

```

import threading,socket,time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self,addr,port,func):
        try:
            self.sck.connect((addr,port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error:Could not connect"
    def listen(self,host,port,func):
        try:
            self.sck.bind((host,port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error:Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x,ho=self.sck.accept()
                self.todo=2
                self.client=ho
                self.handle=self.sck
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
    def send(self,data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()

```

DAM/DAW

P R O G R A M A C I Ó N

04.3

Doc Debug Test

Indice

- [Documentación](#)
- Depuración
 - [Introducción](#)
 - [JDB Introducción](#)
 - [JDB Básico](#)
 - [JDB Remoto](#)
 - [Usando Netbeans](#)
- Pruebas con JUnit
 - [Introducción](#)
 - [Descarga e Instalación](#)
 - [Un ejemplo sencillo](#)
 - [Integración con Netbeans](#)

```
import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sock=socket(socket.AF_INET,socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self,addr,port,func):
        try:
            self.sock.connect((addr,port))
            self.handle=self.sock
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error:Could not connect"
    def listen(self,host,port,func):
        try:
            self.sock.bind((host,port))
            self.sock.listen(2)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error:Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x,ho=self.sock.accept()
                self.todo=2
                self.client=ho
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
            if self.func:
                self.send(self.data)
            self.handle.send(data)
        self.close(self)
        self.sock.close()
```

Documentación (I)

- Con objeto de facilitar y promover el buen uso de los bloques de comentarios y la generación automática de documentación, el JDK incluye una la utilidad *javadoc*
- *javadoc*, a partir de los archivos de código fuente que han sido comentados usando ciertas “etiquetas” o *tags*, producirá una serie de documentos **HTML** describiendo las clases, métodos, atributos y constantes contenidas en ellos.
- Los bloques de comentarios *javadoc* se inician con la secuencia */*** y finalizan con **/*. Cada línea de comentario entre esas dos secuencias suele comenzar con un *** (aunque se ignora). Puede contener **etiquetas HTML**
- Un bloque de comentario precediendo la declaración de una clase, atributo o método, es procesado por *javadoc* como un comentario acerca de la clase, atributo o método correspondiente

Documentación (II)

- Los *tags javadoc* más comunes son (que deberían mantener ese **orden**):
 - **@author *texto***, identifica cada autor (clase/interfaz)
 - **@version *ver***, indica la versión (clase/interfaz)
 - **@param *nombreParam***, identifica un parámetro (método/constructor)
 - **@return *descripción***, tipo y rango del valor devuelto (método)
 - **@throws *exceptionName***, indica una condición de error que puede ser generada (método)
 - **@see *identificador***, para crear una lista (uno por línea) de “*ver también*”. Puede llevar asociada una etiqueta **{@link}** para crear un enlace
 - **@since *release***, indica cuál o cuándo fue la primera versión
 - **@deprecated**, indica que la clase o método ya no se usa. Suele ir acompañada de **@see** o **{@link}**
- **referencia:** [How to Write Doc Comments for the Javadoc Tool](#) (Oracle)

Documentación (III)

❖ Ejemplo:

```
//: Factorial.java

/**
 * La clase Factorial permite realizar el cálculo del factorial de un número natural [1,2,...]
 *
 * @author bowman@hal9000
 * @version 1.0
 * @since 01-01-2001
 */
public class Factorial {
    /**
     * Utiliza una aproximación iterativa para la realización del cálculo del factorial.
     *
     * @param n Número entero del cuál queremos obtener su factorial
     * @return el factorial del número pasado como argumento
     * @see #factRec(int)
     */
    public static long factLoop(int n) {
        long f = 1;

        for(int i=2; i<=n; i++) { f *= i; }

        return f;
    }
}
```

*doc de la clase
Factorial*

*doc del
método
factLoop()*

Documentación (IV)

```
/**
 * Utiliza una aproximación recursiva para la
 * realización del cálculo del factorial.
 *
 * @param n Número entero del cuál queremos obtener su factorial
 * @return el factorial del número pasado como argumento
 * @see #factLoop(int)
 */
public static long factRec(int n) {
    if(n==1)
        return 1;

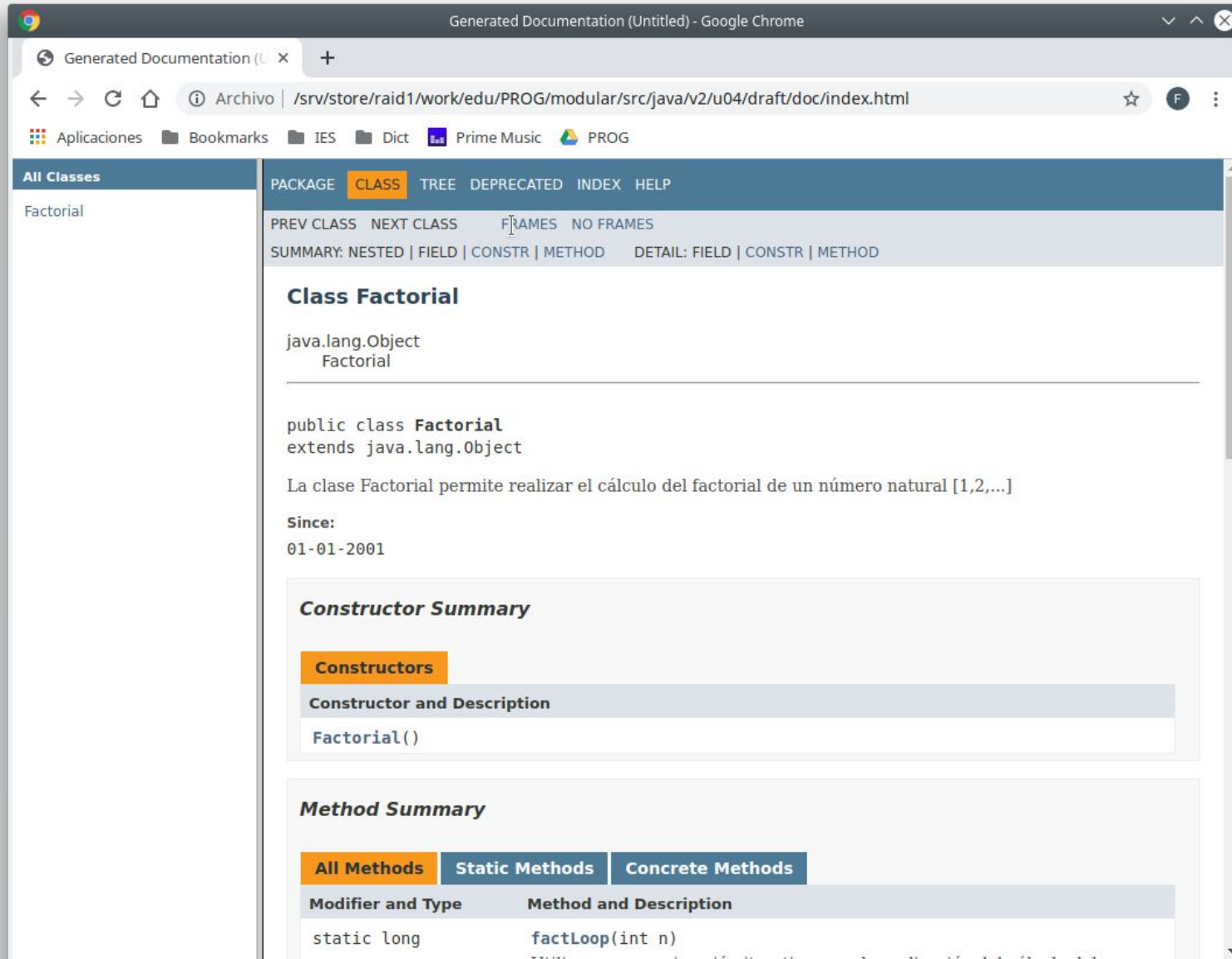
    return n*factRec(n-1);
}
```

*doc del
método
factRec()*

- Para generar la documentación HTML de la clase *Factorial* usaremos el comando *javadoc*. La opción *-d* nos permitirá indicar el directorio de destino. Opcionalmente, podemos añadir *--frames* para generar marcos html (*deprecated*) y *-sourcepath* para indicar el directorio con los fuentes.

```
bowman@hal:~$ javadoc --frames -d doc -sourcepath <path> Factorial.java
```

Documentación (y V)



The screenshot shows a web browser window titled "Generated Documentation (Untitled) - Google Chrome". The address bar shows the URL: `/srv/store/raid1/work/edu/PROG/modular/src/java/v2/u04/draft/doc/index.html`. The browser has tabs for "Generated Documentation (U x +", "Aplicaciones", "Bookmarks", "IES", "Dict", "Prime Music", and "PROG".

The documentation page has a sidebar on the left titled "All Classes" with a link to "Factorial". The main content area has a top navigation bar with tabs: "PACKAGE", "CLASS" (selected), "TREE", "DEPRECATED", "INDEX", and "HELP". Below this are links: "PREV CLASS", "NEXT CLASS", "FRAMES" (selected), and "NO FRAMES". There are also links for "SUMMARY: NESTED | FIELD | CONSTR | METHOD" and "DETAIL: FIELD | CONSTR | METHOD".

The main content area displays the "Class Factorial" documentation. It shows the inheritance hierarchy: `java.lang.Object` and `Factorial`. The class definition is shown as `public class Factorial extends java.lang.Object`. A description states: "La clase Factorial permite realizar el cálculo del factorial de un número natural [1,2,...]". The "Since:" field shows "01-01-2001".

Below the class definition is a "Constructor Summary" section. It has a tab "Constructors" (selected) and a sub-section "Constructor and Description" showing the `Factorial()` constructor.

Below the constructor summary is a "Method Summary" section. It has three tabs: "All Methods" (selected), "Static Methods", and "Concrete Methods". Below the tabs is a table with two columns: "Modifier and Type" and "Method and Description".

Modifier and Type	Method and Description
static long	<code>factLoop(int n)</code>

Depuración: introducción (I)


- La **depuración** es el procedimiento por el cual buscamos y eliminamos errores o defectos ("**bugs**") en nuestros programas para así obtener los resultados esperados
- Si bien ya se venía empleando desde los tiempos de Thomas Edison el término "*bug*" como sinónimo de "*error técnico*", o "*debugging*" en los inicios de la aeronáutica, popularmente se le atribuyó el término "*debugging*" en el campo de la computación a la investigadora y almirante norteamericana Grace Hooper en los años 40.
- Mientras estaba trabajando en un Mark II en la Universidad de Harvard, sus colaboradores detectaron una polilla en un relé impidiendo la operación normal del equipo, tras lo cual ella comentó que estaban haciendo un "*debugging*" del sistema

Depuración: introducción (II)

9/9

0800 Antan started
1000 " stopped - antan ✓
1300 (032) MP - MC ~~1.58264000~~ 1.58264000 { 1.2700 9.037 847 025
2.130476415 (3) 4.615925059(-2)
(033) PRO 2 2.130476415
conv 2.130676415
Relays 6-2 in 033 failed special speed test
in relay " 11.000 test.

1100 Relays changed
Started Cosine Tape (Sine check)
1525 Started Multy Adder Test.

1545  Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.
1630 Antan started.
1700 closed down.

Relay 337

Entrada del *log* del Mark II con la polilla (“*bug*”) encontrada (relé 70 del panel F) y la anotación:
“primer caso real de bicho encontrado”

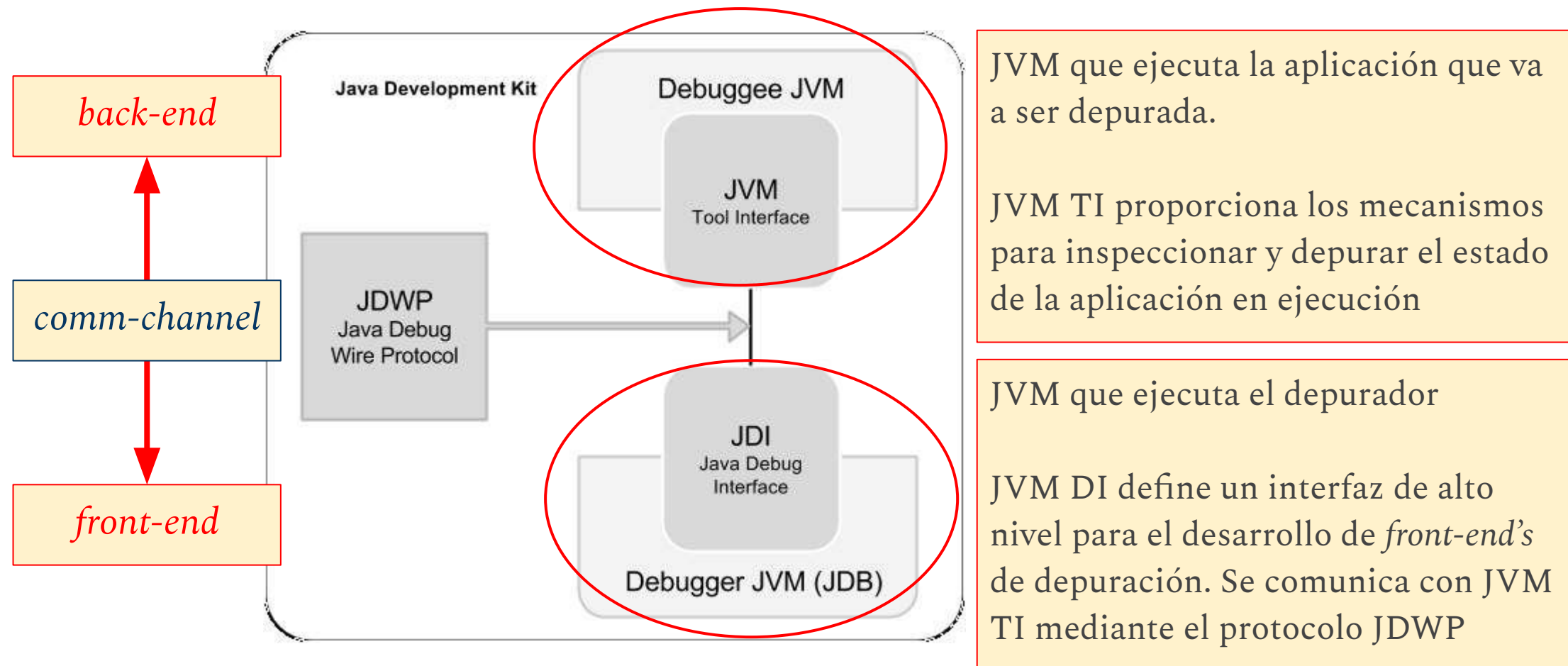
Depuración: introducción (y III)

- Existen diversas técnicas a la hora de depurar un programa informático. La “*clásica*” es la de insertar en el código una serie de “*print's*” que nos permitan ir haciendo un seguimiento de lo que sucede en el programa
- Este método, válido para programas simples, además de la sobrecarga de los constantes impresiones, hace difícil la depuración en sistemas complejos, distribuidos ó remotos. Aún así, es la base de los sistemas de **registro** o **log** de cualquier servicio o aplicación
- Los sistemas de depuración avanzados nos permitirán:
 - **stepping**, ejecución paso a paso del código
 - **breakpoints**, definición de puntos de interrupción del programa
 - **watchpoints**, examen de variables y expresiones en tiempo real
 - **exceptions**, definición de interrupciones en base a condiciones de error

Depuración: JDB. Introducción (I)

- El **JDB** (*Java Debugger*) es la herramienta del JDK que nos permite realizar la depuración de aplicaciones Java

❖ Arquitectura de JDB



Depuración: JDB. Introducción (II)

- JDB es una herramienta incluida en el JDK. Podemos comprobar su instalación y versión mediante el comando:

```
bowman@hal:~$ jdb -version  
This is jdb version 1.8 (Java SE version 1.8.0_201)
```

- La sintaxis de *jdb* es la siguiente:

jdb [opciones] [*clase*] [argumentos]

donde:

- *opciones*, son opciones de línea de comando del *jdb* (-D, -X, -classpath, -attach,...)
- *clase*, es la clase que queremos depurar
- *argumentos*, valores de entrada para el programa en ejecución

Depuración: JDB. Introducción (III)

- Algunas opciones de *jdb*:

opción	descripción
-help	Muestra la ayuda y las diferentes opciones del depurador
-sourcepath	Establece la ruta a las fuentes (por defecto: directorio actual)
-attach <dir>	Conecta el depurador a la VM en la dirección especificada
-listen <dir>	Espera por la VM en la dirección especificada para conectar el depurador
-launch	Lanza inmediatamente la aplicación depurada (sin comando run)
-listconnectors	Lista los conectores disponibles para esta JVM
-connect <conector>	Conecta con la JVM destino indicada
-tclient	Ejecuta la aplicación en el Java HotSpot VM cliente
-tserver	Ejecuta la aplicación en el Java HotSpot VM servidor

Depuración: JDB. Introducción (IV)

❖ Iniciar una Sesión JDB

- Existen diferentes maneras de iniciar una **sesión JDB**. Principalmente:
 - Iniciar la sesión JDB añadiendo la clase principal
 - Incluyendo JDB en una JVM en ejecución para iniciar la sesión

❏ Iniciar la sesión JDB añadiendo la clase principal

- Ejecutaremos el comando:

```
bowman@hal:~$ jdb <CLASSNAME>
```

- Suele ser la forma habitual de utilizar *jdb*. Este comando inicia una segunda JVM, carga la clase principal (método *main*) de nuestra aplicación (*CLASSNAME*) y se detiene antes de ejecutar la primera de sus instrucciones

Depuración: JDB. Introducción (y V)

❑ Incluir JDB en una JVM en ejecución

- Otro método consiste en conectar el depurador a una JVM que ya está en ejecución. Para ello, la JVM debe haber sido iniciada previamente con una serie de opciones para permitir la comunicación con el depurador.

Por ej:

```
bowman@hal:~$ java -Xdebug  
-Xrunjdwp:transport=dt_socket,address=ip:port,server=y,suspend=n <CLASSNAME>
```

- Posteriormente, podremos conectar JDB con la JVM mediante:

```
bowman@hal:~$ jdb -attach ip:port
```

- Fíjate que, en este caso, no indicamos el nombre de la clase pues JDB se conectará a una JVM ya en ejecución en lugar de iniciar una nueva

Depuración: JDB. Básico (I)

❖ Depurando con JDB

- Una vez iniciada una sesión JDB, se iniciará el *shell* o intérprete de *jdb* a través del que podremos introducir diferentes comandos para relizar la depuración de la aplicación
- Para ilustrar el proceso, emplearemos la siguiente clase:

```
//: Suma.java
public class Suma {
    public int suma(int x, int y) {
        int z = x + y;
        return z;
    }
    public static void main(String[] args) {
        int a = 5, b = 6;
        Suma ob = new Suma();

        int c = ob.suma(a, b);
        System.out.println("\nSuma: " + c);
    }
}
```


Depuración: JDB. Básico (II)

Run

- Una vez compilada la clase, iniciamos una sesión de JDB sobre ella:

```
bowman@hal:~$ javac Suma.java
bowman@hal:~$ jdb Suma
Initializing jdb ...
>
```

- En este momento la JVM se habrá iniciado, cargado la clase Suma y se detendrá a la espera de que introduzcamos algún comando (>).

Introducimos el comando *run*, que provocará la ejecución de la clase:

```
bowman@hal:~$ jdb Suma
Initializing jdb ...
> run
run Suma
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started:
Suma: 11

The application exited
```

Depuración: JDB. Básico (III)

❏ Breakpoints

- Habitualmente, introduciremos *breakpoints* para forzar la detención del programa en una línea del mismo, lo que nos permitirá inspeccionar el estado de las variables y objetos en ese punto del programa. La sintaxis del comando es:

```
stop at <class name>:<Line no>  
stop in <class name>.<Method name>
```

- Por ejemplo:
 - stop at MyClass:22 (*breakpoint* en la línea 22 de MyClass)
 - stop in MyClass.myMethod (*breakpoint* al inicio del método myMethod de MyClass)
 - stop in MyClass.<init> (<init> identifica al constructor de MyClass)
- Para eliminar un *breakpoint* utilizaremos el comando *clear*

Depuración: JDB. Básico (IV)

- Siguiendo con nuestro ejemplo, vamos a lanzar una nueva sesión de depuración pero vamos a detener el programa al inicio del método *main()*

1) Iniciamos la sesión y establecemos un *breakpoint* en el método *main()*

```
bowman@hal:~$ jdb Suma
Initializing jdb ...
> stop in Suma.main
Deferring breakpoint Suma.main.
It will be set after the class is loaded.
```

2) Iniciamos la ejecución de la aplicación:

```
> run
run Suma
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint Suma.main
Breakpoint hit: "thread=main", Suma.main(), line=9 bci=0
9      int a = 5, b = 6;
main[1]
```

Se detendrá en la primera sentencia de *main()*: `int a = 5, b = 6;`

Depuración: JDB. Básico (V)

3) Para continuar la ejecución tras una detención, usaremos el comando *cont*

```
main[1] cont  
> Suma: 11  
  
The application exited
```

Al no existir *breakpoints* adicionales, el programa continuará su ejecución normal y finalizará

- Cada vez que detengamos nuestro programa, tendremos la opción de hacerlo avanzar instrucción a instrucción (*stepping*). Dispondremos de los siguientes comandos:
 - *step*, avanza a la próxima sentencia en ejecución
 - *list*, examina el código
 - *cont*, continúa la ejecución normal

Depuración: JDB. Básico (VI)

- Siguiendo con nuestro ejemplo, vamos a detener el programa al inicio del método *main()* y continuar paso a paso

1) Iniciamos la sesión y establecemos un *breakpoint* en el método *main()*

```
bowman@hal:~$ jdb Suma
Initializing jdb ...
> stop in Suma.main
Deferring breakpoint Suma.main.
It will be set after the class is loaded.
```

2) Iniciamos la ejecución de la aplicación:

```
> run
run Suma
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint Suma.main
Breakpoint hit: "thread=main", Suma.main(), line=9 bci=0
9      int a = 5, b = 6;
main[1]
```

Se detendrá en la primera sentencia de *main()* (línea 9) `int a = 5, b = 6;`

Depuración: JDB. Básico (VII)

3) Vamos a avanzar una sentencia con *step*...

```
main[1] step
>
Step completed: "thread=main", Suma.main(), line=10 bci=5
10      Suma ob = new Suma();
main[1]
```

La JVM ejecuta la sentencia anterior y avanza a la siguiente (*línea 10*), donde se volverá a detener antes de ejecutarla: `Suma ob = new Suma();`

4) En este punto, podríamos listar el código mediante *list*...

```
main[1] list
6      }
7
8      public static void main(String[] args) {
9          int a = 5, b = 6;
10 =>    Suma ob = new Suma();
11
12        int c = ob.suma(a, b);
13        System.out.println("\nSuma: " + c);
14    }
15    }
main[1]
```

Depuración: JDB. Básico (VIII)

- En la captura anterior se puede observar la marca => que nos indica la posición actual del control del programa

5) Finalmente, continuamos la ejecución mediante el comando *cont*

```
main[1] cont  
> Suma: 11  
  
The application exited
```

- Cuando estamos haciendo *stepping* y la siguiente instrucción es una llamada a un método, podemos hacer:
 - *step*, entra en el método y se detiene (*step in*)
 - *next*, ejecuta el método sin detenerse en él (*step over*)
 - *step up*, ejecuta el método actual hasta retornar a su llamada

Depuración: JDB. Básico (IX)

- En nuestro ejemplo, tras detener la ejecución en la línea 12:

```
main[1] list
6      }
7
8      public static void main(String[] args) {
9          int a = 5, b = 6;
10         Suma ob = new Suma();
11
12 =>     int c = ob.suma(a, b);
13         System.out.println("\nSuma: " + c);
14     }
15 }
main[1]
```

- En esta línea hay una llamada al método suma(). Tenemos dos opciones:

➤ *Opción 1 (step)*

Saltará a la primera instrucción del método y se detendrá ahí.

Podremos continuar paso a paso por el mismo hasta el final, o emplear *step up* para finalizar el método y volvernos a detener en la llamada

Depuración: JDB. Básico (X)

- Tras hacer *step*...

```
main[1] step
>
Step completed: "thread=main", Suma.suma(), line=4 bci=0
4      int z = x + y;
main[1]
```

Observamos como nos encontramos en la primera sentencia dentro del método `Suma.suma()` (línea 4)

- Vamos a continuar paso a paso desde ahí...

```
main[1] step
>
Step completed: "thread=main", Suma.suma(), line=5 bci=4
5      return z;
main[1] step
>
Step completed: "thread=main", Suma.main(), line=12 bci=19
12     int c = ob.suma(a, b);
main[1]
```

Hemos finalizado el método y retornado a la llamada (línea 12)

Depuración: JDB. Básico (XI)

➤ Opción 2 (*next*)

- Frecuentemente, lo único que nos interesa es que se ejecute el método pero sin tener que recorrerlo paso a paso. En esos casos, usaremos el comando *next*...

```
main[1] next
>
Step completed: "thread=main", Suma.main(), line=13 bci=21
13      System.out.println("Suma: " + c);
main[1]
```

Observamos que se ejecutó el método `suma()`, sin detenernos en él, y el programa se para en la siguiente sentencia a la llamada (línea 13)

- Recuerda que si entramos en un método empleando *step*, siempre podremos forzar que se ejecuten el resto de sus instrucciones sin deternos hasta salir del mismo utilizando *step up*

Depuración: JDB. Básico (XII)

❖ Inspeccionando variables

- Sin duda, la tarea más importante que realizamos durante la depuración de un programa es la inspección del estado actual de variables y objetos.
- Vamos a depurar las siguientes clases:

```
//: Coord.java
public class Coord {
    double x;
    double y;
    Coord(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public double dist(Coord c) {
        double z = Math.sqrt(Math.pow(x-c.x,2) +
                                Math.pow(y-c.y,2));
        return z;
    }
}
```

```
//: CoordTest.java
public class CoordTest {
    public static void main(String[] args) {
        Coord c1, c2;
        c1 = new Coord(-1, -1);
        c2 = new Coord(2, 2);

        System.out.println("Distancia c1-c2: " +
                            c1.dist(c2));
    }
}
```

Depuración: JDB. Básico (XIII)

- Compilaremos ambas clases con la opción **-g** del compilador, lo cual nos permitirá inspeccionar las variables locales desde el depurador

```
bowman@hal:~$ javac -g Coord*.java
```

- Iniciamos una sesión de JDB (asignando la clase principal *TestCoord*) y situamos un *breakpoint* en el constructor de *Coord* (stop in Coord.<init>)

```
bowman@hal:~$ jdb CoordTest
Initializing jdb ...
> stop in Coord.<init>
Deferring breakpoint Coord.<init>.
It will be set after the class is loaded.
>
```

- Si ejecutamos ahora la aplicación (comando `run`), se iniciará el método *main()* de *CoordTest*. Al ejecutar la sentencia: `c1 = new Coord(-1, -1);` se invocará el constructor de la clase *Coord* y la aplicación se detendrá (Podríamos haber situado el *breakpoint* en esa línea y hacer `step`)

Depuración: JDB. Básico (XIV)

- Para inspeccionar las variables y objetos podemos usar los comandos:
 - *print*, muestra el valor de variables locales y objetos
 - *dump*, como *print* pero, para objetos, muestra su estructura interna
 - *locals*, muestra los valores de las variables locales y argumentos
 - *set <nom>=<expr>*, asigna un nuevo valor al atributo/variable/array[i]

```
main[1] run
run CoordTest
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint Coord.<init>

Breakpoint hit: "thread=main", Coord.<init>(), line=4 bci=0
4      Coord(double x, double y) {

main[1] locals
Method arguments:
x = -1.0
y = -1.0
Local variables:
main[1] print x
x = -1.0
```

Depuración: JDB. Básico (XV)

```
main[1] print this
this = "Coord@77afea7d"
main[1] dump this
this = {
  x: 0.0
  y: 0.0
}
```

- Observamos los valores de los parámetros del constructor (x=-1, y=-1), así como la referencia del nuevo objeto (Coord@77afea7d) y el valor de sus atributos que, lógicamente, están inicializados a cero.
- Avanzamos y vemos como los atributos del objeto se van actualizando...

```
main[1] step
>
Step completed: "thread=main", Coord.<init>(), line=5 bci=4
5      this.x = x;
main[1] step
>
Step completed: "thread=main", Coord.<init>(), line=6 bci=9
6      this.y = y;
main[1] print this.x
this.x = -1.0
```

Depuración: JDB. Básico (y XVI)

- Finalizamos la ejecución del constructor y volvemos al método *main()*...

```
main[1] step up
>
Step completed: "thread=main", CoordTest.main(), line=4 bci=13
4          c1 = new Coord(-1, -1);
main[1] step
>
Step completed: "thread=main", CoordTest.main(), line=5 bci=14
5          c2 = new Coord(2, 2);
```

- Si observamos el valor del objeto **c1** vemos que se ha inicializado con los valores que se pasaron en el constructor (-1, -1)

```
main[1] print c1
c1 = "Coord@77afea7d"
main[1] dump c1
c1 = {
  x: -1.0
  y: -1.0
}
```

- Fíjate como la referencia que almacena la variable **c1** (Coord@77afea7d) es la misma que obtuvimos al inspeccionar la variable **this** en el constructor

Depuración: JDB. Remoto (I)

- En esta sección vamos a analizar como realizar una sesión de depuración remota, conectando JDB a una JVM ya **en ejecución**. Esto nos permitirá depurar aplicaciones en ejecución, no sólo en nuestro mismo equipo, si no también en **equipos remotos**
- Este mismo método es el que deberíamos emplear para aplicaciones **interactivas**, de forma que podamos independizar la entrada de comandos de *jdb* de la entrada del usuario en la aplicación en ejecución
- Veremos también un nuevo tipo de *breakpoints*, los de tipo **Exception**. Las excepciones son condiciones de error que se producen durante la ejecución de nuestro programa y que normalmente supondrán su finalización abrupta
- Para realizar los ejemplos, vamos a utilizar el siguiente código:

Depuración: JDB. Remoto (II)

```
//: RemoteDbg.java
import java.util.Scanner;

class RemoteDbg {
    public static void main(String[] args) {
        int a, b;
        Scanner cin = new Scanner(System.in);

        while(true) {
            System.out.print("\nDame un número entero: ");
            a = cin.nextInt();
            System.out.print("Dame otro número entero: ");
            b = cin.nextInt();
            System.out.println(a + "/" + b + " = " + (a/b));
        }
    }
}
```

- La aplicación se mantendrá en un **bucle infinito** (`while(true)`) solicitando del usuario la entrada de dos números y mostrando su división

Depuración: JDB. Remoto (III)

- Ejecutemos la aplicación y veamos que ocurre cuando el valor del segundo número (divisor) es cero o introducimos un valor no válido...

```
bowman@hal:~$ java RemoteDbg
Dame un número entero: 4
Dame otro número entero: 3
4/3 = 1

Dame un número entero: 5
Dame otro número entero: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at RemoteDbg.main(RemoteDbg.java:14)
```

- Si introducimos 0 como valor del divisor, se producirá una excepción del tipo *java.lang.ArithmeticException*
- De igual modo, si introducimos un valor no válido (un texto, por ejemplo), la excepción será del tipo *java.util.InputMismatchException*
- En ambos casos el programa finalizará abruptamente

Depuración: JDB. Remoto (IV)

- Vamos a ver cómo realizar la depuración remota de dicha aplicación...
- En primer lugar, lanzaremos la aplicación, pero con las opciones de JVM necesarias que nos permitan su depuración remota. Debemos tener en cuenta que la ejecución de una JVM en modo depuración va a afectar al rendimiento de la misma (esto puede ser especialmente crítico en sistemas en producción)

```
bowman@hal:~$ java -Xdebug  
-Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n RemoteDbg  
Listening for transport dt_socket at address: 8000  
  
Dame un número entero:
```

- La opción *-Xdebug* habilita el modo depuración para la JVM HotSpot de Oracle. Ten en cuenta que, al tratarse de un *X-argument*, su uso no está estandarizado y podría ser diferente en otras implementaciones de la máquina virtual de Java

Depuración: JDB. Remoto (V)

- La opción *-Xrunjdwp* establece los parámetros de conexión (**transporte** y **conector**) con la JVM desde donde realizaremos la depuración:
 - *transport*, puede tomar dos valores:
 - *dt_socket*, transporte mediante un *socket TCP*
 - *dt_shmem*, en el que se emplea un esquema de *memoria compartida* en la comunicación. Sólo tiene sentido si se realiza la depuración desde la misma máquina
 - *address*, *[ip:port]* a la que se conectará el depurador. Logicamente, la especificación del conector empleado, dependerá del tipo de transporte definido (en este caso, *socket*)
- La opción *-server=y* establece que debe ser JDB el que actuará como *cliente*, conectándose a esta JVM (que actúa como *servidor*).
- La opción *-suspend* establece si se quiere suspender la JVM hasta que el depurador se conecte. En este caso, no (=n) y la aplicación se iniciará

Depuración: JDB. Remoto (VI)

- Llegados a este punto, tendremos la aplicación remota en ejecución sobre una JVM con la depuración activada
- Si tenemos la opción de observar los servicios lanzados en modo escucha (*listen*) en la máquina, observaremos un proceso (la JVM) escuchando en el puerto 8000 (de *localhost*), listo para aceptar conexiones:

```
bowman@hal:~$ netstat -plant4 | grep 8000
tcp        0      0 127.0.0.1:8000      0.0.0.0:*           ESCUCHAR    9716/java
```

NOTA: alternativamente, podemos usar el comando: `ss -ltp | grep 8000`

- Sólo nos queda lanzar la JVM con el depurador...
NOTA: en este caso, ambas JVM estarán sobre la misma máquina o no nos podremos conectar (el servicio se lanzó para *localhost*). En máquinas diferentes, deberíamos haber añadido la dirección IP de la máquina remota en el momento de lanzar su JVM (por ejemplo, *address=192.168.1.25:8000*).

Depuración: JDB. Remoto (VII)

- Iniciamos la sesión remota de JDB...

```
bowman@hal:~$ jdb -attach localhost:8000 -sourcepath ~/src/java/  
Set uncaught java.lang.Throwable  
Set deferred uncaught java.lang.Throwable  
Initializing jdb ...  
>
```

- *-attach*, nos permitirá indicar el conector de la JVM remota. En este caso, la dirección IP y número de puerto del servicio remoto
- *-sourcepath*, nos permite establecer la ruta(s) a los archivos fuente

Alternativamente: `jdb -connect com.sun.jdi.SocketAttach:hostname=localhost,port=8000`

- Si observamos las conexiones establecidas por la máquina local (o en la remota), veremos una conexión TCP abierta entre ambas JVM

```
bowman@hal:~$ netstat -plant4 | grep 8000  
tcp        0      0 127.0.0.1:8000      127.0.0.1:57766    ESCUCHAR   9716/java
```

Alternativamente, podemos usar el comando: `ss -tp | grep 8000`

Depuración: JDB. Remoto (VIII)

- Vamos a establecer un *breakpoint* cuando se produzca una excepción *java.lang.ArithmeticException*

```
bowman@hal:~$ jdb -attach localhost:8000 -sourcepath ~/src/java/  
Set uncaught java.lang.Throwable  
Set deferred uncaught java.lang.Throwable  
Initializing jdb ...  
> catch java.lang.ArithmeticException  
Set all java.lang.ArithmeticException  
>
```

- Veamos lo que ocurre cuando en el sistema depurado introducimos un valor cero como divisor...

```
Dame un número entero: 5  
Dame otro número entero: 0
```

- El programa remoto se detendrá y el depurador tomará el control. Ahora podremos avanzar paso a paso, inspeccionar la variables, ...

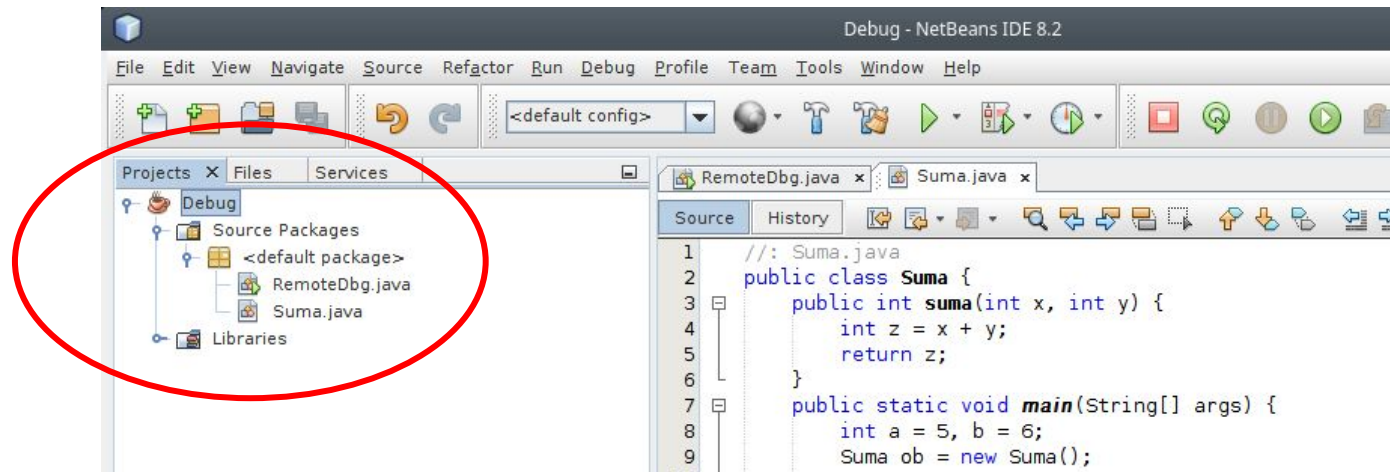
Depuración: JDB. Remoto (y IX)

```
> catch java.lang.ArithmeticException
Set all java.lang.ArithmeticException
>
Exception occurred: java.lang.ArithmeticException (uncaught)
Exception occurred: java.lang.ArithmeticException (uncaught)"thread=main",
RemoteDbg.main(), line=14 bci=44
14          System.out.println(a + "/" + b + " = " + (a/b));

main[1] list
10
11          System.out.print("Dame otro número entero: ");
12          b = cin.nextInt();
13
14 =>          System.out.println(a + "/" + b + " = " + (a/b));
15          }
16      }
17  }
main[1] locals
Method arguments:
args = instance of java.lang.String[0] (id=803)
Local variables:
cin = instance of java.util.Scanner(id=804)
a = 5
b = 0
main[1]
```


Depuración: usando Netbeans (I)

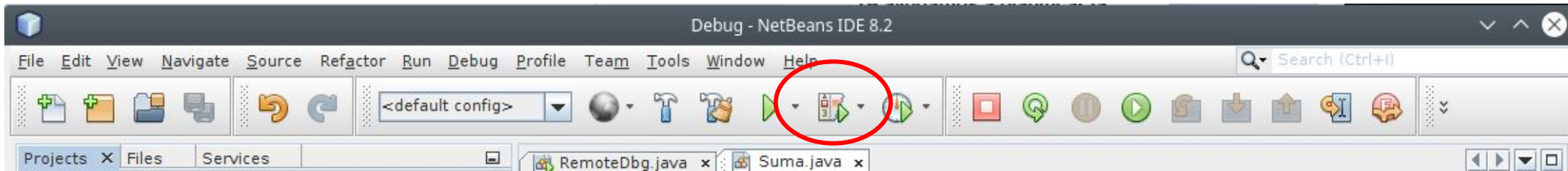
- En general, los IDE's nos facilitan el proceso de depuración o, al menos, realizar la depuración desde un entorno más “amigable”
- Básicamente, las operaciones a realizar serán las mismas: situar *breakpoints*, capturar *excepciones*, ejecutar paso-a-paso, consultar o modificar variables o atributos...
- Para realizar los ejemplos, crearemos un nuevo proyecto en Netbeans al que añadiremos dos de las clases empleadas anteriormente: *Suma* y *RemoteDbg*



Depuración: usando Netbeans (II)

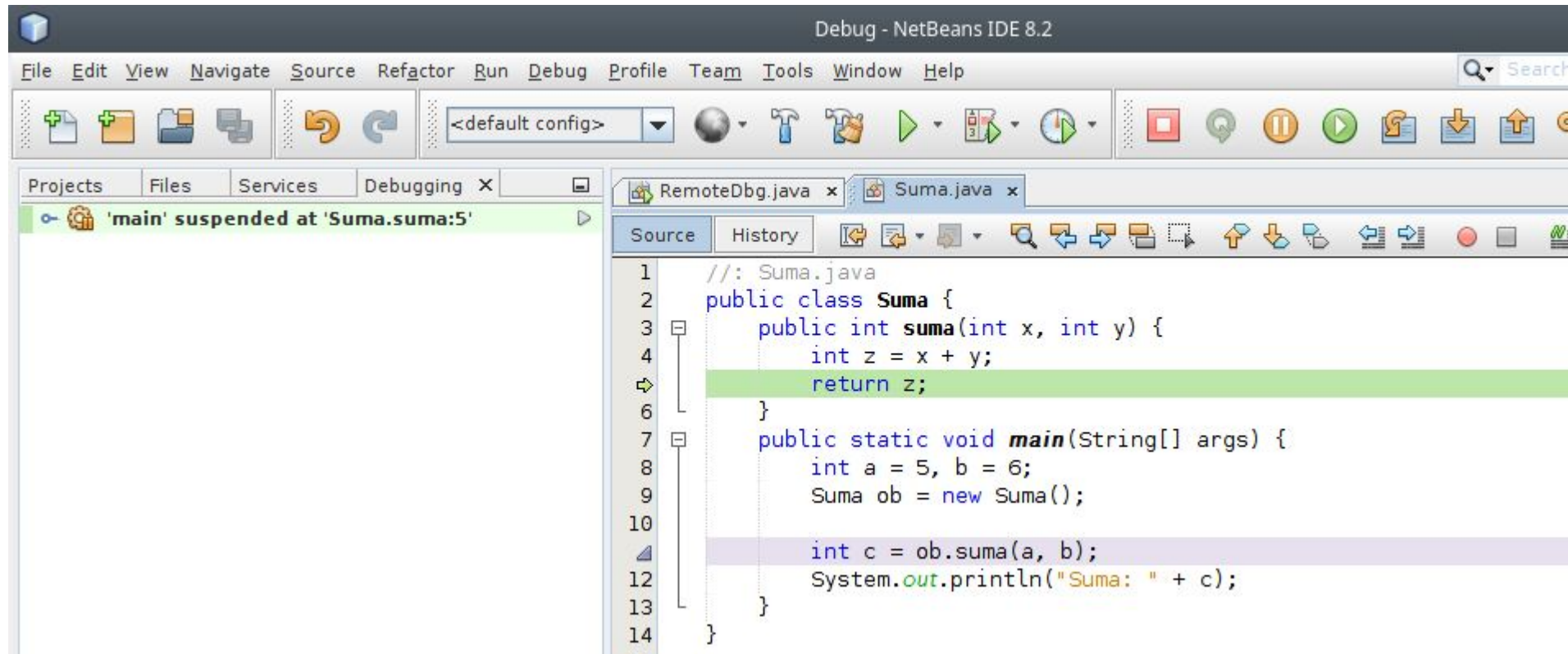
❖ Iniciando sesión de depuración

- Existen varias formas de iniciar un programa en modo depuración (luego veremos como conectarnos a un servicio remoto para su depuración)
- Pulsando sobre el botón correspondiente de la botonera, se lanzará el proyecto actual en modo depuración (Ctrl + F5)



- Alternativamente, desde el menú de depuración, tenemos otras opciones que nos permiten:
 - *Step Into (F7)*, inicia la depuración y se detiene al comienzo de *main()*
 - *Run to Cursor (F4)*, inicia la depuración y se detiene en el momento que el programa alcance la instrucción donde se encuentra el cursor

Depuración: usando Netbeans (III)

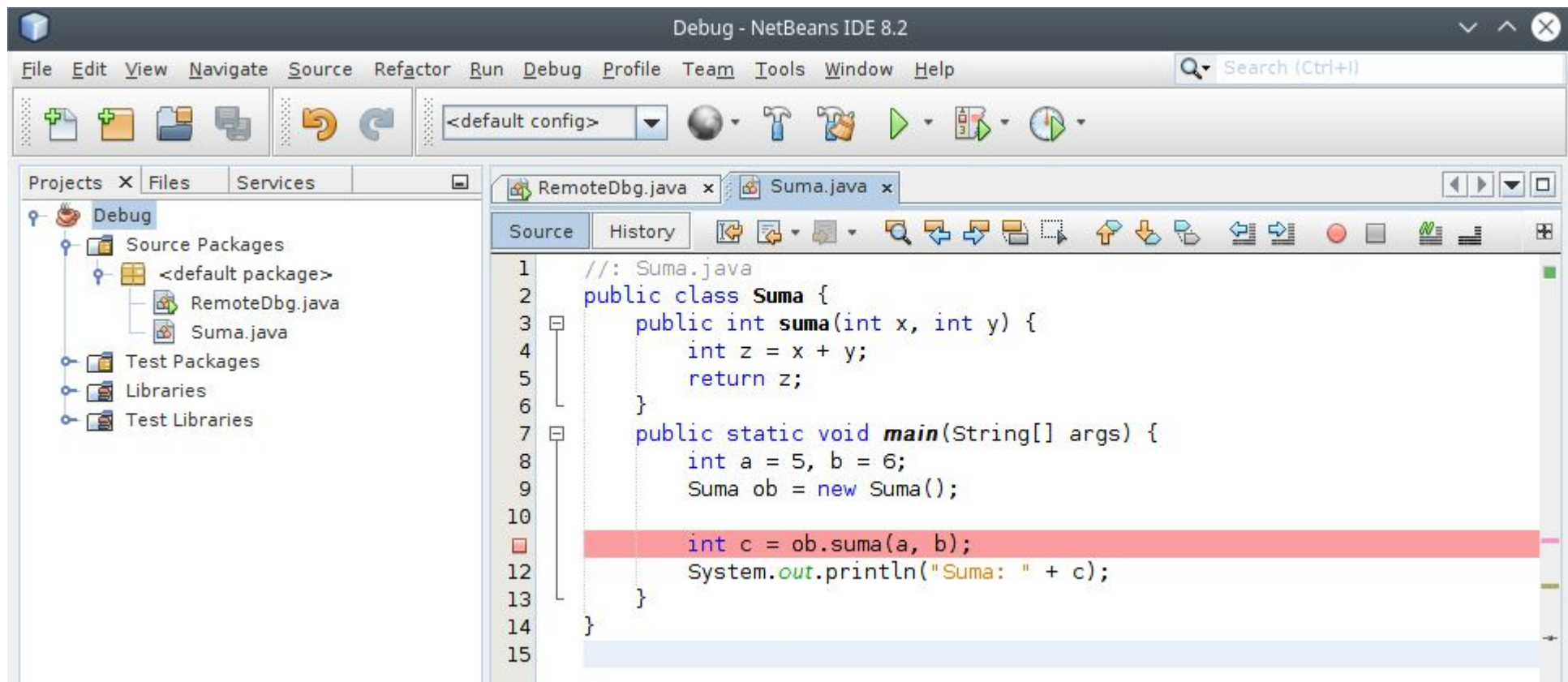


- Al detenerse, netbeans nos muestra resaltada en verde la sentencia donde se detuvo el depurador (en gris claro también vemos resaltada la llamada al método en el que nos encontramos)

Depuración: usando Netbeans (IV)

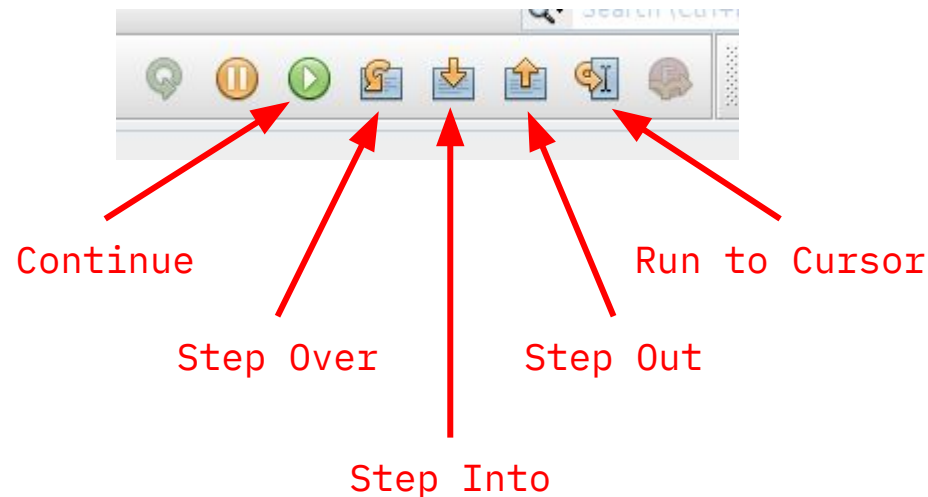
❖ *Breakpoints y Stepping*

- Para situar un *breakpoint* en una línea determinada, basta con clicar sobre el número de línea correspondiente en el editor. Sobre éste aparecerá un cuadrado rojo y la línea en cuestión se resaltará también en rojo



Depuración: usando Netbeans (V)

- Cuando el depurador alcance cualquiera de los *breakpoints* que hayamos establecidos, el programa se detendrá
- Cuando el depurador alcance cualquiera de los *breakpoints* que hayamos establecidos, el programa se detendrá. A partir de ahí, podremos seguir paso-a-paso ejecutando instrucciones y entrando en aquellos métodos que nos interese:

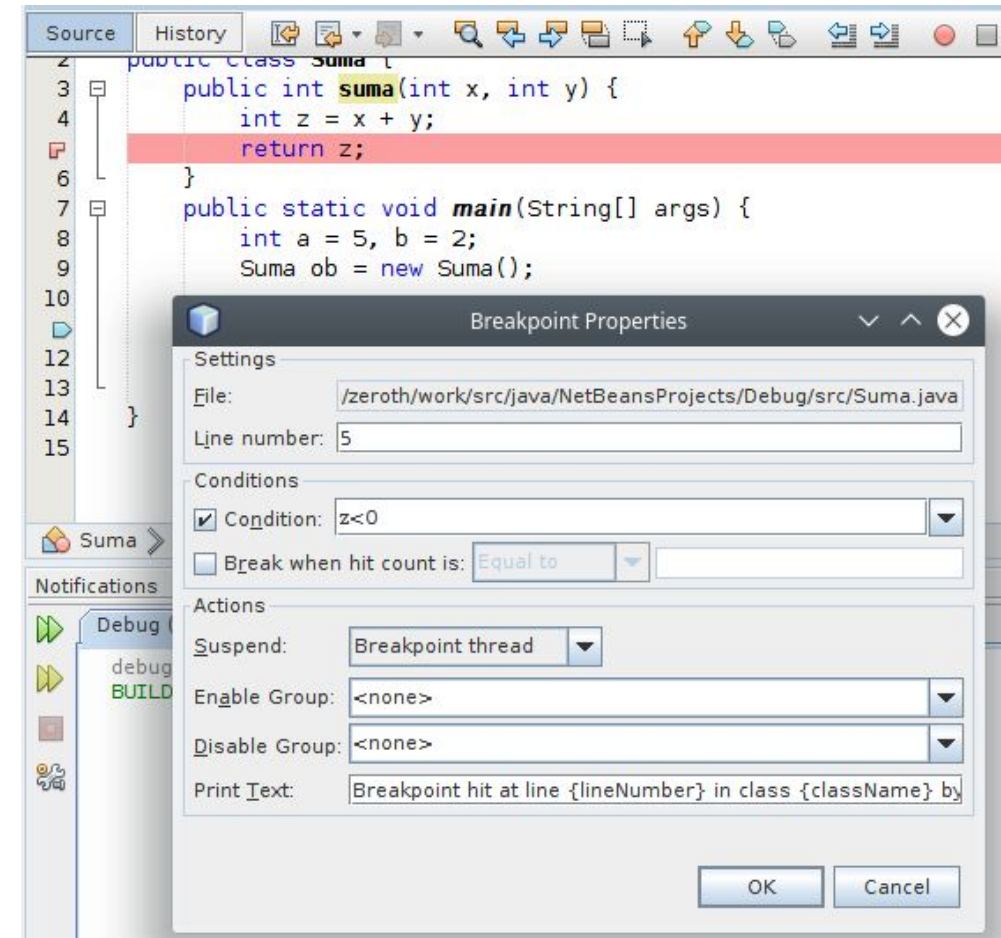


Depuración: usando Netbeans (VI)

- *Step Into (F7)*, ejecuta una línea de código. Si es un método, se detiene en su primera instrucción
- *Step Over (F8)*, ejecuta una línea de código. Si es un método, lo ejecuta sin detenerse en él
- *Step Out (Ctrl + F7)*, ejecuta una línea de código. Si estamos un método, ejecuta todas las sentencias que queden en el método y se detiene en la llamada al mismo (sale del método tras ejecutarlo)
- *Run to Cursor (F4)*, se ejecuta el programa desde la instrucción actual hasta la instrucción donde se encuentra el cursor
- *Continue (F5)*, continúa la ejecución de programa hasta su finalización o hasta el siguiente *breakpoint*
- *Finish Debugger Session (Shift + F5)*, finaliza la sesión de depuración

Depuración: usando Netbeans (VII)

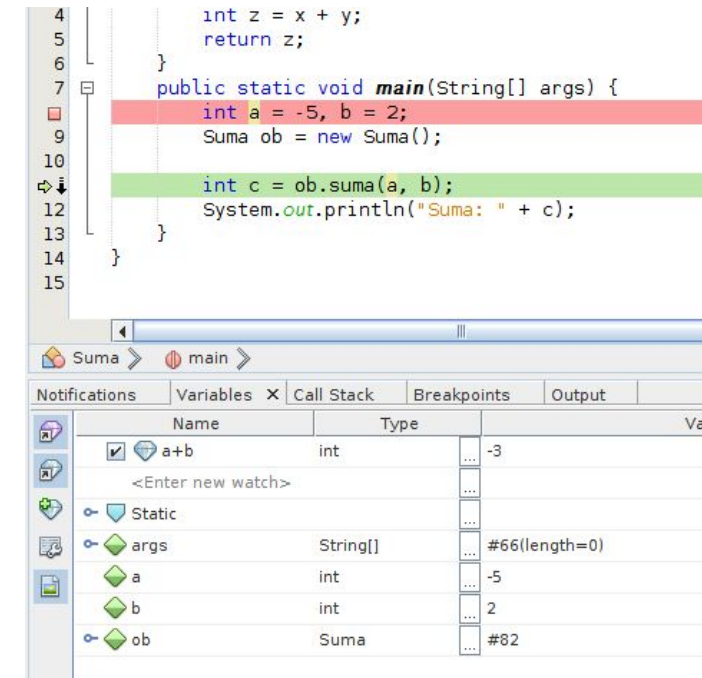
- Podemos crear también *breakpoints condicionales*, es decir, *breakpoints* que detendrán el programa cuando se dé una determinada circunstancia
- Para definir dicha condición, tras establecer el *breakpoint*, clicamos con el botón derecho sobre la marca y abrimos su diálogo de propiedades
- La condición puede ser el resultado de evaluar determinada expresión o que la sentencia se haya ejecutado un número determinado de veces
- En el ejemplo, se establece como condición que el valor de retorno del método sea negativo ($z < 0$)



Depuración: usando Netbeans (VIII)

❖ *Inspeccionando variables*

- Una vez se ha detenido el programa, podremos evaluar expresiones y monitorizar (y modificar) el valor de las variables, atributos,...
- En el panel inferior, se nos mostrarán varias pestañas desde las que podremos ver la pila actual de llamadas (*Call Stack*) o acceder al contenido de variables y objetos (*Variables*)
- En el panel *Variables* se nos mostrarán las diferentes variables locales y objetos dentro del alcance actual. La columna *Value*, a parte de mostrarnos el valor correspondiente, nos permitirá su edición. Además, podremos crear *watches*. Estos son variables o expresiones que se añaden al panel para monitorizar su valor



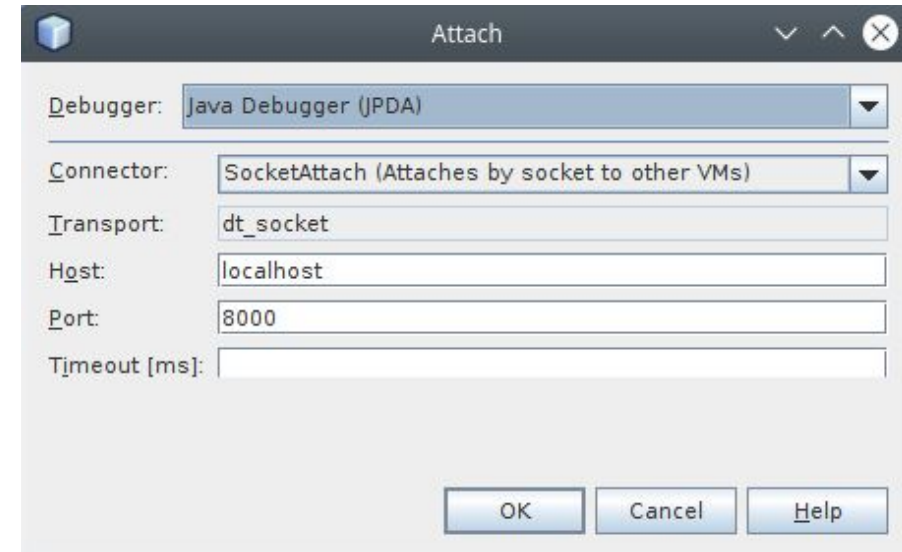
Depuración: usando Netbeans (IX)

❖ *Depuración remota*

- A modo de ejemplo, vamos a volver a lanzar la clase RemoteDbg desde una JVM con la depuración activada:

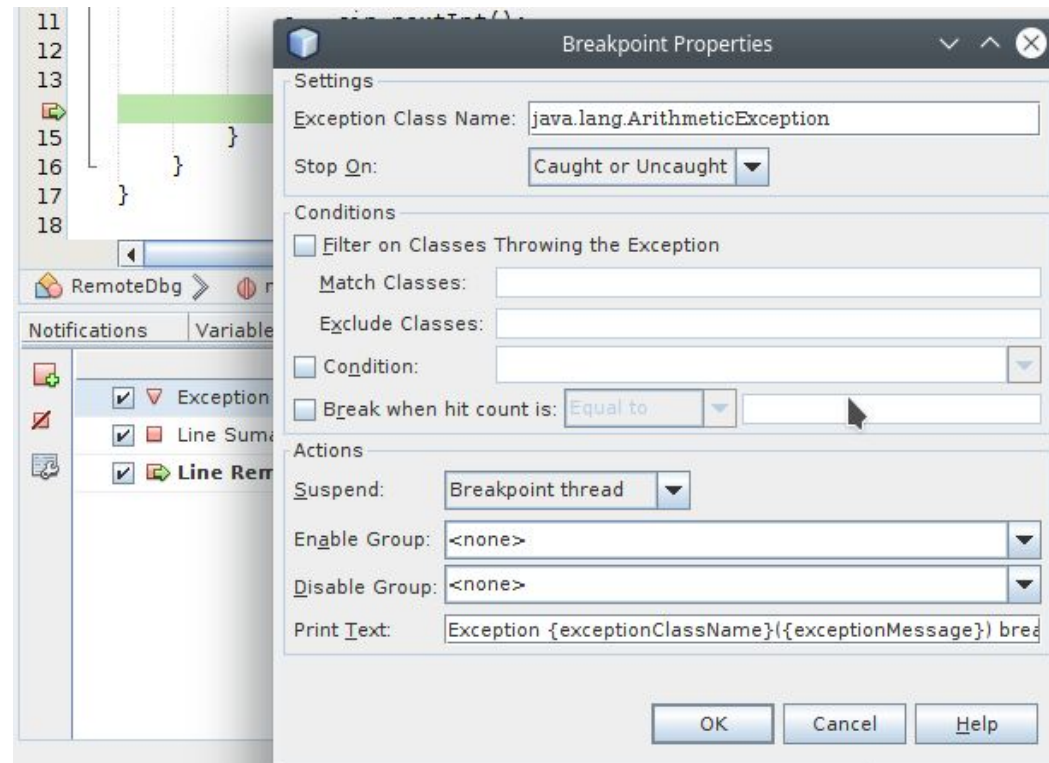
```
bowman@hal:~$ java -Xdebug  
-Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n RemoteDbg  
Listening for transport dt_socket at address: 8000  
  
Dame un número entero:
```

- Vamos a conectarnos a dicha JVM con el depurador de netbeans. Para ello, desde el menú *Debug*, clicamos en *Attach Debugger*. Seleccionamos JPDA como depurador y las opciones de transporte, host y puerto correspondientes (*dt_socket*, *localhost* y *8000*)



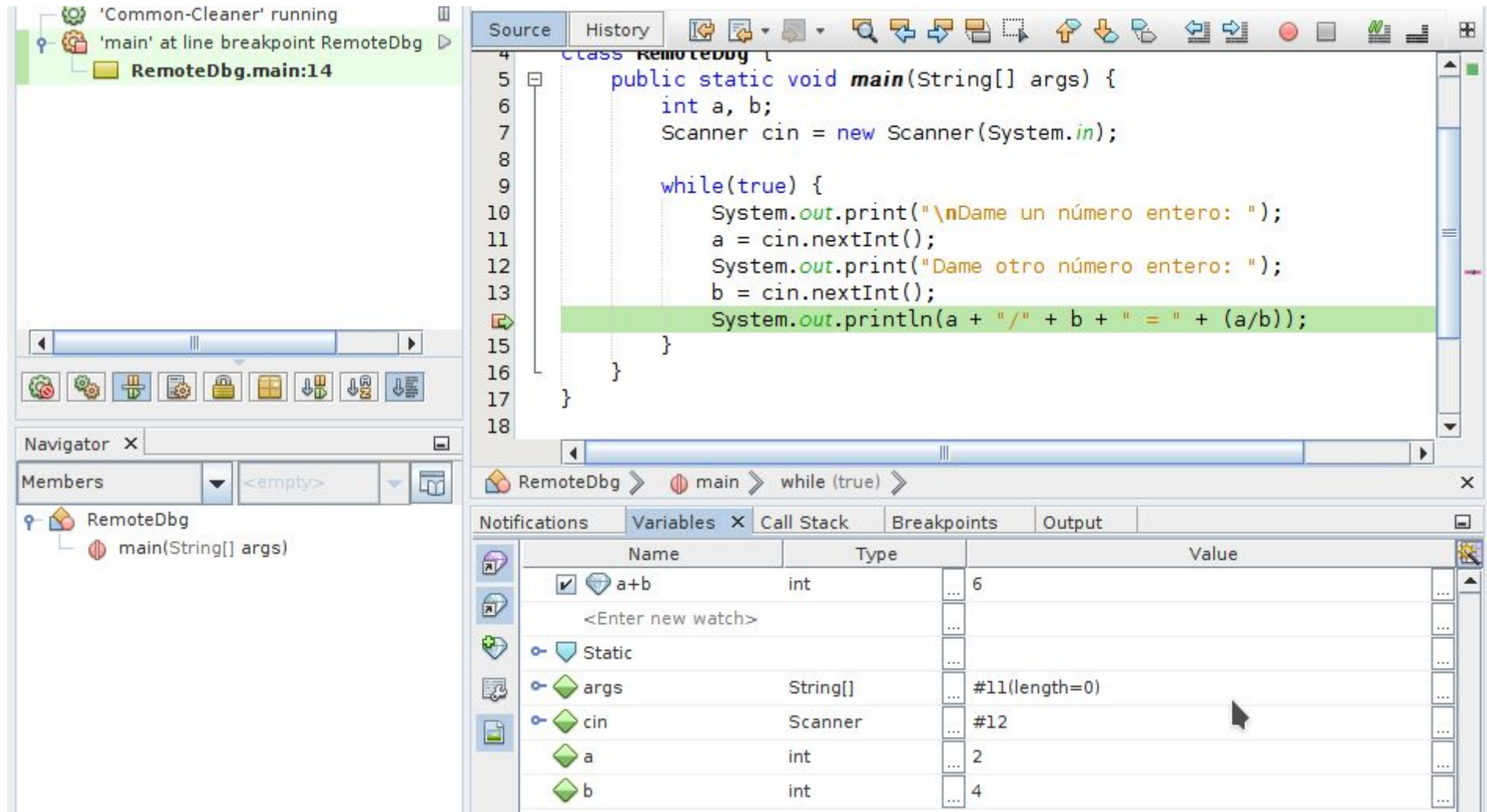
Depuración: usando Netbeans (X)

- Si tenemos acceso al código, podemos añadir *breakpoints* en cualquier línea que nos interese y detener ahí el programa
- Además, desde el **panel *Breakpoints***, podemos crear *breakpoints* basados en la captura de *excepciones*. Por ejemplo, podríamos crear una excepción del tipo *java.lang.ArithmeticException* como hicimos anteriormente:



Depuración: usando Netbeans (y XI)

- Una vez detenido el programa remoto, tendremos acceso a su conjunto de variables, objetos,... y podremos continuar paso a paso



The screenshot shows the NetBeans IDE interface during a remote debug session. The main editor displays the source code of a Java class named `RemoteDbg`. The code includes a `main` method that takes a `String[] args` array and uses a `Scanner` to read two integers, `a` and `b`, from the standard input. The code then prints the sum of `a` and `b` using `System.out.println`. A breakpoint is set at line 14, which is highlighted in green. The left sidebar shows the project structure with 'RemoteDbg' and 'main' method. The bottom right pane shows the 'Variables' tab, which displays a table of the current state of the program's variables.

Name	Type	Value
<input checked="" type="checkbox"/> a+b	int	6
<Enter new watch>		
<input checked="" type="checkbox"/> Static		
<input checked="" type="checkbox"/> args	String[]	#11(length=0)
<input checked="" type="checkbox"/> cin	Scanner	#12
<input checked="" type="checkbox"/> a	int	2
<input checked="" type="checkbox"/> b	int	4

JUnit: introducción (I)

- En general, toda pieza de software debe ser sometida a toda una batería de pruebas antes de su paso a producción con objeto de validar que su funcionamiento va a ser el esperado
- Existen numerosos tipos de pruebas que podemos categorizar como *funcionales* (relativas a acciones o funciones del código) o *no-funcionales* (relativas al comportamiento o restricciones de diseño de la aplicación)

funcionales	no-funcionales
<ul style="list-style-type: none">● Unit Testing● Integration Testing● System Testing● Sanity Testing● Smoke Testing● Interface Testing● Regression Testing● Beta/Acceptance Testing	<ul style="list-style-type: none">● Performance Testing● Load Testing● Stress Testing● Volume Testing● Security Testing● Compatibility Testing● Install Testing● Recovery Testing● Reliability Testing● Usability Testing● Compliance Testing● Localization Testing

referencia: https://en.wikipedia.org/wiki/Software_testing

JUnit: introducción (II)

- En nuestro caso, nos centraremos en las **pruebas unitarias** o *unit testing*, que nos permiten comprobar el correcto funcionamiento de una unidad de código (función en prog. estructurada, clase en OOP)
- Es decir, las pruebas unitarias nos permiten comprobar por separado que cada unidad funcione correctamente
- Además de comprobar la corrección del código, se verificarán los nombres y tipos de los métodos y sus parámetros
- En general, una prueba unitaria debe cumplir que es:
 - **automatizable**, conducida sin intervención humana
 - **repetible**, lo que facilita las *pruebas de regresión* tras cambios
 - **completa**, en cuanto al alcance en la unidad testeada
 - **independiente**, respecto a otras pruebas
- Existen numerosas herramientas para facilitar el diseño y ejecución de pruebas unitarias para distintos lenguajes: JUnit, TestNG, pytest,...

JUnit: introducción (y III)

- JUnit es un *framework* para la realización de pruebas unitarias en Java
- Es miembro de la familia de *frameworks* **xUnit** que se iniciaron con SUnit
- Se distribuye como un par de archivos *.jar* que habrá que añadir al *classpath*. La jerarquía de clases se encuentran bajo el paquete *org.junit*
- JUnit se integra perfectamente con numerosos IDE (Netbeans, IntelliJ, eclipse,...) que nos facilitarán la creación y ejecución de los *tests*
- El trabajo con JUnit se basa en la creación de *TestCase*. Estas serán clases Java en las que definiremos una serie de métodos que se ejecutarán en secuencia para probar el funcionamiento de alguna clase de nuestra aplicación. Cada uno de estos métodos de prueba irá precedido de una *anotación JUnit* para definir dicha secuencia de ejecución
- Podemos crear baterías de pruebas en JUnit agrupando diferentes *TestCase* en un *TestSuite*

JUnit: descarga e instalación

- Actualmente, muchos IDE ya incluyen JUnit en su instalación (por ejemplo, Netbeans a partir de la versión 7). Por lo tanto, no necesitaremos instalar nada para crear y ejecutar tests desde ellos
- Las distribuciones Linux suelen incluir JUnit en sus repositorios
- Podemos descargar los *.jar* de JUnit 4 desde junit.org/junit4
 - *junit.jar*
 - *hamcrest-core.jar*
- En caso de usar *maven* para la creación de proyectos, podemos añadir una nueva dependencia en el **pom.xml** (4.13 es la última versión estable):

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13</version>
  <scope>test</scope>
</dependency>
```

JUnit: un ejemplo sencillo (I)

- Veamos un pequeño ejemplo de cómo crear y ejecutar un test unitario de JUnit desde la consola.
- Para ello, crearemos la siguiente estructura de carpetas y archivos:

```
/ . . .  
|-- junit-ejemplos/ (proyecto)  
    |-- src/  
        |-- main/  
            |-- java/  
                |-- ejemplos/ (package)  
                    |-- Calculadora.java  
        |-- test/  
            |-- java/  
                |-- ejemplos/ (package)  
                    |-- CalculadoraTest.java
```

- *src/main/java* : código de la **aplicación** (*ejemplos/Calculadora.java*)
- *src/test/java* : código de las **pruebas unitarias**

JUnit: un ejemplo sencillo (II)

- Nuestra aplicación constará de la siguiente clase: *ejemplos.Calculadora*

```
package ejemplos;

public class Calculadora {
    public int evalua(String expression) {
        int sum = 0;
        for (String summand: expression.split("\\+")) {
            sum += Integer.valueOf(summand);
        }
        return sum;
    }
}
```

- Su único método, *evalutate()*, recibirá un *String* representando una operación de suma (p.ej.: “3+2+1”), dividirá (*split*) la cadena en operandos utilizando el signo “+” como separador, y los sumará en un bucle
- Compilamos la clase (desde la carpeta *junit-ejemplos*):
`javac -d target/classes src/main/java/ejemplos/Calculadora.java`

JUnit: un ejemplo sencillo (III)

- Vamos a crear ahora una **prueba unitaria** para nuestra clase. Para ello, crearemos una nueva clase (*Test Case*) que se encargará de probar los métodos de *Calculadora*. Por convención, este *Test Case* tendrá el mismo nombre que la clase testeada más el sufijo *Test*: *CalculadoraTest*

```
package ejemplos;

import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculadoraTest {
    @Test
    public void evaluaExpresion() {
        Calculadora calculadora = new Calculadora();
        int sum = calculadora.evalua("1+2+3");
        assertEquals(6, sum);
    }
}
```

JUnit: un ejemplo sencillo (IV)

❖ Anotaciones JUnit 4 (JUnit 5)

- Precediendo al método *evaluaExpresion()* se ha añadido una **anotación** Java (**@Test**) para indicar que ese método debe ser ejecutado como parte de la prueba unitaria. JUnit define diferentes anotaciones para establecer la secuencia y comportamiento de los diferentes métodos la prueba unitaria. Los básicos son:
 - **@BeforeClass**, se ejecuta antes de cualquier método de la clase de pruebas (**public static void**) (JUnit5 **@BeforeAll**)
 - **@AfterClass**, se ejecuta después de se hayan ejecutado todos los métodos de la clase de pruebas (**public static void**) (JUnit5 **@AfterAll**)
 - **@Before**, se ejecuta antes de @Test (**public void**) (JUnit5 **@BeforeEach**)
 - **@After**, se ejecuta después de @Test (**public void**) (JUnit5 **@AfterEach**)
 - **@Test**, indica el método a ejecutar (**public void**)

JUnit: un ejemplo sencillo (V)

❖ Aserciones JUnit

- JUnit nos proporciona una serie de métodos estáticos que nos permiten chequear el valor devuelto por los métodos de la clase testeada. Estos métodos recibirán dos parámetros: **valor esperado** y el **valor obtenido**. Opcionalmente, podemos agregar un mensaje de error que se mostrará si no se cumplió la aserción.
- Algunos de estos métodos son:
 - *assertFalse*, comprueba que el valor devuelto sea *false*
 - *assertTrue*, comprueba que el valor devuelto sea *true*
 - *assertEquals*, comprueba que los valores sean iguales
 - *assertSame*, comprueba que los objetos sean los mismos
 - *assertThat*, emplea *Matchers* para la comprobación
- <https://github.com/junit-team/junit4/wiki/Assertions>
- <https://github.com/junit-team/junit4/wiki/matchers-and-assertthat>

JUnit: un ejemplo sencillo (VI)

- Compilemos nuestra clase de test (desde la carpeta *junit-ejemplos*):

```
[linux]
javac -d target/test-classes
-cp .:target/classes:[junit-path]/junit4.jar:[junit-path]/hamcrest-core.jar
src/test/java/ejemplos/CalculadoraTest.java
```

```
[win]
javac -d target/test-classes
-cp .;target/classes;[junit-path]/junit4.jar;[junit-path]/hamcrest-core.jar
src/test/java/ejemplos/CalculadoraTest.java
```

- Fíjate como usamos la opción `-cp` para establecer el *classpath*. Además del directorio actual, añadiremos la ruta a la clase testeada y las rutas a las librerías descargadas de JUnit para que el compilador las encuentre
- Alternativamente, podríamos haber modificado la variable de entorno `CLASSPATH`

JUnit: un ejemplo sencillo (VII)

- La estructura resultante de archivos y carpetas será:

```
/ . . .
|-- junit-ejemplos/ (proyecto)
    |-- src/ (fuentes)
        |-- main/
            |-- java/
                |-- ejemplos/ (package)
                    |-- Calculadora.java
        |-- test/
            |-- java/
                |-- ejemplos/ (package)
                    |-- CalculadoraTest.java
    |-- target/ (código objeto)
        |-- classes/
            |-- ejemplos/ (package)
                |-- Calculadora.class
        |-- test-classes/
            |-- ejemplos/ (package)
                |-- CalculadoraTest.class
```

JUnit: un ejemplo sencillo (VIII)

- Vamos a lanzar el test (por ejemplo, desde la carpeta *junit-ejemplos*). De nuevo, deberemos establecer el valor correcto de *classpath*:

```
[linux]
java -cp
.:target/classes:target/test-classes:[junit-path]/junit4.jar:[junit-path]/hamcrest-core.jar
org.junit.runner.JUnitCore ejemplos.CalculadoraTest

[win]
java -cp
.;target/classes:target/test-classes;[junit-path]/junit4.jar;[junit-path]/hamcrest-core.jar
org.junit.runner.JUnitCore ejemplos.CalculadoraTest
```

- El resultado será:

```
JUnit version 4.12
.
Time: 0,003

OK (1 test)
```

que nos indica que se ha realizado **un único test** y que ha sido **correcto (OK)**

JUnit: un ejemplo sencillo (y IX)

- Vamos a modificar *Calculadora.java* para provocar un fallo en el test.
 - reemplazamos la sentencia: `sum += Integer.valueOf(summand);`
 - por esta otra: `sum -= Integer.valueOf(summand);`
 - y compilamos de nuevo la clase *Calculadora.java*
- Tras ejecutar el test de nuevo, el resultado será:

```
JUnit version 4.12
.E
Time: 0,005
There was 1 failure:
1) evaluaExpresion(ejemplos.CalculadoraTest)
java.lang.AssertionError: expected:<6> but was:<-6>
. . .
FAILURES!!!
Tests run: 1, Failures: 1
```

- La salida indicará **qué test** falló: `evaluaExpresion(ejemplos.CalculadoraTest)`
y **qué fue mal**: `java.lang.AssertionError: expected:<6> but was:<-6>`

JUnit: integración con netbeans (I)

- Una de las ventajas de JUnit es su integración con los IDE's más utilizados (*netbeans*, *eclipse*, *intelliJ*, *vs code*,...) y con las herramientas para la creación de proyectos como *maven* o *gradle*
- Vamos a ver cómo utilizar JUnit desde netbeans. Desde la versión 7, las librerías de JUnit ya se incluyen en la instalación del IDE. Podemos comprobar que está instalado desde el menú: *tools > plugins > Installed*
- Para nuestro ejemplo, crearemos un nuevo proyecto y una nueva clase *Calculadora* con el código mostrado a continuación. Esta clase, dispondrá de dos métodos para sumar (*add()*) o restar (*sub()*) valores en punto flotante a un acumulador, que mantendrá el resultado de la última operación realizada. Podremos obtener el valor actual del acumulador mediante el método *acc()* y borrarlo mediante el método *clear()*

JUnit: un ejemplo sencillo (II)

- *ejemplos.Calculadora*

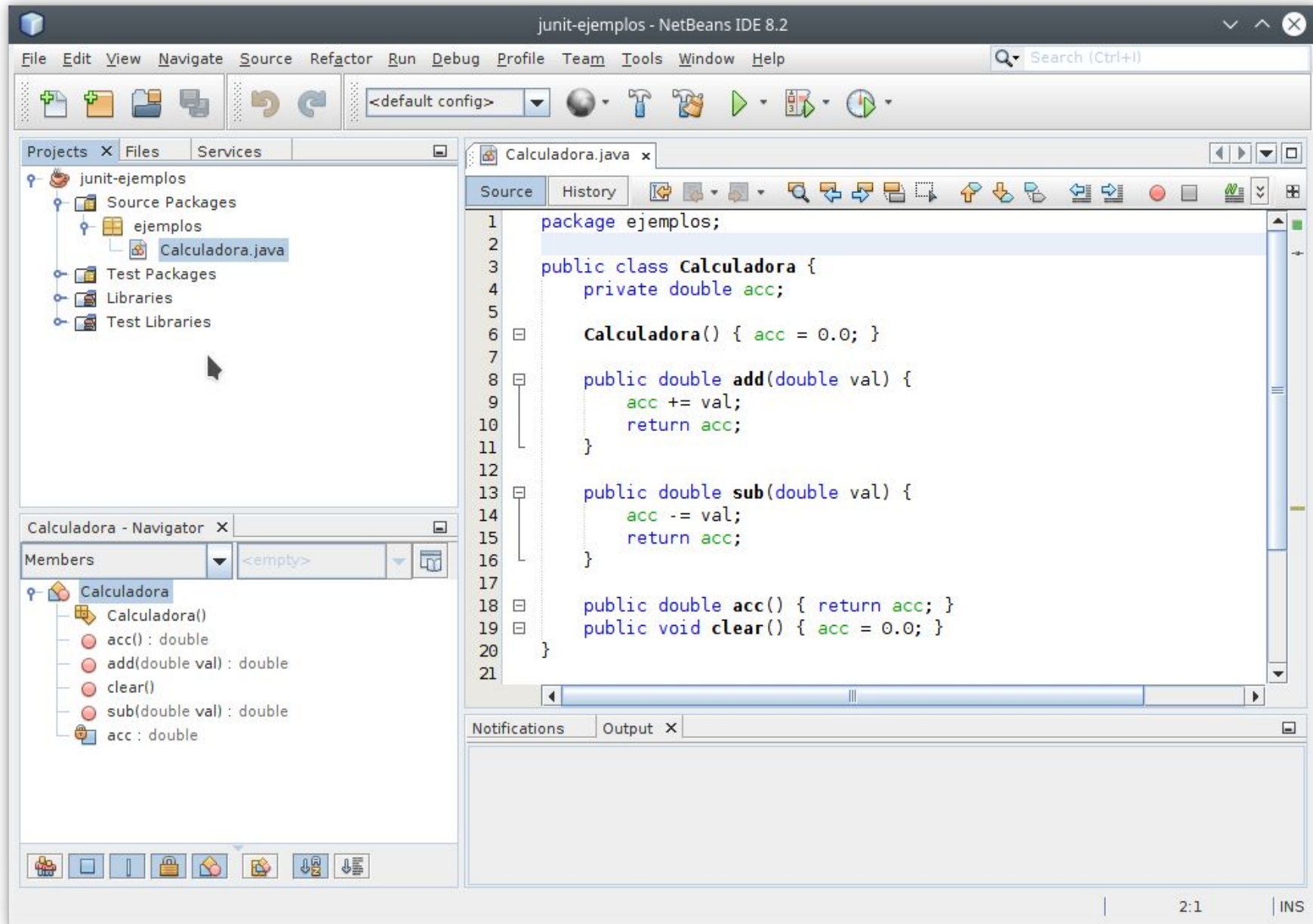
```
package ejemplos;

public class Calculadora {
    private double acc;

    public Calculadora() { acc = 0; }

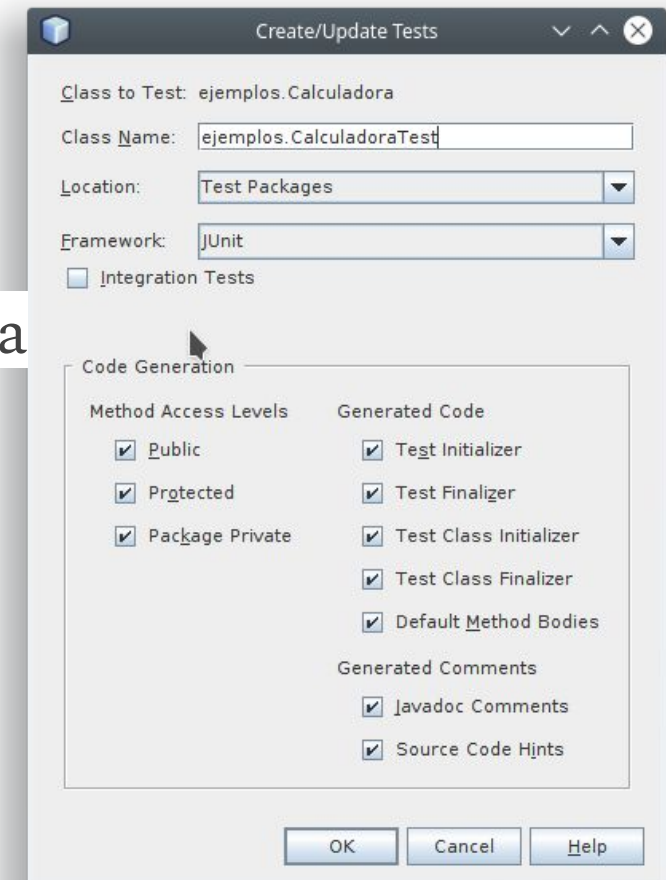
    public double add(double val) {
        acc += val;
        return acc;
    }
    public double sub(double val) {
        acc -= val;
        return acc;
    }
    public double acc() { return acc; }
    public void clear() { acc = 0; }
}
```

JUnit: un ejemplo sencillo (III)

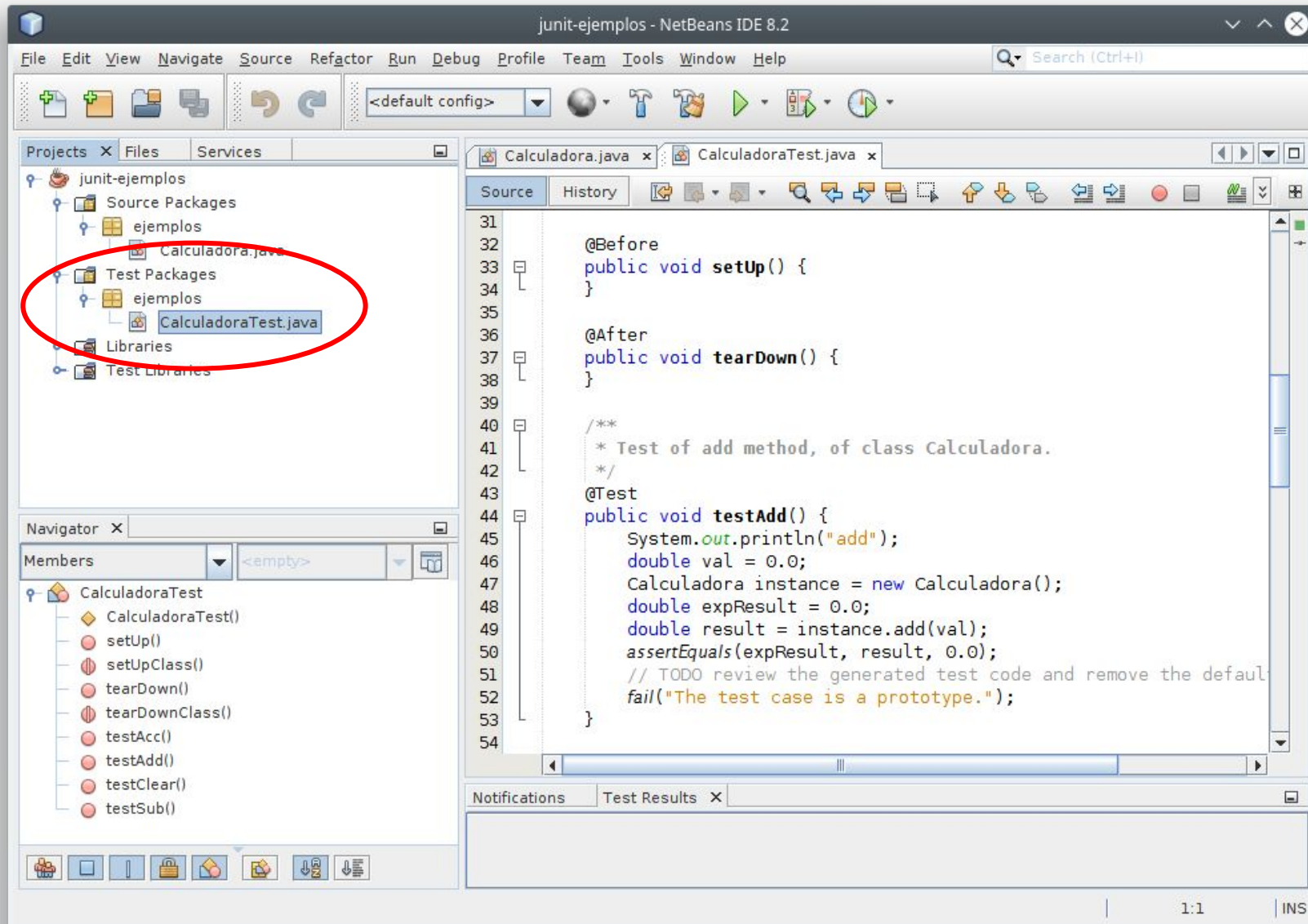


JUnit: integración con netbeans (IV)

- Para crear un *Test Case* para la clase *Calculadora*, sólo tenemos que ir al explorador del proyecto, clicar con el botón derecho sobre la clase, y seleccionar: *Tools > Create/Update Tests*
- Se nos abrirá un diálogo para parametrizar la creación de nuestra clase para la prueba unitaria que llamaremos *CalculadoraTest*
- Por defecto, nos generará métodos de prueba para cada uno de los métodos de nuestra clase. Estos métodos contendrán código que debemos editar para preparar nuestras pruebas
- A nuestro proyecto se añadirá un nuevo paquete que contendrá la nueva clase generada para realizar el test unitario



JUnit: un ejemplo sencillo (V)



JUnit: integración con netbeans (VI)

- Vamos a modificar el código autogenerated de los métodos *testAdd()* y *testSub()* que prueban los métodos *add()* y *sub()* de la clase *Calculadora*
- Además, limpiaremos todo el código superfluo de la clase para que sea un poco más legible

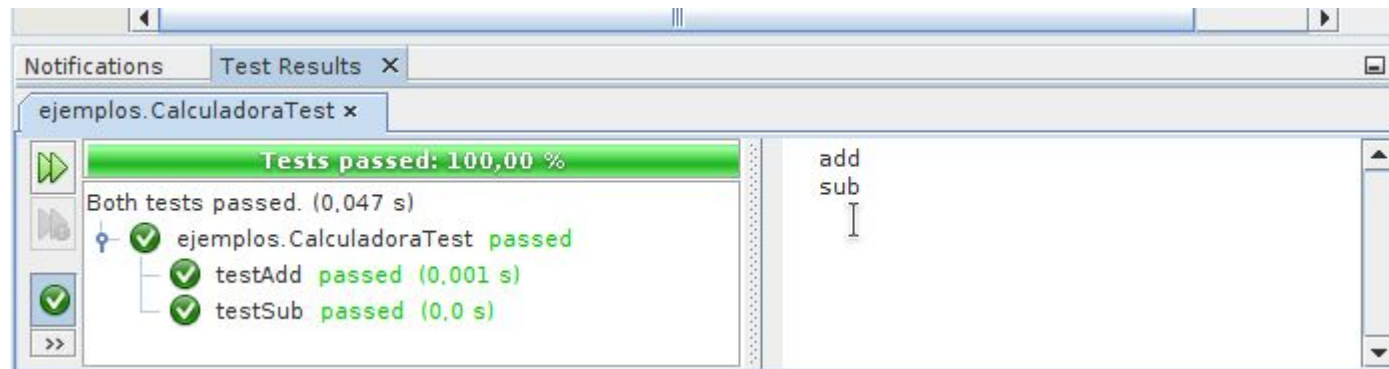
```
package ejemplos;

import static org.junit.Assert.assertEquals;
import org.junit.Test;

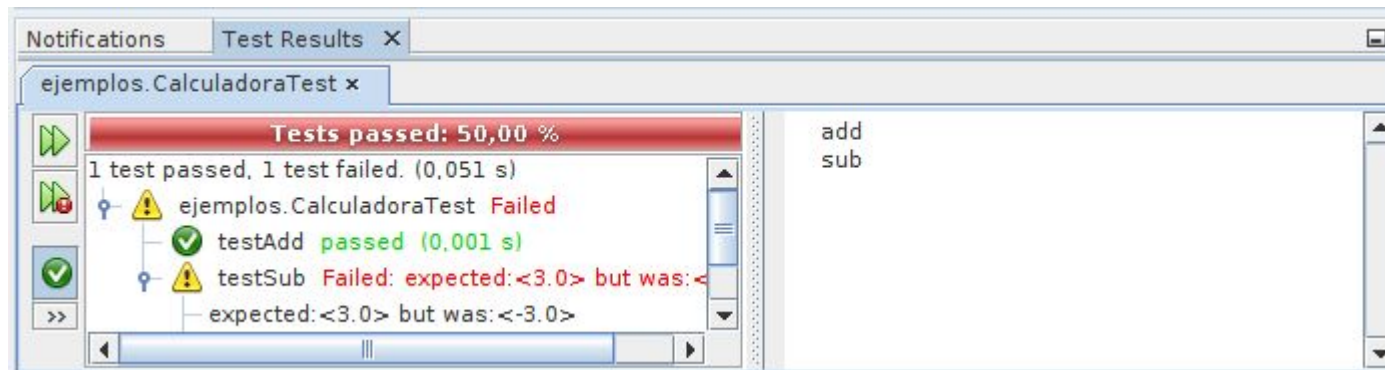
public class CalculadoraTest {
    @Test
    public void testAdd() {
        System.out.println("add");
        Calculadora calc = new Calculadora();
        assertEquals(5.0, calc.add(5.0), 0.0);
    }
    @Test
    public void testSub() {
        System.out.println("sub");
        Calculadora calc = new Calculadora();
        assertEquals(-3.0, calc.sub(3.0), 0.0);
    }
}
```


JUnit: integración con netbeans (VII)

- Para ejecutar el test podemos ir al menú: *Run > Test File (Ctrl + F6)*
- En el panel inferior se nos mostrará el resultado de los tests y la consola con los mensajes de salida generados por los métodos de prueba
- Por ejemplo, todos los test OK:



- Otro ejemplo: un test OK (*testAdd()*) pero el segundo no (*testSub()*):



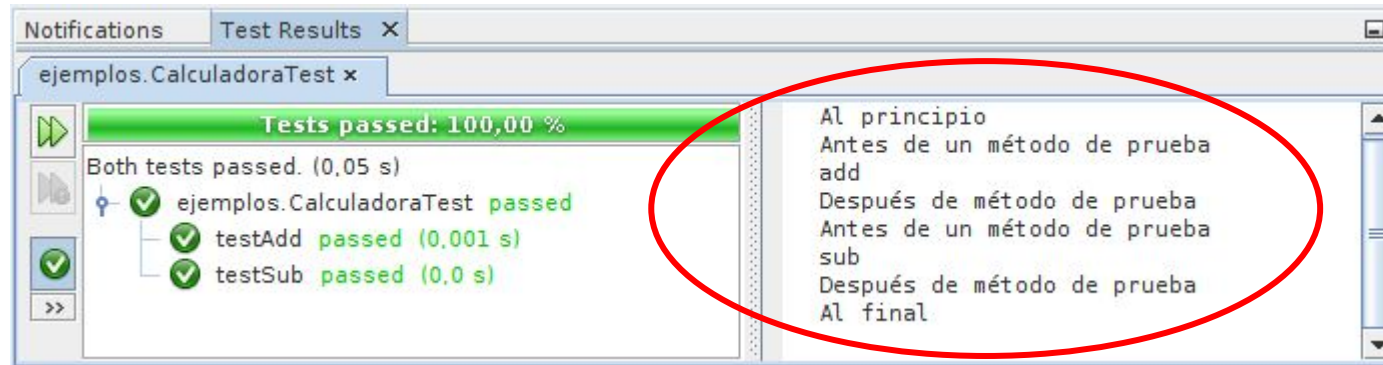
JUnit: integración con netbeans (VIII)

- Veamos en netbeans como funcionan las anotaciones de secuencia de JUnit: *@BeforeClass*, *@AfterClass*, *@Before*, *@After*
- Para ello, vamos a añadir los siguientes métodos que simplemente mostrarán un mensaje en la consola pero nos permitirán ver dicha secuencia de ejecución:

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
. . .
@BeforeClass
public static void alPrincipio() { System.out.println("Al principio"); }
@AfterClass
public static void alFinal() { System.out.println("Al final"); }
@Before
public void antesDeMetodo() { System.out.println("Antes de un método de prueba"); }
@After
public void despuesDeMetodo() { System.out.println("Después de método de prueba"); }
. . .
```


JUnit: integración con netbeans (IX)

- Tras ejecutar el test, se nos mostrará en la consola:



- Vemos como el método *alPrincipio()*, anotado con *@BeforeClass*, se ejecuta una única vez al inicio del test. Los métodos *antesDeMetodo()* y *despuesDeMetodo()*, anotados con *@Before* y *@After*, se ejecutan antes y después de cada método de prueba (*add()* y *sub()*). El método *alFinal()*, anotado con *@AfterClass*, se ejecuta una única vez al final del test.
- El empleo de estos métodos nos facilitará la realización de tareas comunes de inicializació y finalización de nuestros métodos de prueba.

JUnit: integración con netbeans (X)

- En el siguiente ejemplo se crea un único objeto *Calculadora* que será usado por todos los métodos de prueba:

```
. . . (imports)

public class CalculadoraTest {
    private static Calculadora calc;

    @BeforeClass
    public static void alPrincipio() { calc = new Calculadora(); }
    @Before
    public void antesDeMetodo() { calc.clear(); }

    @Test
    public void testAdd() { assertEquals(5.0, calc.add(5.0), 0.0); }
    @Test
    public void testSub() { assertEquals(-3.0, calc.sub(3.0), 0.0); }
    @Test
    public void testCombinado() {
        calc.add(3.5);
        calc.sub(1.2);
        calc.add(0.8);
        assertEquals(3.1, calc.acc(), 1.0E-12);
    }
}
```

JUnit: integración con netbeans (y XI)

- Por último, podemos establecer un tiempo máximo para la ejecución de nuestro método de prueba. Para ello, añadiremos a la anotación `@Test` el parámetro `timeout` con el valor máximo de tiempo de ejecución (en ms) para el método (por ejemplo: `@Test(timeout = 100)`). Esto nos permitiría chequear la eficiencia de nuestros algoritmos o encontrar situaciones de bucles infinitos
- Finalizamos así esta pequeña introducción a JUnit. Nos quedan aún ciertas funcionalidades pendientes de ver. Entre otras:
 - Creación de baterías de pruebas unitarias (*Test Suites*)
 - Creación de pruebas parametrizadas
 - Pruebas de lanzamiento de excepciones
- Volveremos sobre ello después de ver la generación de excepciones en Java y la creación de colecciones y estructuras de datos