
Práctica 5. Herramientas de Depuración.

Objetivo:

- Conocer y dominar, paso a paso y con ejemplos prácticos, las herramientas de depuración que ofrece el entorno de Eclipse para la programación en Java.

La depuración, o debugging, es la habilidad de localizar y corregir los errores de un programa. Se trata de una competencia básica para un ingeniero en informática, ya que debe ser capaz de identificar, comprender y resolver los problemas que se presentan cuando el software no se comporta de la manera esperada. Sin los conocimientos y herramientas adecuadas, la depuración puede convertirse en una tarea tediosa que consumirá una parte importante del tiempo de desarrollo, sobre todo en aplicaciones de gran complejidad y tamaño.

Debido a esto, los entornos de desarrollo (como Eclipse), incorporan herramientas potentes de depuración. Aprender a usar estas herramientas no solo acelera el trabajo de desarrollo, sino que también aporta habilidades para analizar y comprender los sistemas complejos.

Contenido de la sesión

Esta práctica usa proyectos con dos ejemplos sencillos: SumDemo, y BankTransfer (multihilo). Los ejemplos contienen errores intencionales para su localización y corrección, utilizando las herramientas de Eclipse.

Preparación del entorno

Para esta sesión de prácticas se facilitan los dos proyectos de Eclipse: debugDemo y concurrentDebugDemo. El primero de ellos (debugDemo) se utilizará durante la primera parte de la sesión, y el segundo para explicar las herramientas de depuración con aplicaciones concurrentes (varios hilos de ejecución). El código fuente de ambos proyectos está disponible en el anexo de este documento.

Antes de continuar, se debe importar el primero de los proyectos en Eclipse.

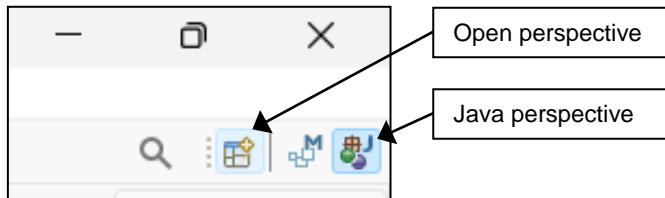
Contenido

1. Conceptos básicos de depuración	3
1.1 Abrir la perspectiva de depuración (Debug Perspective)	3
1.2 Poner un breakpoint y ejecutar en modo debug.....	3
1.3 Pasos (Stepping) – navegar en el código	4
1.4 Inspeccionar variables.....	4
1.5 Trigger points	6
2. Depuración avanzada	7
2.1 Breakpoints condicionales	7
2.2 Hit count (contador de veces)	9
2.3 Log / Actions en un breakpoint (con posibilidad de no suspender).....	9
2.4 Cambiar el valor de una variable (Change Value).....	10
2.5 Exception breakpoint	11
2.6 Watchpoints (campo / field).....	12
2.7 Drop to Frame	13
3. Depuración de Hilos (threads) y concurrencia	15
3.1 Filtering	18
4. Anexo	19
4.1 Código fuente del proyecto debugDemo	19
4.2 Código fuente del proyecto concurrentDebugDemo	20

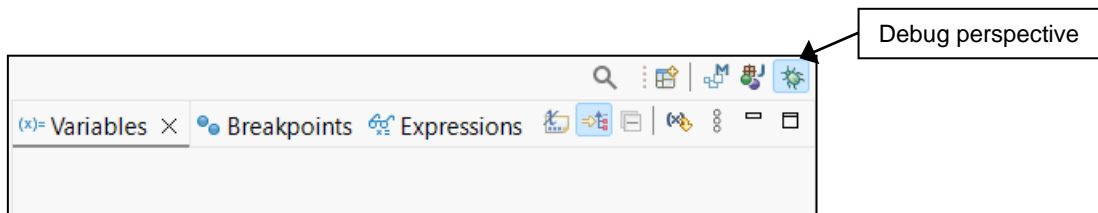
1. Conceptos básicos de depuración

1.1 Abrir la perspectiva de depuración (Debug Perspective)

- En Eclipse: Window > Perspective > Open Perspective > Other... y elegir Debug. Se puede alternar también entre perspectivas con el botón de la esquina superior derecha.

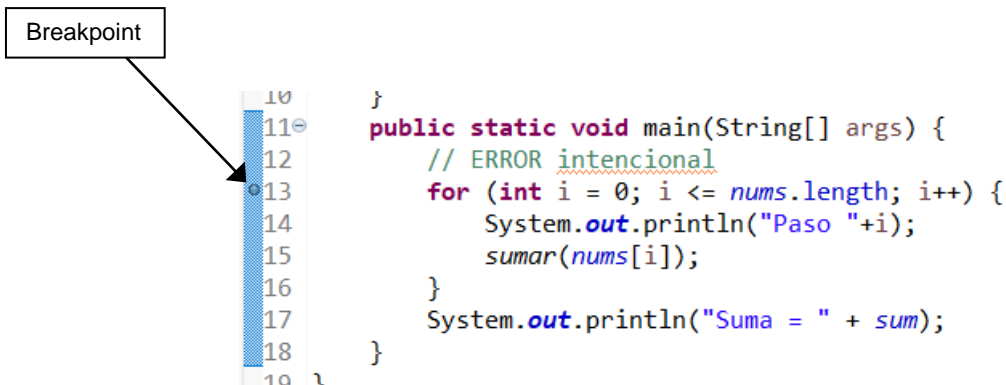


- La perspectiva Debug organiza varias vistas útiles: Debug (call stack e hilos), Variables, Breakpoints, Console, Expressions.



1.2 Poner un breakpoint y ejecutar en modo debug

- Abre SumDemo.java en el editor.
- Haz doble clic en la barra izquierda, junto a la línea `for (int i = 0; i <= nums.length; i++)` - aparecerá un marcador azul que indica el breakpoint.



- Ejecuta: clic derecho en el archivo SumDemo > Debug As > Java Application.
- Si Eclipse pregunta si quieres cambiar a la perspectiva Debug, acepta.

La ejecución se detiene en la línea del breakpoint, y la línea queda resaltada.

```
11 public static void main(String[] args) {  
12     // ERROR intencional  
13     for (int i = 0; i <= nums.length; i++) {  
14         System.out.println("Paso "+i);  
15         sumar(nums[i]);  
16     }  
17     System.out.println("Suma = " + sum);  
18 }  
19 }
```

1.3 Pasos (Stepping) – navegar en el código

- **Step Into (F5):** entra en la llamada interna de un método.
- **Step Over (F6):** ejecuta la línea completa y va a la siguiente (no entra en llamadas internas).
- **Step Return (F7):** termina el método actual y vuelve a la línea desde donde se llamó.
- **Resume (F8):** reanuda la ejecución hasta el siguiente breakpoint.

Se pueden usar también, además de los atajos, los botones en la barra de herramientas de Debug

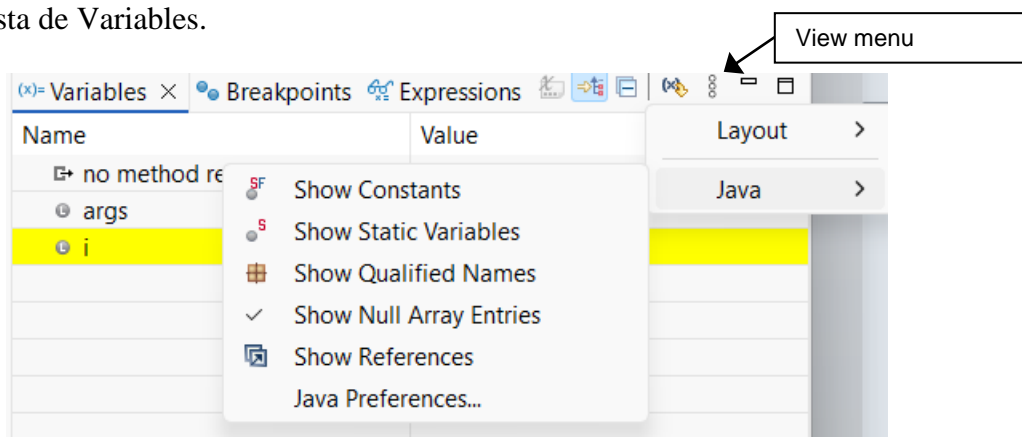


1.4 Inspeccionar variables

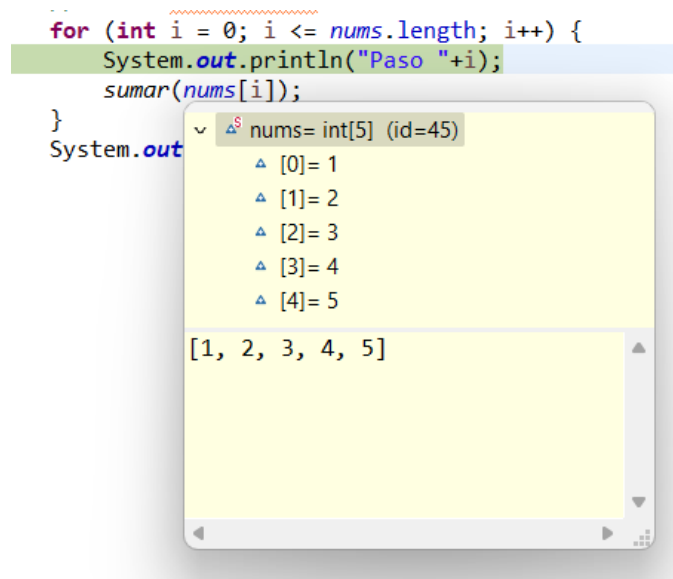
- **Pestaña Variables:** muestra variables locales y sus valores cuando la ejecución está suspendida.

(x)= Variables × Breakpoints Expressions	
Name	Value
no method return value	
args	String[0] (id=20)
i	3

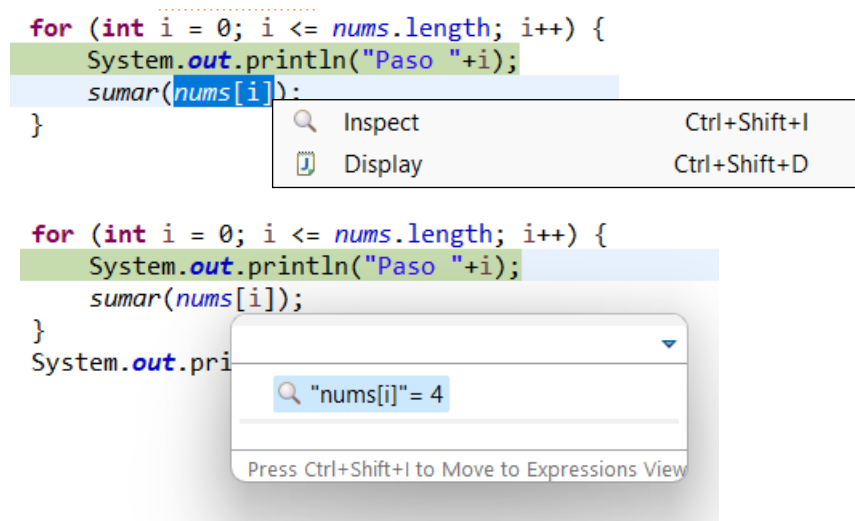
Por defecto, no suelen aparecer las variables estáticas ni las constantes. Se puede personalizar el contenido de la vista de Variables.



- **Hover:** coloca el cursor sobre una variable en el editor para ver su valor (ventana emergente).



- **Inspect / Display:** seleccionar una expresión en el editor, y luego hacer clic derecho > Inspect o Display para evaluarla.



Haz un debug de nuevo del programa, y cuando se detenga en el bucle for añade 3 expresiones para inspeccionar, en la pestaña correspondiente: i, nums[i], sum

Expressions	
Name	Value
> $x+y$ $?$ "i"	<error(s)_during_the_evaluation>
> $x+y$ $?$ "nums[i]"	<error(s)_during_the_evaluation>
$x+y$ $?$ "sum"	0
+ Add new expression	

Se produce un error al intentar evaluar las expresiones donde aparece la variable i, ya que aún no existe por no haberse ejecutado todavía el bucle for. Se admite cualquier tipo de expresión. Ej: sum+i*nums[3]

Ejemplo: Realiza un step over (F6) varias veces y observa cómo cambian los valores hasta identificar por qué se produce un error.

1.5 Trigger points

Los Trigger Breakpoints (también conocidos como trigger points), son una característica algo menos conocida en Eclipse, pero muy útil cuando se depuran programas complejos con muchos breakpoints.

Un Trigger Breakpoint es un breakpoint especial que activa o desactiva temporalmente otros breakpoints durante la depuración.

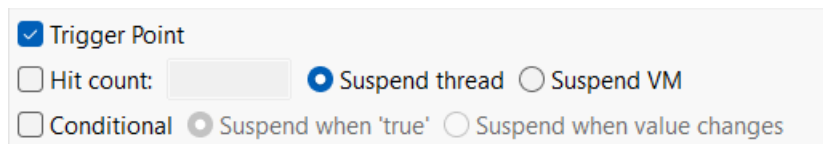
Al crear un Trigger point, todos los demás breakpoints quedan inactivos. Al ejecutar el programa en modo depuración, cuando la ejecución alcanza el trigger, Eclipse habilita los demás breakpoints (que estaban inactivos). De esta forma, al pasar del punto de activación, esos breakpoints se comportan normalmente hasta que se detiene o reinicia la sesión de depuración.

Esto permite controlar cuándo comienzan a ser relevantes los demás breakpoints, evitando que el programa se detenga antes de tiempo.

Un ejemplo típico de uso es cuando tenemos un programa con muchos breakpoints en distintos métodos, pero sólo interesa depurar esas partes después de que ocurra cierto evento o se invoque un método concreto (por ejemplo, después de una autenticación, una conexión de red o la carga de un archivo).

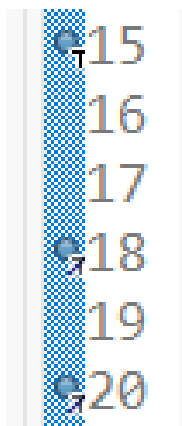
En lugar de activar y desactivar breakpoints manualmente, podemos poner un Trigger Breakpoint en el método o línea clave, de forma que todos los demás breakpoints se mantienen desactivados hasta que se alcanza el trigger.

Para convertir un breakpoint en un Trigger Breakpoint, basta con marcar esa casilla en las propiedades del breakpoint.



The image shows the 'Properties' dialog for a breakpoint in Eclipse. The 'Trigger Point' checkbox is checked. Below it, there are three options: 'Hit count' (unchecked), 'Suspend thread' (selected with a radio button), and 'Suspend VM' (unchecked). At the bottom, there are three options: 'Conditional' (unchecked), 'Suspend when 'true'' (selected with a radio button), and 'Suspend when value changes' (unchecked).

El trigger se distingue porque aparece una pequeña T sobre el punto, en la línea donde está definido. El resto de breakpoints, que quedan inhabilitados, aparecen con una T tachada superpuesta.



2. Depuración avanzada

2.1 Breakpoints condicionales

Como su nombre indica, los breakpoints condicionales sólo detienen la ejecución en un punto determinado si, además se cumple cierta condición. Por ejemplo, imaginemos que sólo nos interesa parar la ejecución cuando `i == 5`.

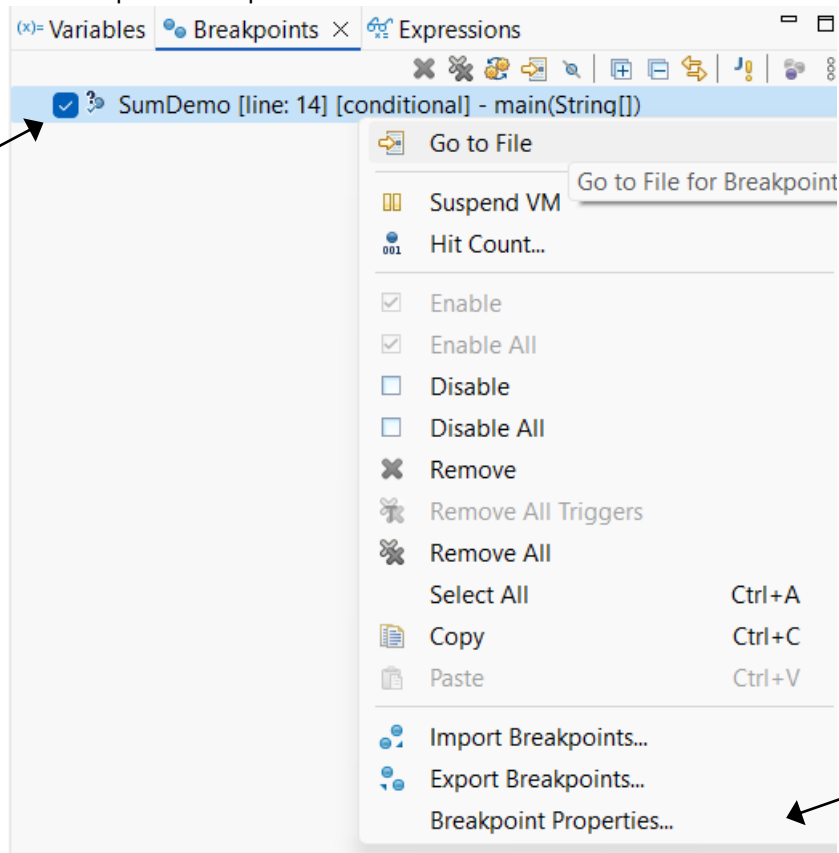
Cómo la variable `i` sólo existe dentro del bucle `for`, el breakpoint condicional sólo se puede definir en alguna de las líneas que hay dentro del `for`.

Borramos todos los breakpoints: con doble click sobre ellos en la barra a la izquierda de la instrucción, o desde la pestaña de vista de breakpoints con botón derecho > remove.

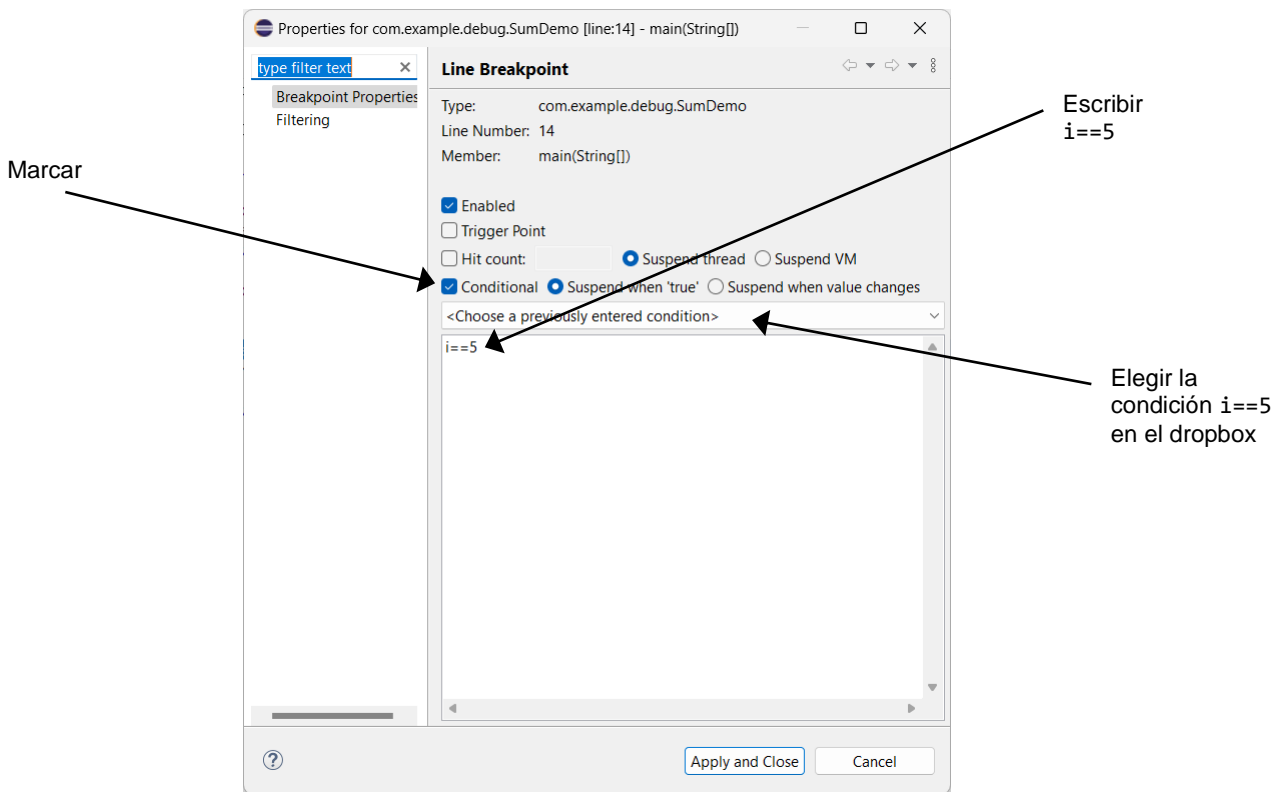
Creamos un nuevo breakpoint en la línea `System.out.println("Paso "+i);`

1. En la pestaña de Vista de Breakpoints
(si no es visible: Window > Show View > Other... > Debug > Breakpoints)
Localizamos el breakpoint que habíamos creado.
2. Right-click > Breakpoint Properties...

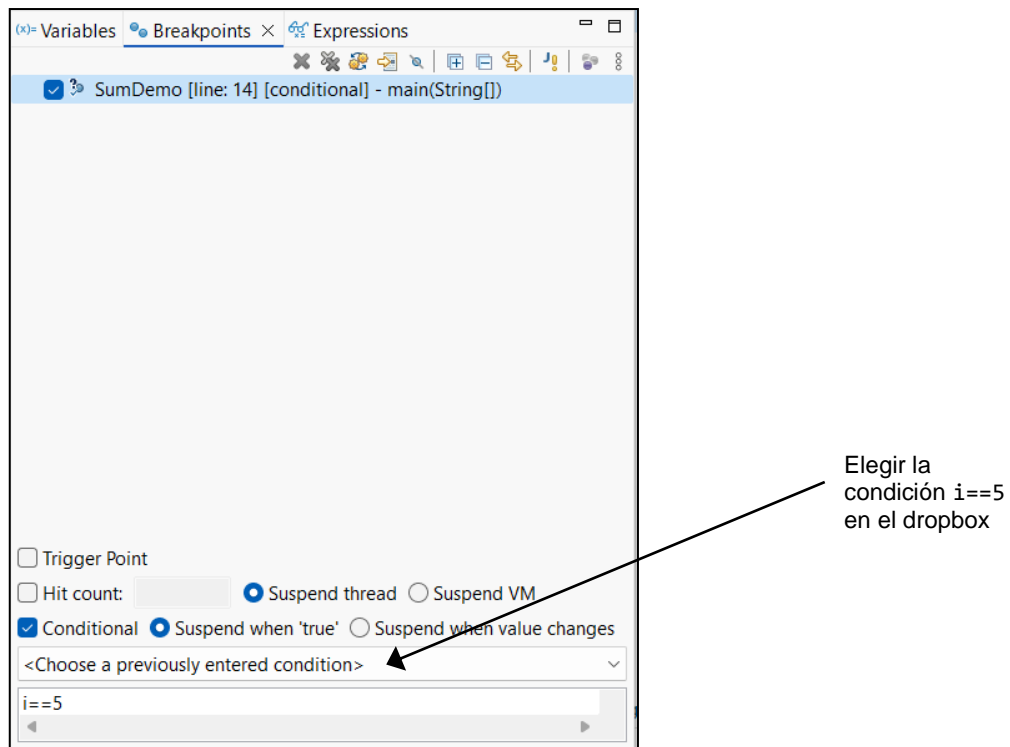
La casilla marcada indica que el breakpoint está activo



3. Activa la condición (marca “Conditional”) y escribe: `i == 5`.



También es posible marcarlo directamente en la pestaña de la vista Breakpoints



4. Se puede dejar marcada que la condición debe evaluarse en modo “suspend when true”.

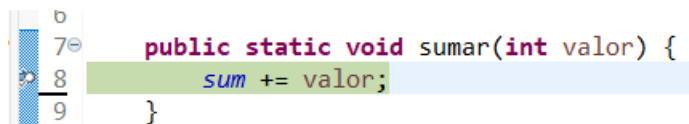
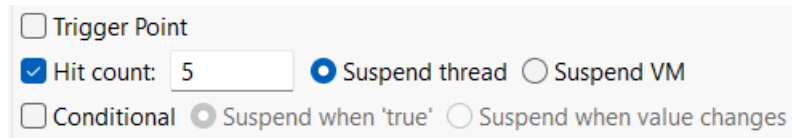
Los breakpoints condicionales hacen la depuración más lenta (Eclipse evalúa la expresión cada vez que pasa por esa línea), por lo que no conviene abusar de ellos.

2.2 Hit count (contador de veces)

Se puede pedir a Eclipse que sólo suspenda la ejecución después de que se haya ejecutado N veces la instrucción con el breakpoint.

Para ello se puede marcar en las propiedades del breakpoint la opción Hit Count y configurar el número de veces.

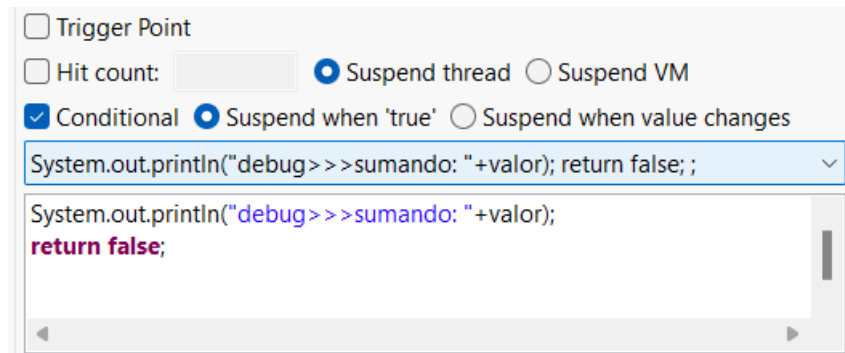
Ejemplo: borra todos los breakpoints y establece uno nuevo en la instrucción `sum += valor;`. Especifica un valor de 5 y ejecútalo en modo debug.



2.3 Log / Actions en un breakpoint (con posibilidad de no suspender)

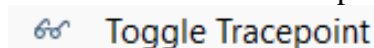
Si solo queremos mostrar cuando se alcanza una línea sin detener la ejecución, podemos hacerlo también con un breakpoint condicional.

Ejemplo: cambiamos el breakpoint de `sum +=valor;` desmarcando el Hit count y marcando el Conditional



En la expresión condicional se puede escribir cualquier código, siempre que se devuelva un valor de verdadero o falso. Este código se ejecutará cada vez que se alcance el breakpoint condicional para ver si se cumple la condición. Si en el código incluimos que se muestre un mensaje y devolvemos siempre un valor false, la ejecución no se detendrá, pero se mostrará por consola el mensaje cada vez que se ejecute ese breakpoint.

El mismo efecto se consigue en las versiones modernas de Eclipse pulsando con el botón derecho en la barra lateral de la instrucción `sum +=valor;` y seleccionando la opción Toggle Trace point, que creará automáticamente el breakpoint condicional con la línea `System.out.println.`



```

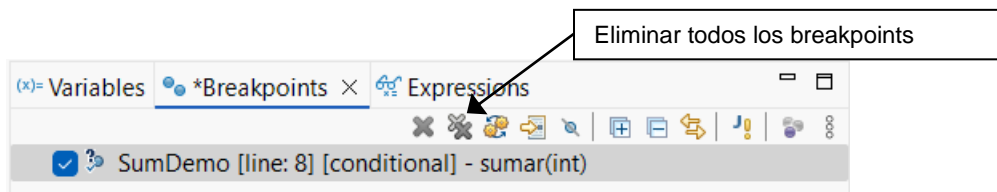
Console × Problems Debug Shell
SumDemo [Java Application] C:\eclipse2025\plugins\org.eclipse.justj
debug>>>sumando: 2
Paso 2
debug>>>sumando: 3
Paso 3
debug>>>sumando: 4
Paso 4
debug>>>sumando: 5
Paso 5

```

2.4 Cambiar el valor de una variable (Change Value)

Cuando la ejecución está suspendida en un breakpoint, se puede cambiar el valor de una variable, por ejemplo para hacer pruebas de valores concretos al depurar un error.

Ejemplo: En la vista de breakpoints, haz clic sobre el icono para eliminar todos los breakpoints definidos.



Eclipse solicita una confirmación y aceptaremos.

Una vez eliminados todos los breakpoints, definimos un breakpoint simple en la línea `sumar(nums[i]);`

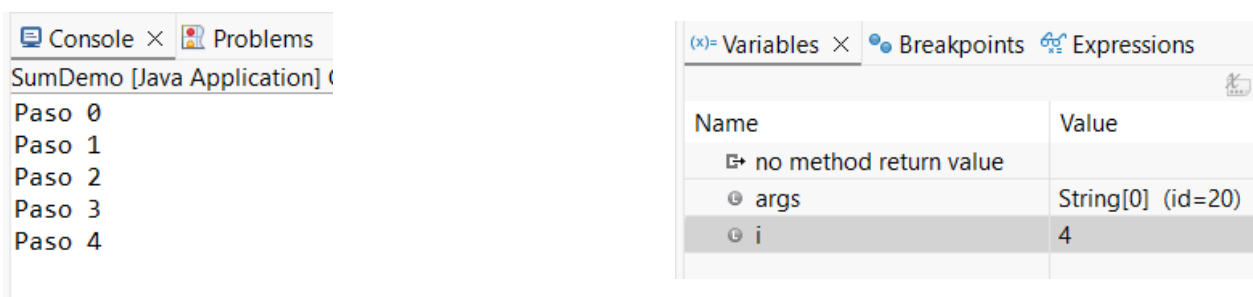
```

13     for (int i = 0; i <= nums.length; i++) {
14         System.out.println("Paso "+i);
15         sumar(nums[i]);

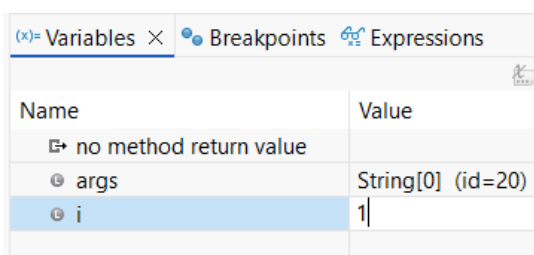
```

Ejecutamos en modo Debug y seleccionamos la pestaña variables para ver los valores que toma la variable `i`. Cada vez que se detiene la ejecución en el breakpoint, pulsamos F8 (resume).

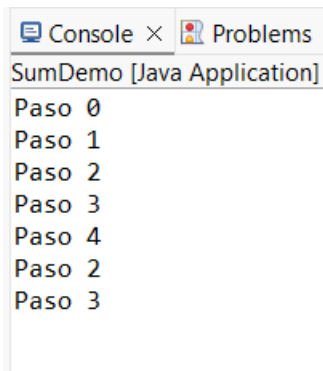
En la Consola se muestran los mensajes que se imprimen, y en la vista de variables los valores de la `i`.



Cuando el valor de la `i` llega a 4, lo modificamos en la vista de variables, asignando el valor 1.



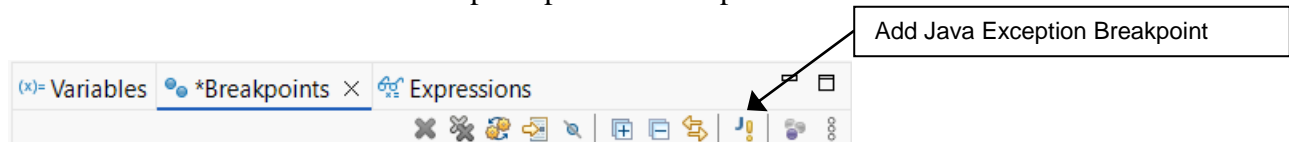
Si pulsamos de nuevo F8 (resume), vemos que la ejecución continúa, pero a partir del nuevo valor.



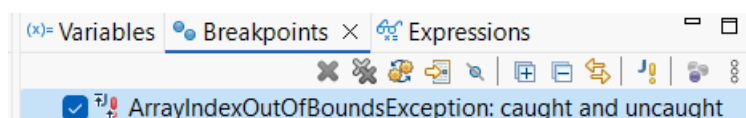
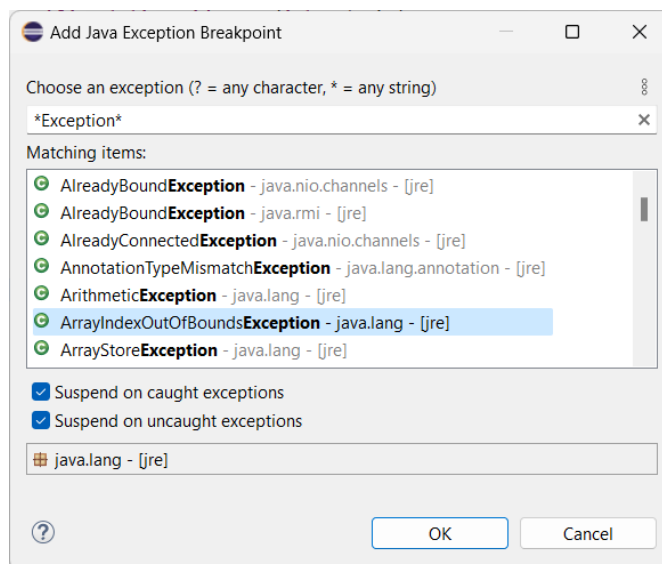
2.5 Exception breakpoint

Se pueden definir breakpoints globales, que no estén definidos para una instrucción en concreto. Un ejemplo son los breakpoints de excepciones. Se puede definir un breakpoint para que se suspenda la ejecución cuando el programa lance una excepción determinada.

Una vez eliminados todos los breakpoints, vamos a definir un exception breakpoint. Pulsa sobre el icono de añadir breakpoint para una excepción Java.



En la ventana que se abre, seleccionaremos la excepción `ArrayIndexOutOfBoundsException` y pulsaremos Ok



Si ejecutamos en modo debug el programa, se detiene, después de iterar varias veces, justo en la línea donde se provoca esa excepción.

```

11 public static void main(String[] args) {
12     // ERROR intencional
13     for (int i = 0; i <= nums.length; i++) {
14         System.out.println("Paso "+i);
15         sumar(nums[i]);
16     }
17     System.out.println("Suma = " + sum);

```

2.6 Watchpoints (campo / field)

Otro ejemplo de breakpoint global son los puntos de vigilancia (watchpoints). Un watchpoint detiene la ejecución cuando una variable de instancia (atributo/campo) es leída o escrita.

Ejemplo: Corrige el código fuente para que ya no salte una excepción.

Cambiar el <= por un <

```

for (int i = 0; i <= nums.length; i++) {
    System.out.println("Paso "+i);
    sumar(nums[i]);
}

```

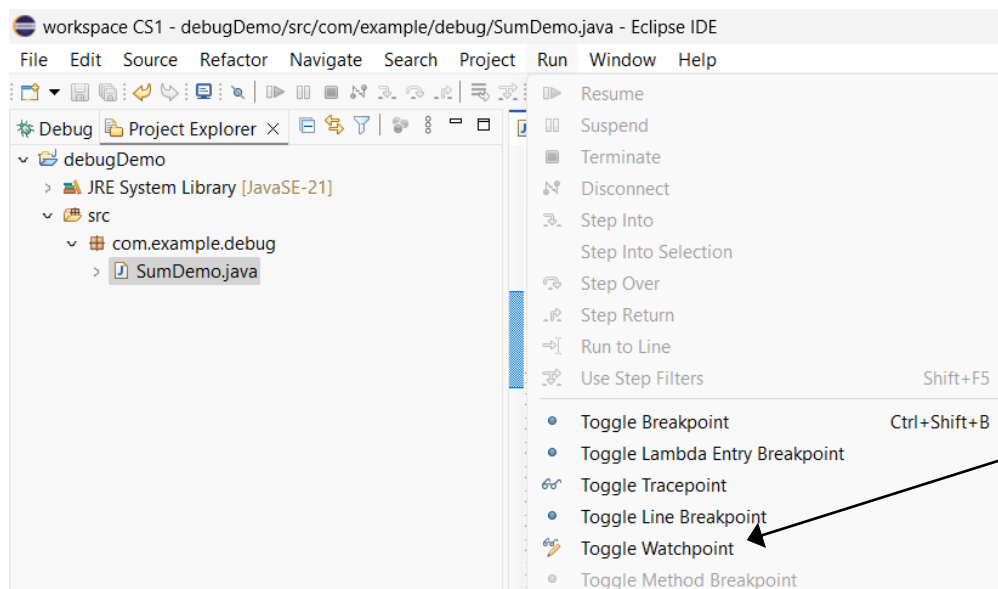
Borra todos los breakpoints y selecciona el atributo `variable` en la línea 6 del código.

```

SumDemo.java x
1 package com.example.debug;
2
3 public class SumDemo {
4     static int[] nums = {1, 2, 3, 4, 5};
5     static int sum = 0;
6     private static int variable;
7     public static void sumar(int valor) {
8         sum += valor;
9     }

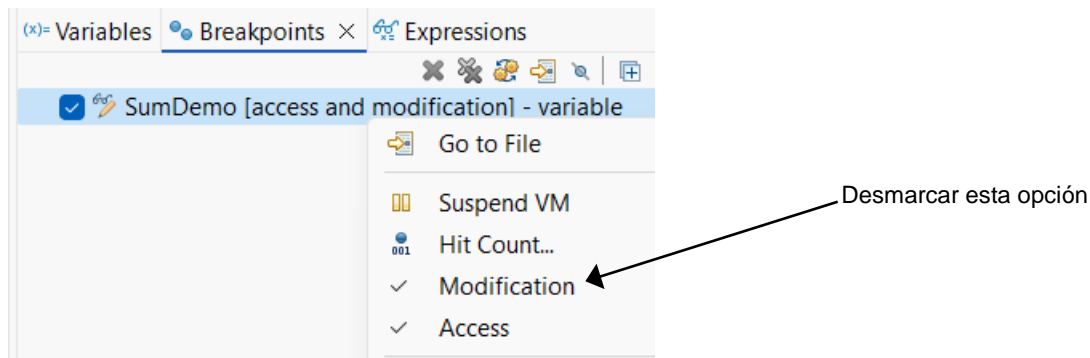
```

En el menú Run, de la barra superior de Eclipse, seleccionar Toggle Watchpoint.



Nota: Si la opción no aparece habilitada, ejecutar en modo Depuración, detener la depuración, volver a seleccionar el atributo `variable` y asegurarse de que la ventana que muestra el código es la ventana activa y el cursor está parpadeando en ella, antes de seleccionar de nuevo el menú Run.

En la vista de breakpoints aparece el watchpoint que acabamos de definir. Por defecto, se detendrá en cualquier línea donde se acceda o modifique el valor del campo. Si pulsamos el botón derecho sobre él, nos aparece un menú contextual donde podemos cambiar este comportamiento.



Desmarcamos la opción de Modificación (Modification) y ejecutamos en modo debug. La ejecución se detiene en la línea donde se consulta por primera vez el valor del atributo.

```
17      System.out.println("Suma = " + sum);
18      variable=1;
19      System.out.println("Variable =" + variable);
```

Si ahora cambiamos el comportamiento para que sólo quede marcada la opción de Modificación, al ejecutar en modo debug, se detiene en la línea donde el valor se modifica por primera vez.

```
17      System.out.println("Suma = " + sum);
18      variable=1;
19      System.out.println("Variable =" + variable);
```

2.7 Drop to Frame

Si durante la depuración de un método, nos hemos saltado alguna línea importante que queríamos depurar, no es necesario volver a ejecutar toda la depuración del programa completo desde el principio.

Ejemplo: Eliminar todos los breakpoints del programa y marcar uno en la primera línea del método "metodo()".

```
22 private static void metodo() {
23     System.out.println("línea del método");
24     System.out.println("línea del método");
```

Ahora ejecutamos en modo Debug y se detiene en esa línea. Si vamos pulsando repetidas veces F6 (Step over), iremos avanzando en las líneas del método.

```

22 private static void metodo() {
23     System.out.println("linea del método");
24     System.out.println("linea del método");
25     System.out.println("linea del método");
26     System.out.println("linea del método");
27     System.out.println("linea del método");
28     System.out.println("linea del método");
29     System.out.println("linea del método");

```

En la barra del menú de depuración, podemos pulsar sobre el icono de Drop to Frame, para "deshacer" todos los pasos que hemos dado y regresar a la primera línea del método.

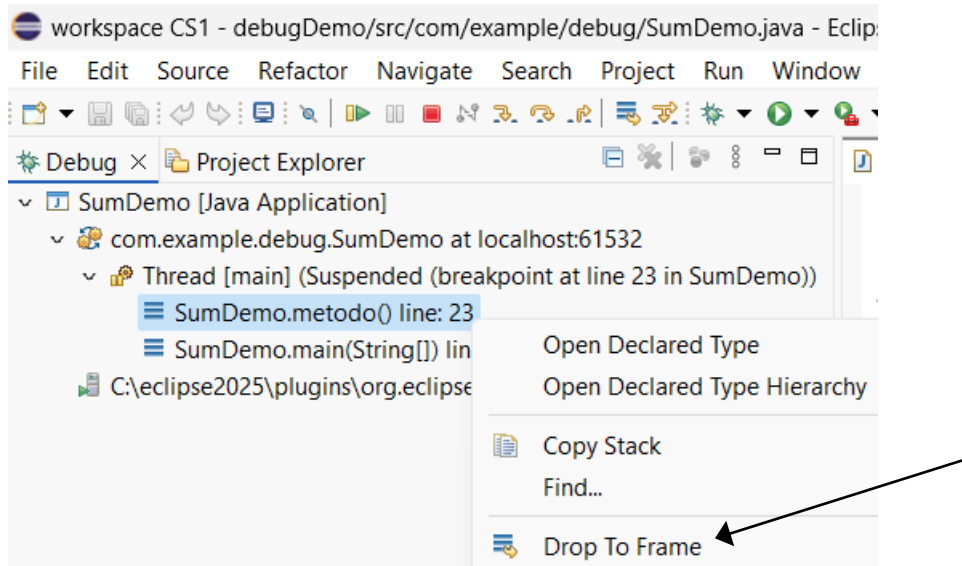


```

22 private static void metodo() {
23     System.out.println("linea del método");
24     System.out.println("linea del método");
25     System.out.println("linea del método");
26     System.out.println("linea del método");
27     System.out.println("linea del método");
28     System.out.println("linea del método");

```

Además de utilizar el icono de la barra de depuración, también podemos hacerlo desde la vista Debug, pulsando el botón derecho sobre el stack actual (el método que estamos ejecutando) y seleccionando la opción Drop To Frame en el menú desplegable.



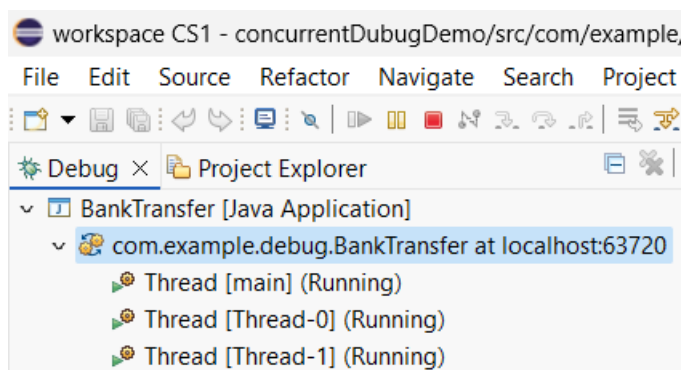
3. Depuración de Hilos (threads) y concurrencia

Para practicar la depuración de hilos (concurrencia), utilizaremos el proyecto `concurrentDebugDemo`, que importaremos en Eclipse.

El programa crea dos cuentas bancarias, A y B, con un balance de 1000 euros cada una. A continuación se crean dos hilos, que realizarán 100 transferencias de 10 euros. El primero de la cuenta A a la B, y el segundo a la inversa. Al finalizar el programa, el saldo de las dos cuentas debería ser de 1000 euros. Sin embargo, debido a condiciones de carrera, el saldo podría ser diferente. Se introduce un pequeño retardo a la hora de realizar las transferencias, para aumentar la probabilidad de carrera.

```
Console × Problems Debug Shell
<terminated> BankTransfer [Java Application]
Balances finales: A=970 B=990
```

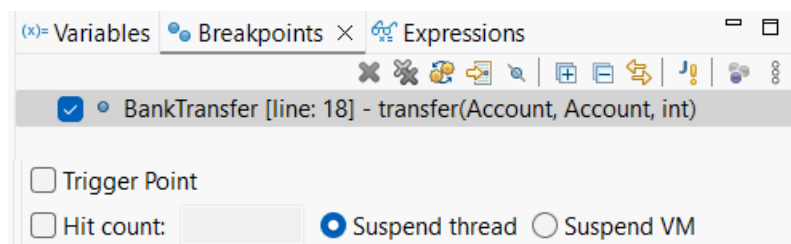
Si ejecutamos el programa en modo Debug, observamos en la pestaña de depuración los tres hilos que se crean, correspondientes al programa principal (main) y a los dos threads que realizan las transferencias.



Creamos un breakpoint en la línea 18, `from.balance -= amount;`

```
15         if (from.balance >= amount) {
16             // Simulamos retardo para aumentar la probabilidad de race condition
17             try { Thread.sleep(100); } catch (InterruptedException e) {}
18             from.balance -= amount;
```

En la vista de breakpoints, observamos dos propiedades: "Suspend thread" y "Suspend VM"

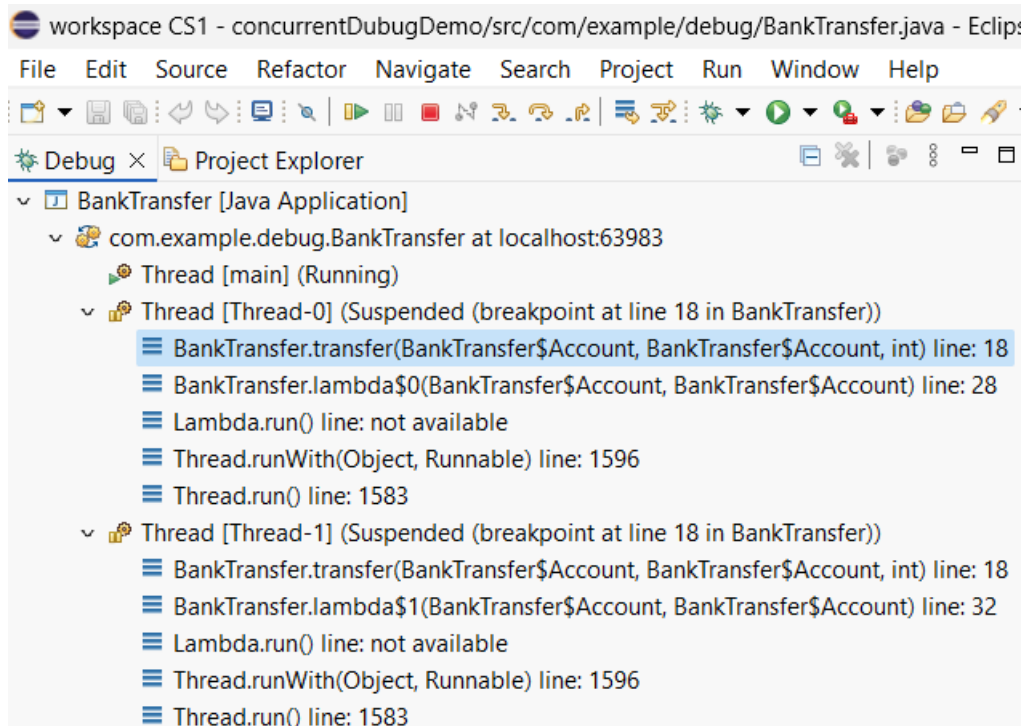


Suspend VM, pausa toda la máquina virtual de Java (todos los hilos), mientras que Suspend thread pausa sólo el hilo que alcanzó el breakpoint.

Por defecto se utiliza el Suspend thread, para no dejar congelada toda la aplicación.

En nuestro caso, esto pausará los dos hilos que realizan la transferencia, cuando ejecuten el método transfer, pero no pausará el programa principal (método main). El programa sin embargo, no terminará su ejecución, ya que el método main invoca los métodos join , que esperan a que terminen los dos hilos antes de continuar.

Si ejecutamos de nuevo en modo de depuración, los threads se detendrán en el breakpoint.



En la vista de Debug, vemos que ambos threads (Thread-0 y Thread-1), se han detenido en la línea 18 del método transfer, en la clase BankTransfer.

En la vista de Variables, podemos ver las variables que pertenecen al thread seleccionado (Thread-0)

(x)= Variables × Breakpoints Expressions	
Name	Value
no method return value	
from	BankTransfer\$Account (id=25)
balance	1000
name	"A" (id=27)
to	BankTransfer\$Account (id=26)
balance	1000
name	"B" (id=35)
amount	10

Pulsando F8 (resume) hacemos que continúe la ejecución del Thread-0, hasta que se detiene de nuevo al llegar al breakpoint. Si volvemos a seleccionar el método transfer del Thread-0, en la vista de debug, se observa, en la vista de Variables, que se ha producido una transferencia de 10 euros, en este caso de la cuenta A a la cuenta B.

(x)= Variables × Breakpoints Expressions	
Name	Value
no method return value	
from	BankTransfer\$Account (id=25)
balance	990
name	"A" (id=27)
to	BankTransfer\$Account (id=26)
balance	1010
name	"B" (id=35)
amount	10

Si pulsamos repetidamente F6 (Step over), comprobamos que continúa la ejecución del hilo que tenemos seleccionado, mientras que el otro sigue pausado, y se realizan transferencias sólo de la cuenta A a la cuenta B.

(x)= Variables × Breakpoints Expressions	
Name	Value
no method return value	
from	BankTransfer\$Account (id=25)
balance	920
name	"A" (id=27)
to	BankTransfer\$Account (id=26)
balance	1080
name	"B" (id=35)
amount	10

De esta forma podemos depurar los hilos de manera individual.

Si queremos depurar el otro hilo (Thread-1), sólo tenemos que seleccionarlo en la vista de Debug.

workspace CS1 - concurrentDebugDemo/src/com/example/debug/BankTransfer.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help

Debug × Project Explorer

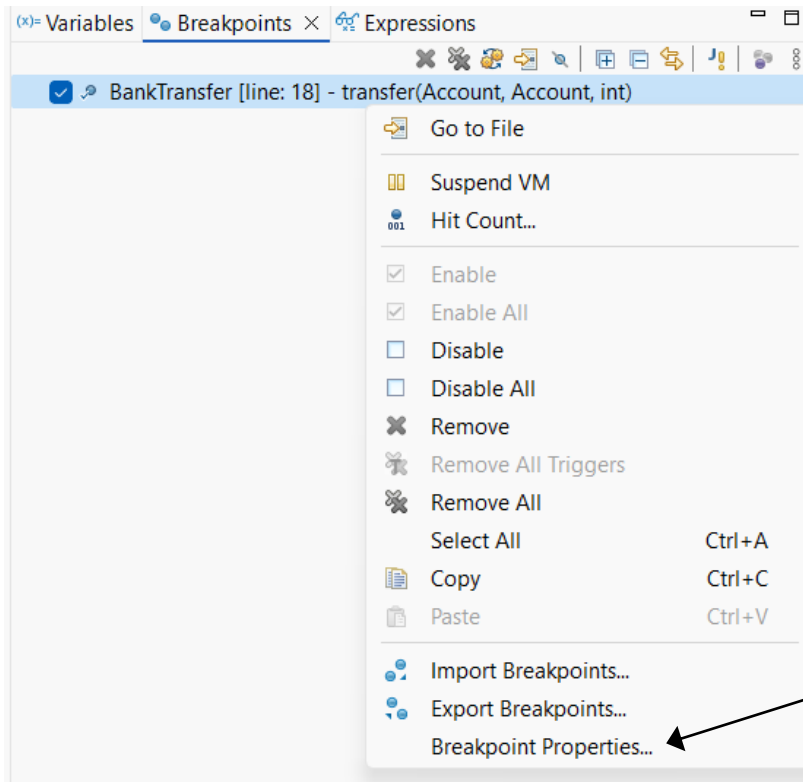
- BankTransfer [Java Application]
 - com.example.debug.BankTransfer at localhost:63986
 - Thread [main] (Running)
 - Thread [Thread-0] (Suspended)
 - BankTransfer.transfer(BankTransfer\$Account, BankTransfer\$Account, int) line: 21
 - BankTransfer.lambda\$0(BankTransfer\$Account, BankTransfer\$Account) line: 28
 - Lambda.run() line: not available
 - Thread.runWith(Object, Runnable) line: 1596
 - Thread.run() line: 1583
 - Thread [Thread-1] (Suspended (breakpoint at line 18 in BankTransfer))
 - BankTransfer.transfer(BankTransfer\$Account, BankTransfer\$Account, int) line: 18
 - BankTransfer.lambda\$1(BankTransfer\$Account, BankTransfer\$Account) line: 32
 - Lambda.run() line: not available
 - Thread.runWith(Object, Runnable) line: 1596
 - Thread.run() line: 1583

3.1 Filtering

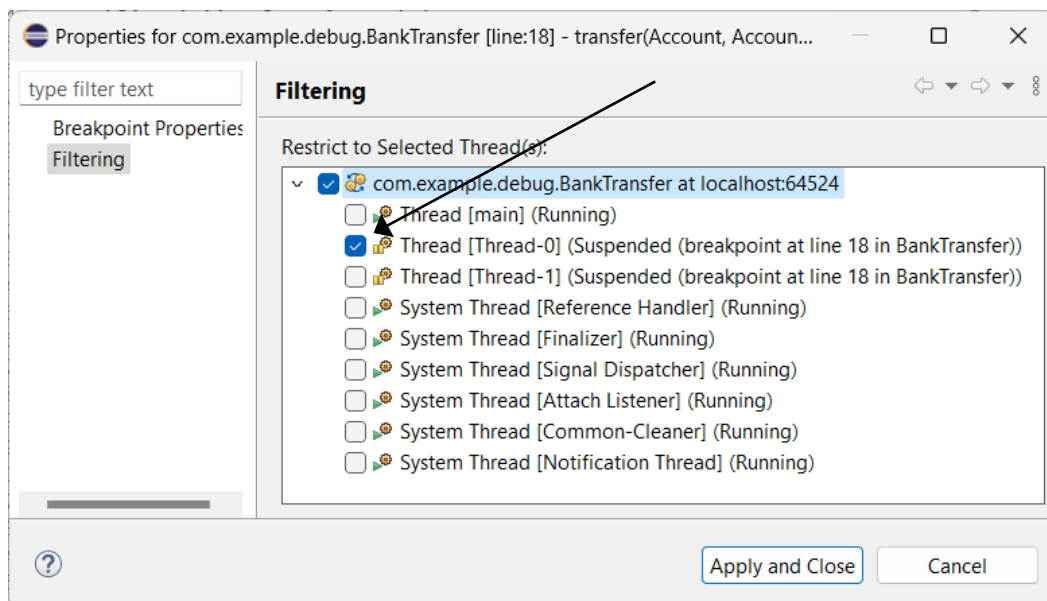
Podemos especificar que determinados breakpoints sólo se apliquen a la ejecución de ciertos hilos. Para ello ejecutamos el programa en modo Debug, y pausamos la ejecución, bien porque se alcance un breakpoint o porque pulsamos el icono "pausa".



En la vista de breakpoints, seleccionamos el breakpoint que queremos filtrar y pulsamos el botón derecho sobre él, para acceder a sus propiedades.



En el apartado Filtering, podemos seleccionar el hilo de ejecución al que queremos aplicar el breakpoint.



4. Anexo

4.1 Código fuente del proyecto debugDemo

```
package com.example.debug;

public class SumDemo {
    static int[] nums = {1, 2, 3, 4, 5};
    static int sum = 0;
    private static int variable;
    public static void sumar(int valor) {
        sum += valor;
    }
    public static void main(String[] args) {
        // ERROR intencional
        for (int i = 0; i <= nums.length; i++) {
            System.out.println("Paso "+i);
            sumar(nums[i]);
        }
        System.out.println("Suma = " + sum);
        variable=1;
        System.out.println("Variable =" + variable);
        metodo();
    }
    private static void metodo() {
        System.out.println("línea del método");
        System.out.println("línea del método");
        System.out.println("línea del método");
        System.out.println("línea del método");
        System.out.println("línea del método");
        System.out.println("línea del método");
        System.out.println("línea del método");
        System.out.println("línea del método");
        System.out.println("línea del método");
        System.out.println("línea del método");
    }
}
```

4.2 Código fuente del proyecto concurrentDebugDemo

```
package com.example.debug;

public class BankTransfer {
    public static class Account {
        public String name;
        public int balance;

        public Account(String name, int balance) {
            this.name = name;
            this.balance = balance;
        }
    }

    public static void transfer(Account from, Account to, int amount) {
        if (from.balance >= amount) {
            // Simulamos retardo para aumentar la probabilidad de race condition
            try { Thread.sleep(100); } catch (InterruptedException e) {}
            from.balance -= amount;
            to.balance += amount;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Account a = new Account("A", 1000);
        Account b = new Account("B", 1000);

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 100; i++) transfer(a, b, 10);
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 100; i++) transfer(b, a, 10);
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Balances finales: A=" + a.balance + " B=" + b.balance);
    }
}
```