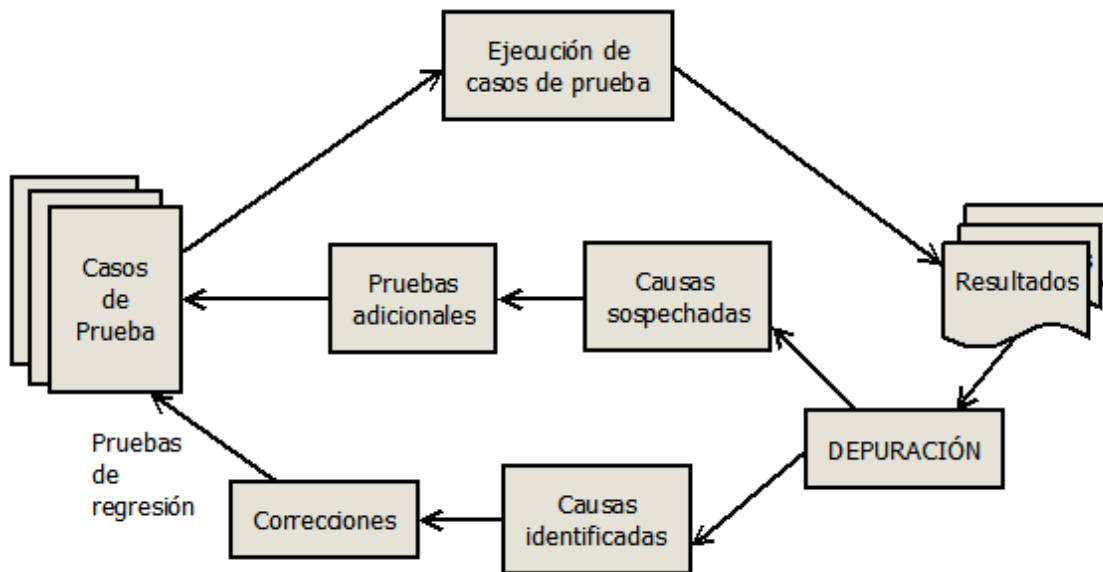


El proceso de depuración comienza con la ejecución de un caso de prueba. Se evalúan los resultados de la ejecución y fruto de esa evaluación se comprueba que hay una falta de correspondencia entre los resultados esperados y los obtenidos realmente.

El proceso de depuración siempre tiene uno de los dos resultados siguientes:

1. Se encuentra la causa de error, se corrige y se elimina.
2. No se encuentra la causa de error. Sospechar la causa, diseñar casos de prueba que ayuden a confirmar las sospechas y volver a repetir las pruebas.




Depurador o debugger.

Al desarrollar programas cometemos dos tipos de errores: errores de **compilación** y errores **lógicos**. Los primeros son fáciles de corregir ya que normalmente usamos un IDE que nos ayuda a hacerlo. Los errores de tipo lógico son más difíciles de detectar ya que el programa se puede compilar con éxito y sin embargo su ejecución puede devolver resultados inesperados o erróneos. A estos errores de tipo lógico se les suele llamar *bugs*.

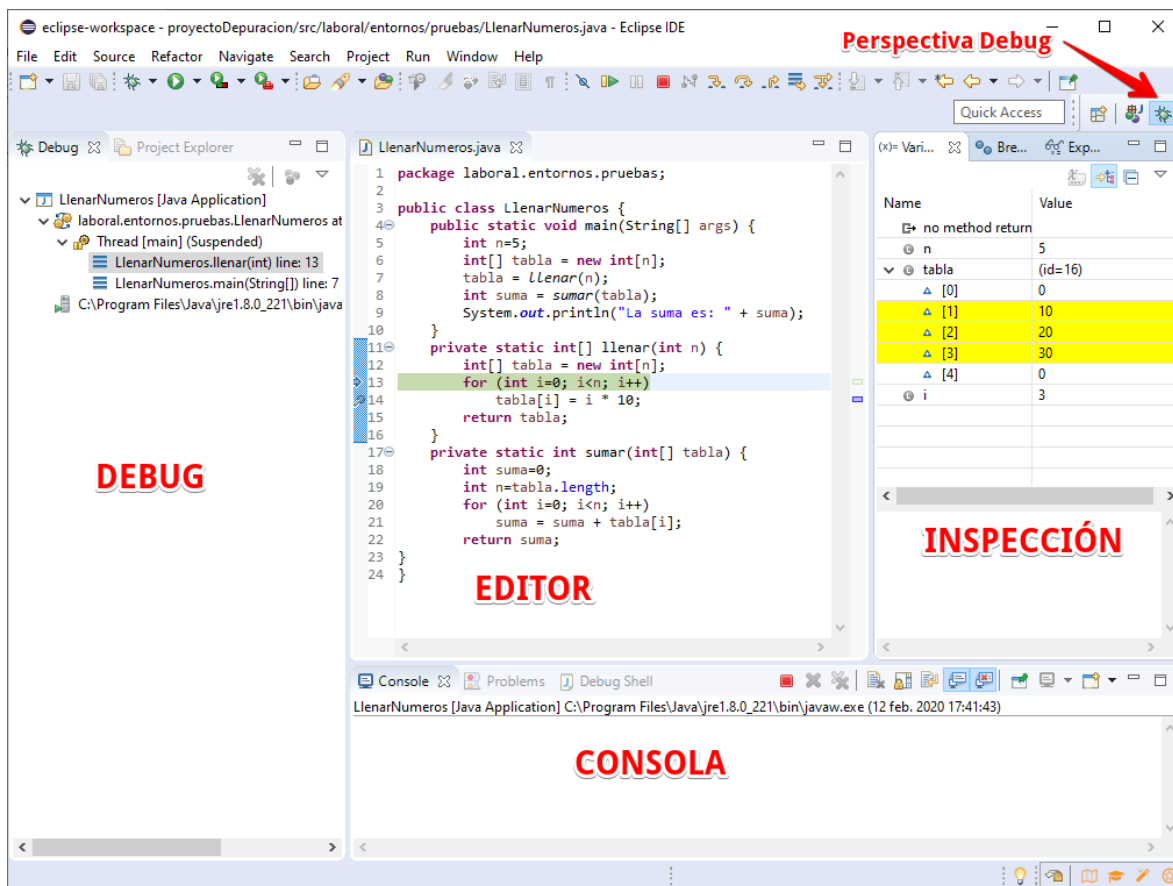
Los entornos de desarrollo incorporan una herramienta conocida como **depurador** (*debugger*) para ayudarnos a corregir este tipo de errores.

El depurador nos permite analizar el código del programa mientras se ejecuta. Permite establecer puntos de interrupción o de ruptura, suspender la ejecución del programa, ejecutar el código paso a paso y examinar el contenido de las variables.

Depurador de Eclipse.

En Eclipse podemos lanzar el depurador de varias formas: con el botón Debug,  seleccionando el menú **Run/Debug** (F11) o con el menú contextual que se muestra al hacer clic con el botón derecho en la clase que se va a ejecutar y seleccionando **Debug As /Java Application**.

Trabajaremos desde la perspectiva depuración (*Debug*), que tiene este aspecto:



En esta perspectiva distinguimos varias zonas:

- En la vista **EDITOR** se va marcando la traza de ejecución del programa mostrándose una flechita azul en el margen izquierdo de la línea que se está ejecutando.
- La vista **DEBUG** muestra los hilos de ejecución, en este caso solo muestra un hilo (*Thread [main]*) y debajo la clase en la que está parada la ejecución mostrando los métodos y los números de línea.
- La vista **INSPECCIÓN** permite ver los valores de las variables y de los puntos de ruptura (*breakpoints*) que intervienen en el programa en un instante dado.


Desde aquí también se puede modificar el valor de las variables, basta con hacer clic en el valor y cambiarlo; el nuevo valor se usará en los siguientes pasos de ejecución.


También desde la pestaña Breakpoints se puede activar o desactivar un punto de ruptura, eliminarlo, configurarlo para que la ejecución se detenga cuando se pase por él un determinado número de veces, etc.


- La vista **CONSOLA** muestra la consola de ejecución del programa que se está depurando.


Desde el menú **Run** de la perspectiva de depuración se pueden observar varias opciones, algunas de ellas también se encuentran en los botones que aparecen en la barra de herramientas:





 **Resume** Reanuda un hilo suspendido. Se utiliza cuando queremos analizar instrucción por instrucción y deseamos que el depurador se pare en la siguiente línea donde hay un breakpoint.


 **Suspend** Suspende el hilo seleccionado.

 **Terminate** Finaliza el proceso de depuración.

 **Step Into** Se ejecuta paso a paso cada instrucción. Si el depurador encuentra una llamada a un método, al pulsar Step Into se irá a la primera instrucción de dicho método.

 **Step Over** Se ejecuta paso a paso cada instrucción, pero si el depurador encuentra un método, se irá a la siguiente instrucción, sin entrar en el código del método.

 **Step Return** Si nos encontramos dentro de un método, el depurador sale del método actual.

 **Use Step Filters** Cambia los filtros de paso de activado a desactivado. Estos filtros están definidos en el menú **Windows/Preferences/Java/Debug/Step Filtering**. Se utilizan normalmente para filtrar tipos que no se desean recorrer durante la depuración.

Puntos de ruptura y seguimiento.

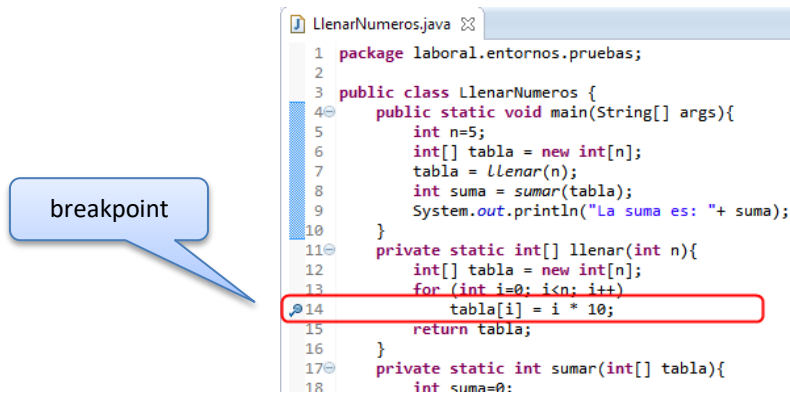
Partimos de la siguiente clase para empezar a utilizar el depurador:


```
package laboral.entornos.pruebas;

public class LlenarNumeros {
    public static void main(String[] args) {
        int n=5;
        int[] tabla = new int[n];
        tabla = llenar(n);
        int suma = sumar(tabla);
        System.out.println("La suma es: " + suma);
    }
    private static int[] llenar(int n) {
        int[] tabla = new int[n];
        for (int i=0; i<n; i++)
            tabla[i] = i * 10;
        return tabla;
    }
    private static int sumar(int[] tabla) {
        int suma=0;
        int n=tabla.length;
        for (int i=0; i<n; i++)
            suma = suma + tabla[i];
        return suma;
    }
}
```

Abrimos la perspectiva de depuración (**Window/Perspective/Open Perspective/Debug**) o botón 

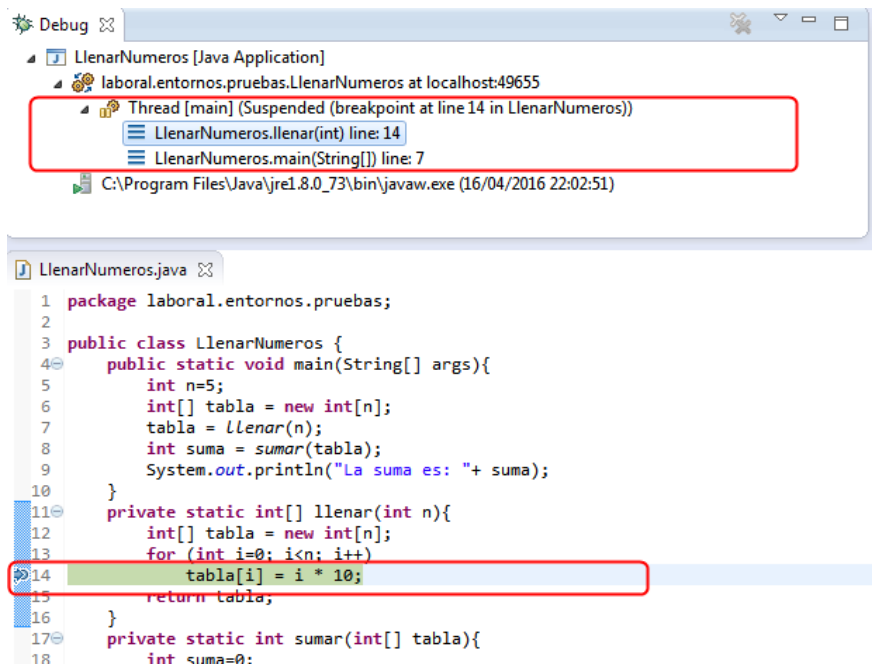
Para poner un breakpoint, hacemos doble clic en el margen izquierdo del editor, justo en la línea donde queremos que se detenga la ejecución, aparecerá un circulito a la izquierda.



Una vez establecido el punto de ruptura ejecutamos el programa en modo depuración, botón **Debug** . El programa se ejecutará de forma normal hasta que la ejecución llegue al punto de ruptura establecido, en ese momento se detendrá.

En la ventana Debug aparece la **pila de llamadas**, donde se ven cada uno de los hilos de ejecución. Debajo de esta línea se muestra la clase con el **método donde está parada la ejecución de un método**, clase *LlenarNumeros*, método *llenar()*. Se muestra también el número de línea donde está detenida la ejecución, en este caso 14. La siguiente línea muestra quién ha llamado a este método, clase *LlenarNumeros* y dentro del método *main* en la línea 7 está la sentencia *tabla = llenar(n)*;

Al hacer clic en estas líneas se resalta, en el editor, la línea de código en la que el programa está parado.



A continuación podemos usar los botones:

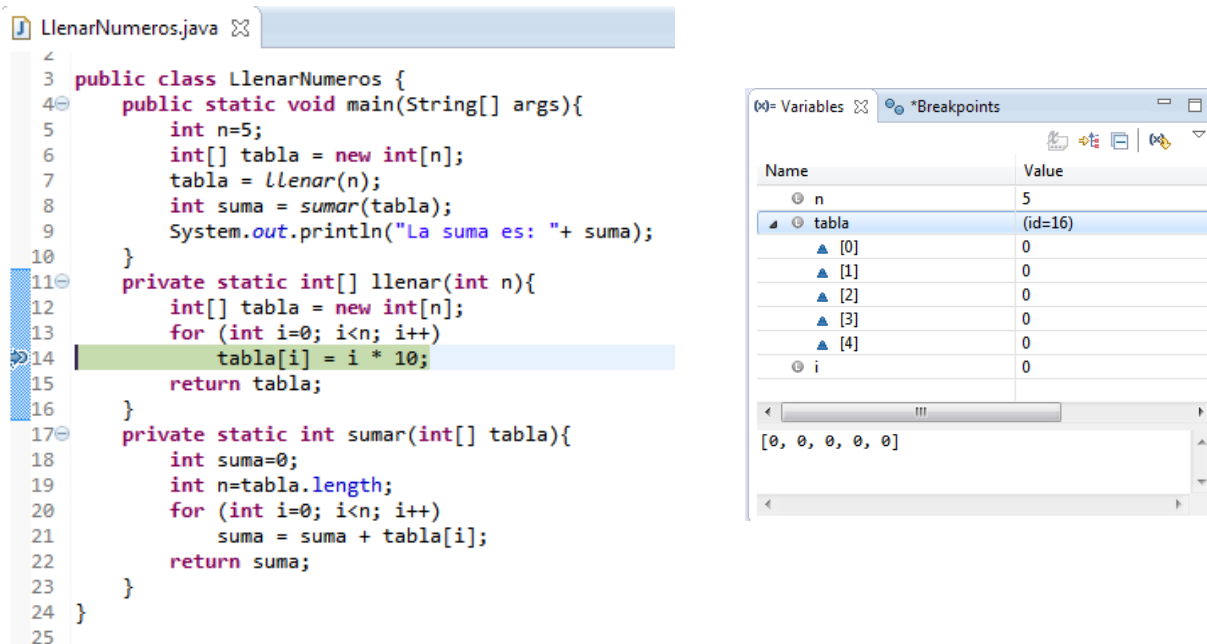


- **Step into (F5)**: para ejecutar paso a paso.
- **Step Over (F6)**: para ejecutar paso a paso, pero si encuentra un método saltarlo.
- **Step Return (F7)**: para salir de un método si estamos dentro.

Examen y modificación de variables.

Desde la vista de INSPECCIÓN, desde la pestaña de **Variables** podemos inspeccionar las variables definidas en el punto en el que el programa está detenido en este momento.

En este punto se muestra el valor de la variable `n`, los elementos de la tabla inicializados a 0 y el valor inicial de la `i` que es 0. En la parte inferior se muestra el valor de la variable seleccionada, en este caso `tabla[0,0,0,0,0]`.

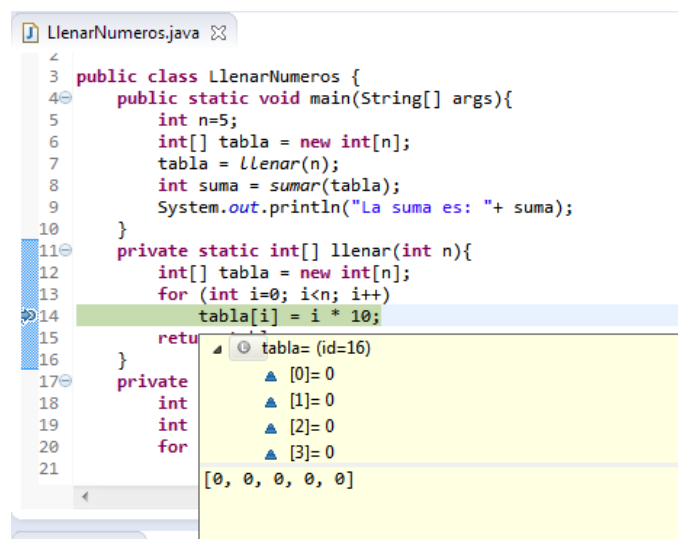


```
4
5 public class LlenarNumeros {
6     public static void main(String[] args){
7         int n=5;
8         int[] tabla = new int[n];
9         tabla = llenar(n);
10        int suma = sumar(tabla);
11        System.out.println("La suma es: " + suma);
12    }
13    private static int[] llenar(int n){
14        int[] tabla = new int[n];
15        for (int i=0; i<n; i++)
16            tabla[i] = i * 10;
17        return tabla;
18    }
19    private static int sumar(int[] tabla){
20        int suma=0;
21        int n=tabla.length;
22        for (int i=0; i<n; i++)
23            suma = suma + tabla[i];
24        return suma;
25    }
26 }
```

Name	Value
n	5
tabla (id=16)	
[0]	0
[1]	0
[2]	0
[3]	0
[4]	0
i	0

[0, 0, 0, 0, 0]

Desde aquí se puede modificar el valor de una variable haciendo doble clic sobre él y escribiendo el nuevo valor.



```
4
5 public class LlenarNumeros {
6     public static void main(String[] args){
7         int n=5;
8         int[] tabla = new int[n];
9         tabla = llenar(n);
10        int suma = sumar(tabla);
11        System.out.println("La suma es: " + suma);
12    }
13    private static int[] llenar(int n){
14        int[] tabla = new int[n];
15        for (int i=0; i<n; i++)
16            tabla[i] = i * 10;
17        return tabla;
18    }
19    private static int sumar(int[] tabla){
20        int suma=0;
21        int n=tabla.length;
22        for (int i=0; i<n; i++)
23            suma = suma + tabla[i];
24        return suma;
25    }
26 }
```

tabla= (id=16)

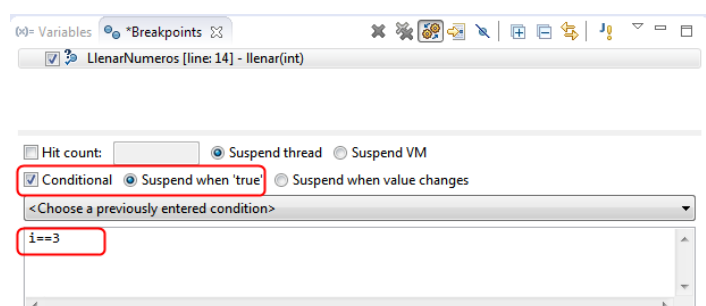
- [0]= 0
- [1]= 0
- [2]= 0
- [3]= 0

[0, 0, 0, 0, 0]

Otra forma de ver el contenido de la variable es pasando el puntero del ratón por ella, se abre una ventana mostrándonos la información.

Podemos establecer puntos de ruptura **condicionales**, por ejemplo, con el punto de ruptura establecido en la línea 14 podemos indicar que la ejecución se detenga cuando el valor de la `i` sea 3.

Desde la vista inspección, en la pestaña **Breakpoint**:



Hit count: Suspend thread ☒ Suspend VM ☐ Suspend when value changes

☒ Conditional ☒ Suspend when 'true' ☐ Suspend when value changes

<Choose a previously entered condition>

i==3