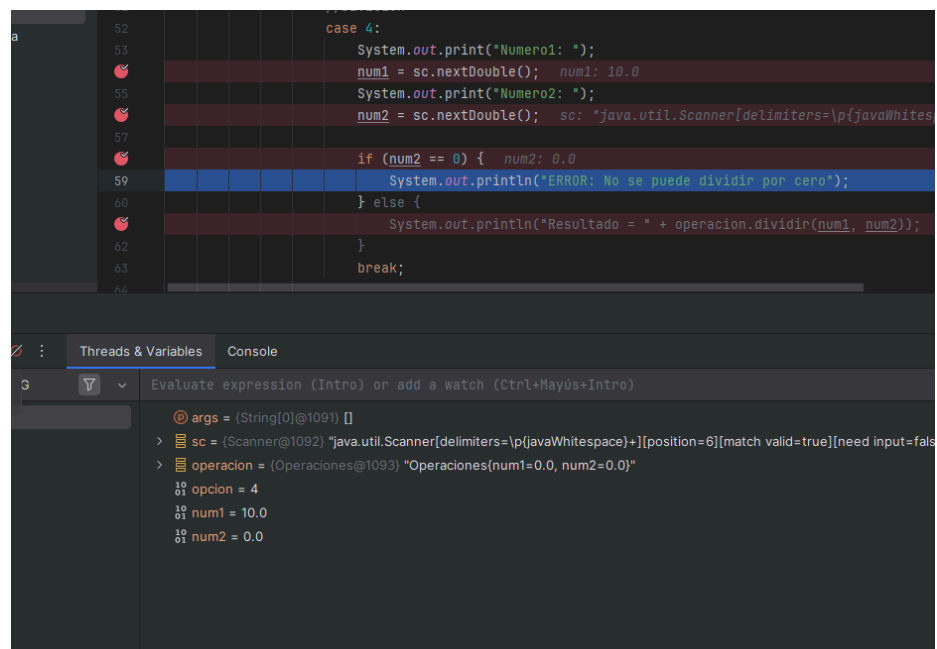
- (Anexo 2.1 - imagen 5):



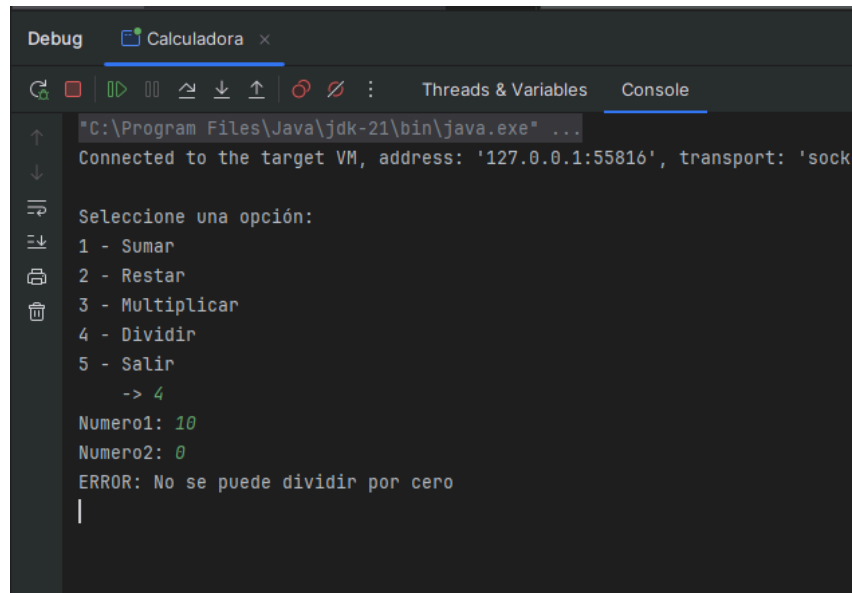
- (Anexo 2.1 - imagen 6):



- (Anexo 2.1 - imagen 7):

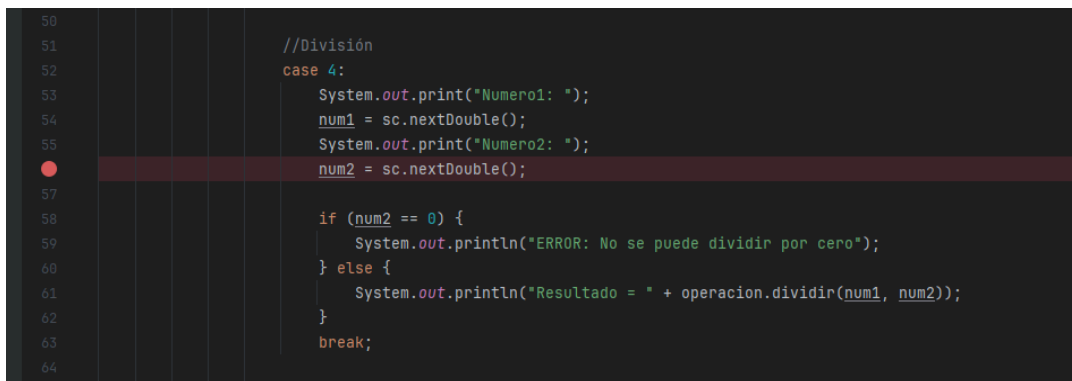


- (Anexo 2.1 - imagen 8):



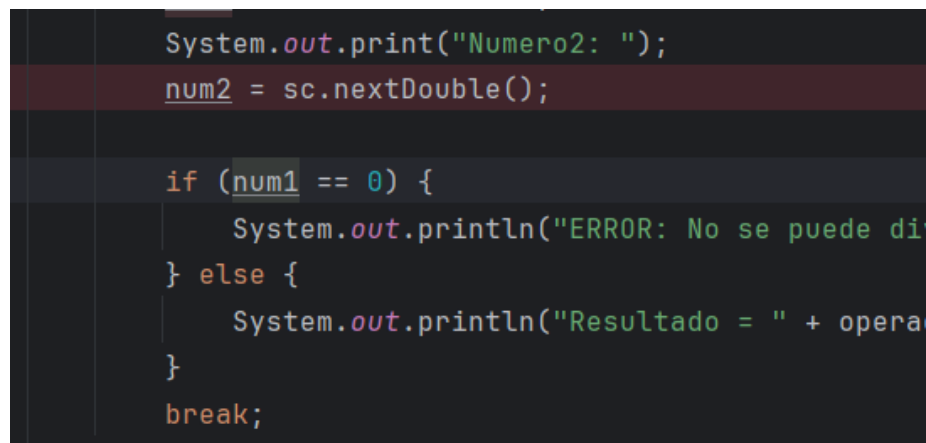
```
Debug  Calculadora x
C:\Program Files\Java\jdk-21\bin\java.exe" ...
Connected to the target VM, address: '127.0.0.1:55816', transport: 'sock
Seleccione una opción:
1 - Sumar
2 - Restar
3 - Multiplicar
4 - Dividir
5 - Salir
-> 4
Numero1: 10
Numero2: 0
ERROR: No se puede dividir por cero
|
```

- (Anexo: 2.2 - Ejemplo 1 - Imagen 1):



```
50
51 //División
52 case 4:
53     System.out.print("Numero1: ");
54     num1 = sc.nextDouble();
55     System.out.print("Numero2: ");
56     num2 = sc.nextDouble();
57
58     if (num2 == 0) {
59         System.out.println("ERROR: No se puede dividir por cero");
60     } else {
61         System.out.println("Resultado = " + operacion.dividir(num1, num2));
62     }
63     break;
64
```

- (Anexo: 2.2 - Ejemplo 1 - Imagen 2):



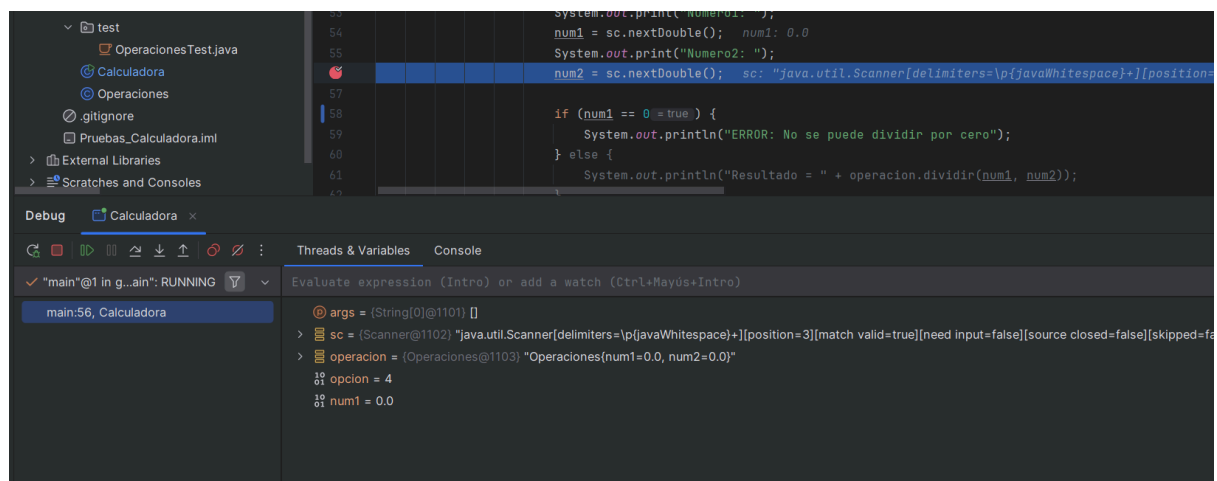
```
System.out.print("Numero2: ");
num2 = sc.nextDouble();

if (num1 == 0) {
    System.out.println("ERROR: No se puede di
} else {
    System.out.println("Resultado = " + opera
}
break;
```

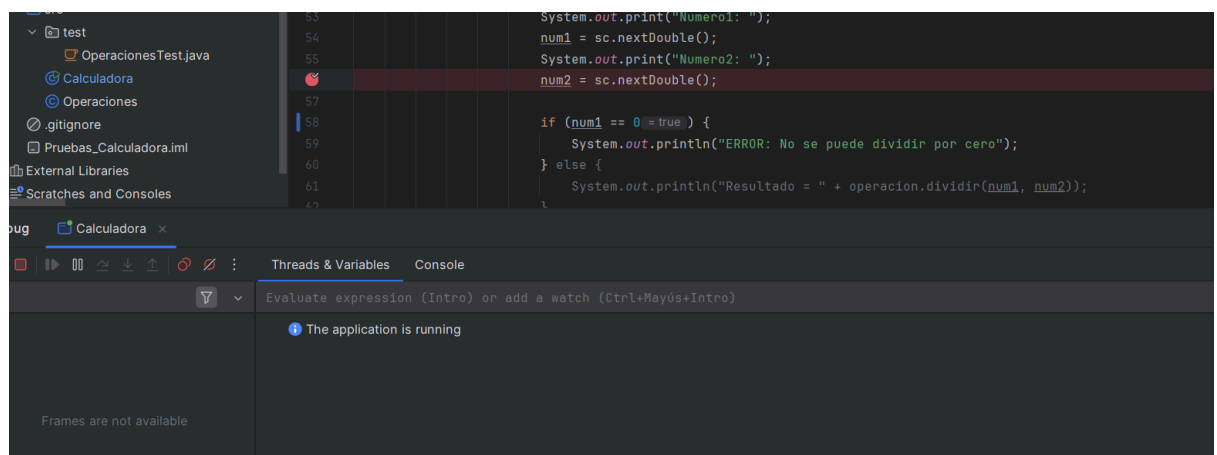
- (Anexo: 2.2 - Ejemplo 1 - Imagen 3):

```
Seleccione una opción:
1 - Sumar
2 - Restar
3 - Multiplicar
4 - Dividir
5 - Salir
    -> 4
Numero1: 0|
```

- (Anexo: 2.2 - Ejemplo 1 - Imagen 4):



- (Anexo: 2.2 - Ejemplo 1 - Imagen 5):



- (Anexo: 2.2 - Ejemplo 1 - Imagen 6):

```
Seleccione una opción:
1 - Sumar
2 - Restar
3 - Multiplicar
4 - Dividir
5 - Salir
-> 4

Numero1: 0
Numero2: 10
ERROR: No se puede dividir por cero
```

- (Anexo 2.2 - Ejemplo 2 - Imagen 1):

```
64      */
65      public double multiplicar(double num1, double num2) { 1 usage      num1: 5.0      num2: 2.0
66          return num1 * num2; num1: 5.0      num2: 2.0
67      }
68
69      /**
70       * Metodo para dividir dos números
71       * @param num1
72       * @param num2
73       * @return
```

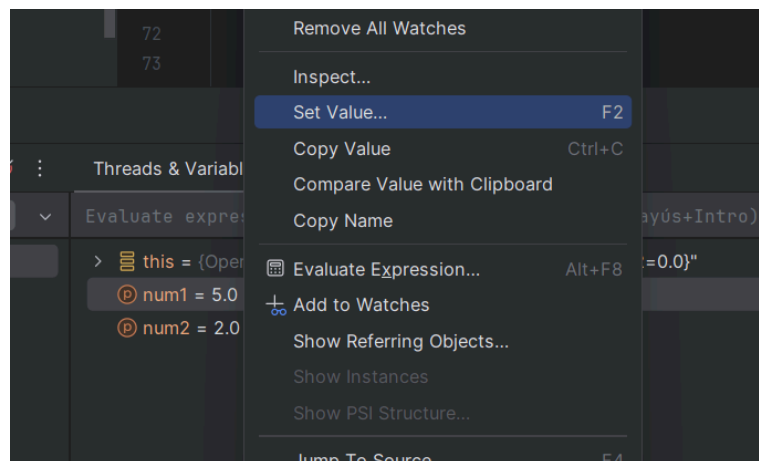
Threads & Variables Console

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

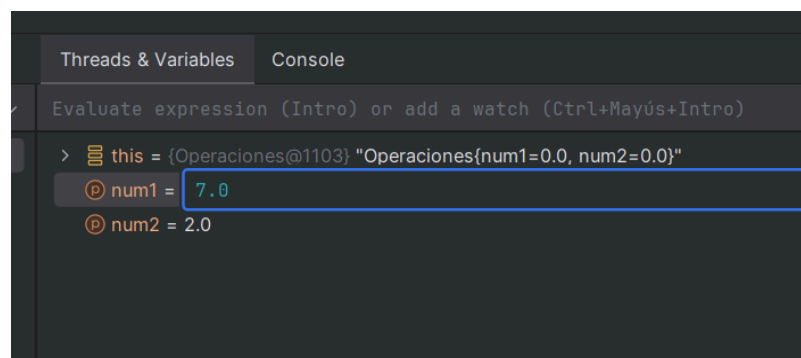
```
> this = {Operaciones@1103} "Operaciones(num1=0.0, num2=0.0)"
num1 = 5.0
num2 = 2.0
```



- (Anexo 2.2 - Ejemplo 2 - Imagen 2):



- (Anexo 2.2 - Ejemplo 2 - Imagen 3):



- (Anexo 2.2 - Ejemplo 2 - Imagen 4):

