

# *TEMA 3: Pruebas del Software*

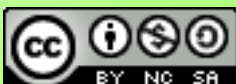
Módulo

Entornos de Desarrollo

para los ciclos

Desarrollo de Aplicaciones Web

Desarrollo de Aplicaciones Multiplataforma



ED FP-GS; Tema3:PruebasDelSoftware

© Gerardo Martín Esquivel, Febrero de 2023

Algunos derechos reservados.

Este trabajo se distribuye bajo la Licencia "Reconocimiento-No comercial-  
Compartir igual 3.0 Unported" de Creative Commons disponible en  
<http://creativecommons.org/licenses/by-nc-sa/3.0/>

<b>3.1 Casos de prueba.....</b>	<b>3</b>
3.1.1 Caja blanca y caja negra.....	4
<b>3.2 Secuencia de pruebas.....</b>	<b>5</b>
<b>3.3 Pruebas habituales.....</b>	<b>6</b>
3.3.1 Prueba del camino básico (caja blanca).....	6
Grafo de flujo.....	6
Complejidad Ciclomática.....	8
Casos de prueba.....	8
Caminos imposibles.....	10
3.3.2 Clases de equivalencia (caja negra).....	12
3.3.3 Valores límite.....	15
<b>3.4 Herramientas de depuración.....</b>	<b>17</b>
3.4.1 Establecer breakpoints.....	17
3.4.2 Lanzar el depurador (debugger).....	18
3.4.3 Herramientas del debugger.....	18
<b>3.5 JUnit.....</b>	<b>21</b>
3.5.1 Creación de una clase de prueba.....	21
3.5.2 Ejecución de casos de prueba.....	23
3.5.3 Tests de prueba para métodos con excepciones.....	27
3.5.4 Anotaciones JUnit.....	30
3.5.5 Pruebas parametrizadas.....	31
@ValueSource.....	31
@CsvSource.....	31
@MethodSource.....	32
3.5.6 Pruebas parametrizadas con excepciones.....	34

### 3.1 Casos de prueba

Buena parte del tiempo de desarrollo de software se dedica a probar el código que se ha generado previamente. Cada prueba realizada sobre el código puede detectar errores (a menudo los detecta) que tienen que ser reparados.

Un **caso de prueba** se compone de entradas y resultados (o comportamiento) esperados. Cada caso de prueba incluirá también unas condiciones de ejecución. Una vez ejecutado el caso de prueba se analiza el comportamiento y se comparan los resultados con lo que se esperaba para determinar si el sistema ha pasado o no la prueba.

**Ejemplo 1: Caso de prueba para un programa que calcula la división de dos números de entrada llamados dividendo y divisor.**

Datos de entrada		Resultado o comportamiento esperado
dividendo	divisor	dato de salida
8	2	4

Si decidimos usar este caso de prueba entonces ejecutaremos el programa aportando los datos de entrada indicados: **8 como dividendo** y **2 como divisor**. Tras ejecutarlo observamos el resultado obtenido y lo comparamos con el resultado esperado: **4**. Si son iguales significa que el programa ha superado la prueba. Si no son iguales habrá que detectar y solucionar el error.

**Ejemplo 2: Caso de prueba para un programa que calcula la división de dos números de entrada llamados dividendo y divisor.**

Datos de entrada		Resultado o comportamiento esperado
dividendo	divisor	dato de salida
8	0	mensaje de error informativo

En este caso el resultado esperado tras ejecutar el programa no es un resultado numérico, sino un comportamiento: un mensaje de error informativo. Si la ejecución termina con un comportamiento distinto significa que hay un fallo que reparar.

Cuando diseñamos **una prueba tendremos que incluir varios casos de prueba, de forma que se cubran todos los grupos de posibilidades**. No podremos probar todas posibles entradas para el programa de la división. Pero si podemos crear un caso de prueba de cada situación distinta: con números enteros, con números decimales, con números negativos, con ceros, etc.

### 3.1.1 Caja blanca y caja negra

Se distinguen dos tipos de pruebas:

- **De caja blanca:** (o estructurales) Se analiza la estructura interna del programa inspeccionando el código, por eso, en las pruebas de caja blanca se incluyen casos de prueba para que:
  - ◆ **Todas las sentencias se ejecutan al menos una vez.** Si hay un bucle habrá que incluir casos de prueba en los que la ejecución entra en el bucle y casos de prueba cuya ejecución se salta el bucle.
  - ◆ **Todas las condiciones se ejecutan en su parte verdadera y en su parte falsa.** Habrá al menos dos casos de prueba: uno que pasa por la parte en la que se cumple la condición y otro que pasa por la parte en la que no se cumple esa condición.
  - ◆ **Los bucles se ejecutan en sus límites.** Todos los programadores, incluso los más experimentados, pueden tropezar en los límites de los bucles. Si un bucle se ejecuta exclusivamente para valores que van desde **n** hasta **m**, tendremos que incluir casos de prueba que llegan al bucle con los valores **n-1**, **n**, **m** y **m+1**. Observa que se trata del primer y último valor que entran al bucle y de los valores adyacentes que no entran.
  - ◆ **Todas las estructuras de datos se usan para asegurar su validez.** Por ejemplo, si nuestro programa incluye algún **array**, la prueba no estará completa si no incluye un caso de prueba que use ese **array**.
- **De caja negra:** (o de comportamiento) Sólo se comprueba la funcionalidad, es decir, si la salida es adecuada en función de los datos de entrada, sin fijarse en el funcionamiento interno. En las pruebas de caja negra también serán necesarios múltiples casos de prueba que incluyan todos los grupos de entradas que tienen comportamientos distintos.

### 3.2 Secuencia de pruebas

La prueba completa del software requiere los siguientes pasos:

- **Prueba de unidad:** Se centra en cada unidad de software, en cada módulo del código fuente. Aquí es donde se utilizan las pruebas de caja blanca y de caja negra. Probaremos cada método, cada clase, etc.
- **Prueba de integración:** A partir de los módulos probados individualmente, se prueba el funcionamiento conjunto. Hay dos enfoques:
  - ◆ **Big bang:** Consiste en ser estrictos en el orden de probar primero todos los módulos por separado antes de pasar al conjunto. Con este enfoque los errores se acumulan y suele ser más complicado.
  - ◆ **Integración incremental:** Se van incorporando módulos poco a poco y probando su integración. Los errores son más fáciles de localizar así.
- **Prueba de validación:** En el entorno real de trabajo y con intervención del usuario final. Se basa en determinar si se cumplen los requisitos y se usan para ello pruebas de caja negra de dos tipos:
  - ◆ **Prueba alfa:** Con el usuario final en el lugar de desarrollo. El desarrollador observa y registra los errores y problemas.
  - ◆ **Prueba beta:** Con el usuario final en su propio lugar de trabajo. El desarrollador no está presente, pero será informado por el usuario.
- **Prueba del sistema:** Para verificar funcionalidad y rendimiento. Se prueba su integración con otros elementos del sistema. Incluye las siguientes:
  - ◆ **Prueba de recuperación:** Se fuerza el fallo del software y se comprueba que la recuperación es satisfactoria.
  - ◆ **Prueba de seguridad:** Se intenta verificar que el sistema está protegido contra accesos ilegales.
  - ◆ **Prueba de resistencia o estrés:** Enfrenta al sistema con situaciones que demandan gran cantidad de recursos: accesos simultáneos, uso de memoria, etc.

Todas las pruebas que se realizan deben de estar documentadas incluyendo calendarios, objetivos, resultados, personal responsable y los detalles de cada prueba. Naturalmente también se deben acompañar de conclusiones y, en su caso, instrucciones para solucionar los problemas detectados.

### 3.3 Pruebas habituales

#### 3.3.1 Prueba del camino básico (caja blanca)

La prueba del camino básico es una prueba de caja blanca que permite obtener una **medida de la complejidad** del diseño de un procedimiento (por ejemplo, un método **Java**). A partir de esa medida se establece un **conjunto básico de caminos** de ejecución y, finalmente, se diseñan casos de prueba para cada uno de esos caminos. Se trata de conseguir que, durante la prueba, todas las sentencias del programa se ejecutan al menos una vez, por eso se trata de una prueba de caja blanca.

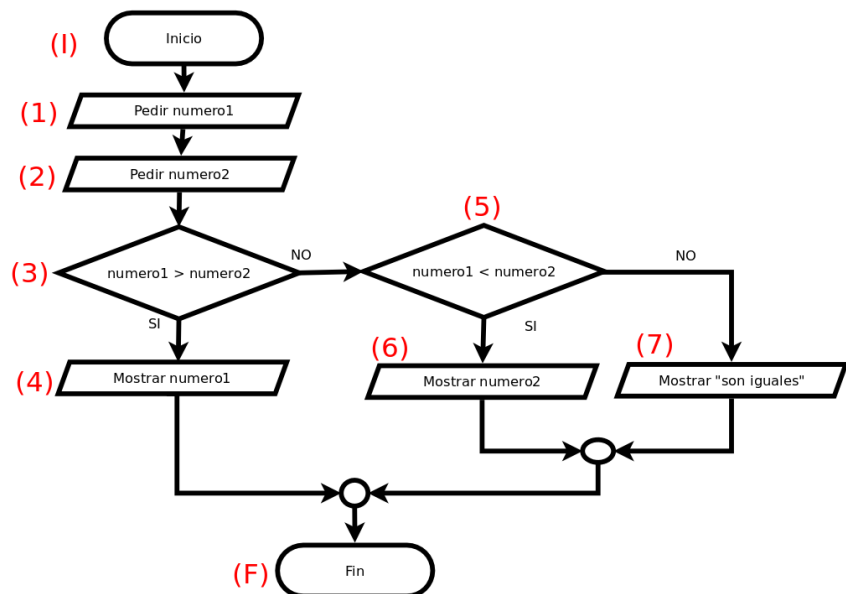
#### GRAFO DE FLUJO

En un grafo de flujo los nodos son círculos que representan un grupo de sentencias que siempre se ejecutan juntas y las flechas indican la dirección que sigue el flujo entre los nodos. Podemos obtener el grafo de flujo a partir de un diagrama de flujo o a partir del código.

#### Ejemplo 1: Grafo de flujo a partir de un diagrama de flujo (Mayor)

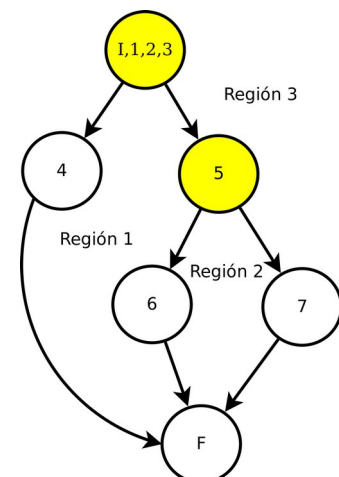
Aquí tenemos el diagrama de flujo para un algoritmo que solicita dos números al usuario y dice cual de ellos es mayor, o si son iguales.

Para obtener el grafo de flujo se numera cada elemento del diagrama (lectura, escritura, condición, etc), incluido el inicio (**I**) y el final (**F**). Posteriormente, al dibujar el grafo, agrupamos en un solo nodo todos los elementos del diagrama que siempre se ejecutan juntos.



La siguiente imagen muestra el grafo de flujo. Tenemos:

- **Nodos:** cada uno de los círculos. En el ejemplo hay 6 nodos.
- **Aristas:** cada una de la flechas. En el ejemplo hay 7 aristas.
- **Nodos predicado:** son los nodos de los que salen dos o más aristas (son los nodos que contienen una condición). En el ejemplo hay 2 nodos predicado (los marcados en amarillo).
- **Regiones:** cada una de las áreas encerradas por nodos y aristas. El área exterior del grafo es una región más. En el ejemplo hay 3 regiones.



### Ejemplo 2: Grafo de flujo a partir de código o pseudocódigo (Sumador)

En realidad, no hay diferencias en la construcción del grafo a partir de un diagrama de flujo, pseudocódigo o código. En el código (o pseudocódigo) se numera cada elemento y se procede de la misma forma.

En el siguiente ejemplo tenemos un método **Java** en el que se pide al usuario que introduzca dos enteros positivos y devuelve la suma. Antes de sumar comprueba que los valores introducidos son, efectivamente, positivos. Cada línea de código ha sido numerada (incluyendo inicio y fin), pero observa que la línea que tiene **la condición compleja tiene dos números**. Esto es porque, aunque nosotros lo escribimos como una única condición, en realidad se comprueban una tras otra.

**Nota:** Hay que tener especial cuidado con las condiciones complejas porque tendrán que representarse mediante varios nodos, tantos como condiciones simples equivalentes.

```
(I) public void suma () {
(1)   Scanner teclado = new Scanner(System.in);
(2)   int a, b, resultado;

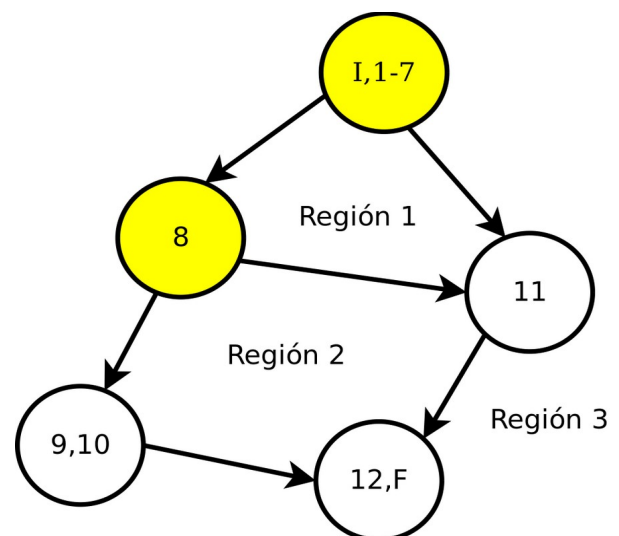
(3)   System.out.println("Introduce un número positivo");
(4)   a = teclado.nextInt();
(5)   System.out.println("Introduce otro número positivo");
(6)   b = teclado.nextInt();
(7)y(8) if ((a>0) && (b>0)) {
(9)       resultado = a + b;
(10)      System.out.println("El resultado es: "+resultado);
(11)  } else {
(11)      System.out.println("No son positivos");
(12)  }
(12)  teclado.close();
(F) }
```

El número (7) corresponde a la primera condición:  $a > 0$ . Si cuando el compilador evalúa esta condición obtiene un valor **false**, no evaluará la segunda porque independientemente de su valor, el camino a seguir ya está determinado.

El número (8) corresponde a la segunda condición:  $b > 0$ . Esta condición sólo será evaluada en caso de que la primera tenga un valor de **verdadero**.

El grafo de flujo, que aparece en la siguiente imagen, tiene:

- 5 nodos
- 6 aristas
- 2 nodos predicado (en amarillo)
- 3 regiones



### COMPLEJIDAD CICLOMÁTICA

Mide la complejidad de un programa y coincide con el número de caminos distintos que puede seguir el flujo. Contar el número de caminos puede ser complicado, sobre todo cuando aumenta el tamaño y complejidad del grafo. Por eso nos ayuda saber que ese número coincide con:

- El número de **caminos** distintos.
- El número de **regiones** del grafo.
- El número de aristas, restando el número de nodos y sumando 2: (**aristas-nodos+2**)
- El número de nodos predicado más 1: (**predicados + 1**)

Una complejidad ciclomática por encima del valor 20 en un procedimiento entraña un riesgo alto porque indica que se trata de un programa complejo. Si pasa de 50 hablamos de "muy alto riesgo" o incluso, de programas no testeables. Para evitar estos valores podemos ayudarnos de la programación modular, es decir, dividir el programa en varios módulos o subprogramas.

#### Ejemplo 1: Complejidad ciclomática del programa Mayor.

El grafo tiene 6 nodos, 7 aristas, 2 nodos predicado y 3 regiones. Si aplicamos las fórmulas veremos que la complejidad ciclomática del ejemplo 1 es **3**, es decir, hay 3 caminos distintos y son:

- **camino 1:** I, 1, 2, 3, 4, F
- **camino 2:** I, 1, 2, 3, 5, 6, F
- **camino 3:** I, 1, 2, 3, 5, 7, F

#### Ejemplo 2: Complejidad ciclomática del Sumador

Este grafo tiene 5 nodos, 6 aristas, 2 nodos predicado y 3 regiones. Aplicando cualquiera de las fórmulas expuestas anteriormente concluimos que tiene **3 caminos** distintos y por tanto, tendremos que elaborar 3 casos de prueba. Los caminos son:

- **camino 1:** I, 1-7, 8, 9-10, 12, F
- **camino 2:** I, 1-7, 8, 11, 12, F
- **camino 3:** I, 1-7, 11, 12, F

### CASOS DE PRUEBA

Conocidos los caminos hay que construir casos de prueba para cada camino, eligiendo datos de entrada que provoquen la bifurcación de los nodos predicado hacia donde nosotros queremos.

Construimos una tabla con una fila para cada camino, describimos el caso de prueba, normalmente con los valores de entrada y el resultado que sería correcto.

#### Ejemplo 1: Casos de prueba necesarios para chequear todos los caminos (Mayor)

Caso de prueba	Valores entrada	Resultado esperado
El usuario introduce el primer número mayor que el segundo. Camino 1	numero1=10 numero2=5	Mostrará el número 10
El usuario introduce el primer número menor que el segundo. Camino 2	numero1=1 numero2=5	Mostrará el número 5
El usuario introduce el primer número igual que el segundo. Camino 3	numero1=8 numero2=8	Mostrará el mensaje "son iguales"



**Ejemplo2: Casos de prueba necesarios para chequear todos los caminos del Sumador****Caso de prueba**

El usuario introduce dos valores positivos. Camino 1

El usuario introduce un valor positivo y un valor no positivo. Camino 2

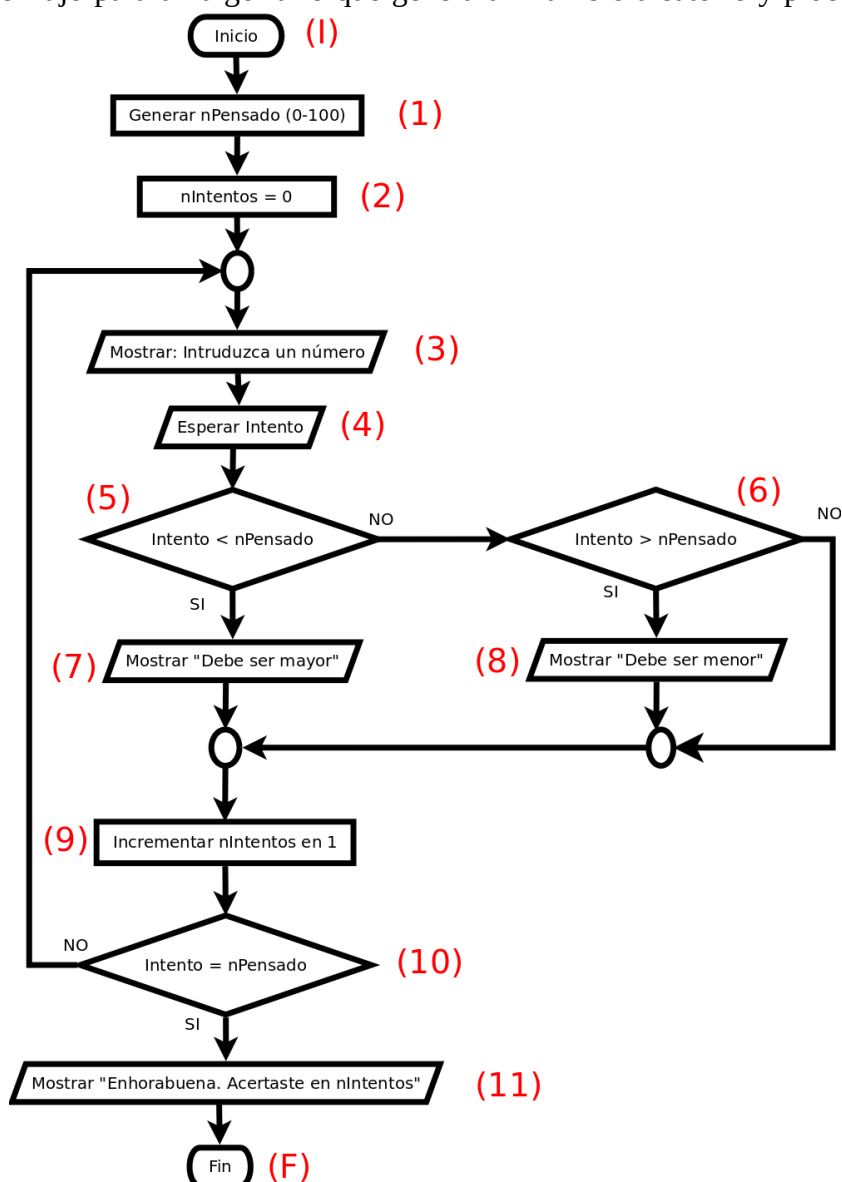
El usuario introduce un primer no positivo. (el segundo valor es irrelevante). Camino 3

Valores entrada	Resultado esperado
a=24 y b=45	Mensaje: "El resultado es: 69" y final.
a=18 y b=-12	Mensaje: "No son positivos" y final.
a=-15 y b=4	Mensaje: "No son positivos" y final.

**Ejemplo 3: Grafo de flujo a partir de un diagrama de flujo (Adivinador)**

Aquí tenemos el diagrama de flujo para un algoritmo que genera un número aleatorio y pide al usuario que lo adivine. El usuario podrá seguir intentándolo hasta acertar y finalizar. Si no acierta, se le informa si el número que tiene que adivinar es mayor o menor que el introducido.

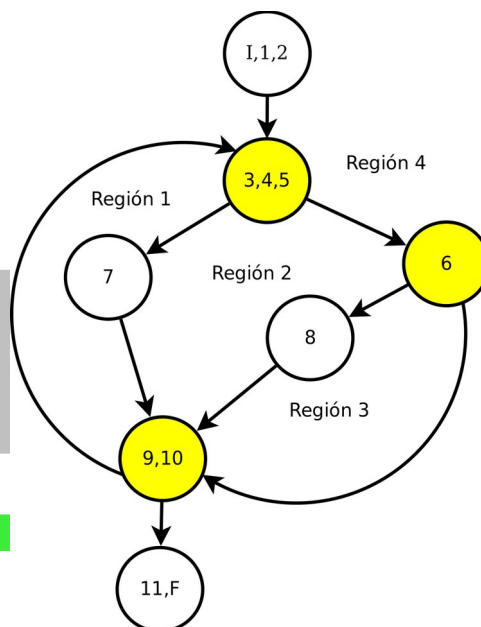
Para obtener el grafo de flujo se numera cada elemento del diagrama (lectura, escritura, condición, etc), incluido el inicio (I) y el final (F). Posteriormente, al dibujar el grafo, agrupamos en un solo nodo todos los elementos del diagrama que siempre se ejecutan juntos.



La siguiente imagen muestra el grafo de flujo. Tenemos:

- 7 nodos.
- 9 aristas.
- 3 nodos predicado (en amarillo).
- 4 regiones.

**Nota:** La realidad es que habría infinitos caminos, puesto que un usuario podría introducir eternamente el mismo valor. Pero debes recordar que buscamos un conjunto básico de caminos en el que todas las sentencias se ejecutan al menos una vez.



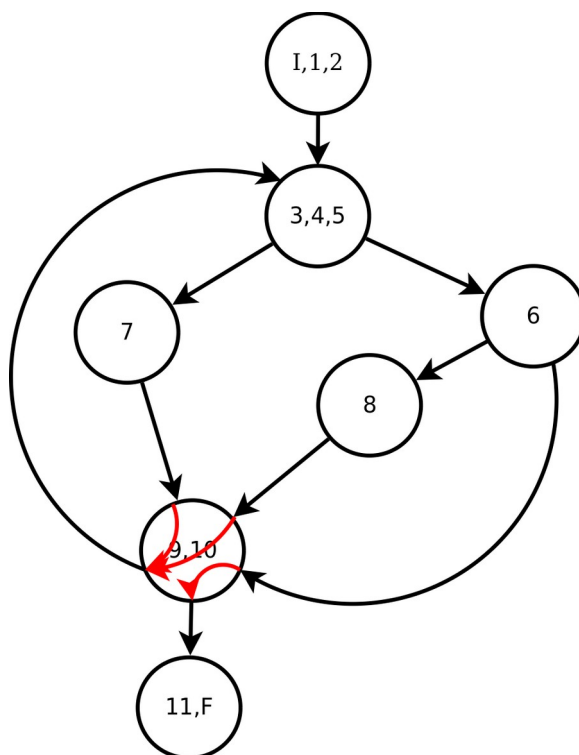
### CAMINOS IMPOSIBLES

Si aplicamos las fórmulas veremos que la complejidad ciclomática es **4**, es decir, hay 4 caminos distintos. Sin embargo, en este caso no es cierto porque algunos de los caminos resultantes no son posibles. Por ejemplo, uno de los caminos que podríamos obtener de este grafo es: **I, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, F**. Si este camino pasa del nodo **6** al **8** es porque la condición **intento > nPensado** se evalúa como **cierta**. Puesto que el valor de las variables **intento** y **nPensado** no se alteran en los nodos **8** y **9**, cuando llegamos al nodo **10**, la condición **intento = nPensado** se tiene que evaluar como **falsa**. Por tanto, desde el nodo **10** debe pasar al nodo **3** y no al nodo **11**.

La consecuencia es que algunos caminos que resultan del grafo anterior son caminos imposibles. Esta situación se produce **cuando hay nodos predicado a los que llega más de una flecha** y tenemos que mejorar estos grafos indicando cuáles de las salidas son posibles para cada entrada:

Observa las flechas rojas que hemos añadido al grafo. El **nodo 9,10** tiene tres entradas y dos salidas, pero no todas son compatibles entre sí: cuando el flujo llega desde el **nodo 6** la única dirección posible es hacia el **nodo 11**. Y cuando el flujo llega desde el **nodo 7** o desde el **nodo 8** la única salida posible es hacia el **nodo 3**.

Una vez descubierta esta situación dibujaremos el verdadero grafo de flujo, en el que repetiremos el nodo **9,10** tantas veces como sea necesario (en este caso, dos veces) para que se ajuste a la situación real.

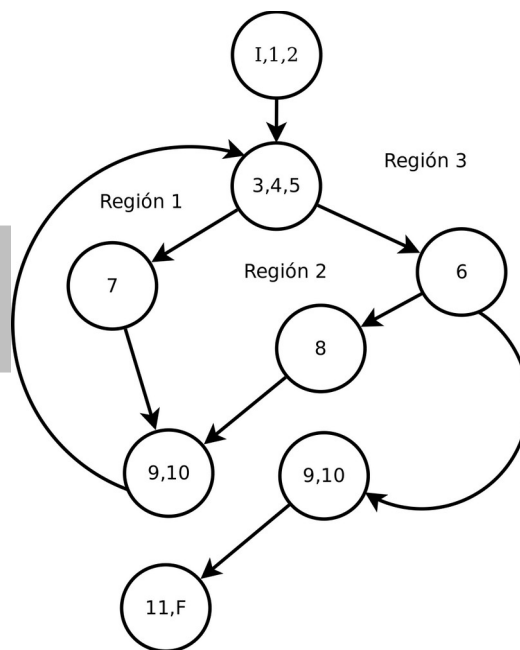


Este será el resultado final, que solo tiene 3 caminos. Y son:

- **camino 1:** I,1-6, 9, 10, 11, F
- **camino 2:** I,1-6, 8-10, 3-6, 9, 10, 11, F
- **camino 3:** I,1-5, 7, 9, 10, 3-6, 9, 10, 11, F

**Nota:** Algunos caminos no pueden ser probados por sí solos, sino como parte de otros caminos. En nuestro ejemplo, los **caminos 2 y 3** también están probando el **camino 1**.

Puesto que ya hemos conseguido un grafo en el que todos los caminos son posibles, ahora toca crear un caso de prueba para cada uno de esos caminos:



#### Caso de prueba

El usuario acierta el número al primer intento. Camino 1

El usuario introduce un número mayor y después el número correcto. Camino 2

El usuario introduce un número menor y después el número correcto. Camino 3

Valores entrada	Resultado esperado
nPensado=34 Intento=34	"Enhorabuena. Acertaste en 1"
nPensado=52 Intento=75 y 52	"Debe ser menor" "Enhorabuena. Acertaste en 2"
nPensado=17 Intento=5 y 17	"Debe ser mayor" "Enhorabuena. Acertaste en 2"

### 3.3.2 Clases de equivalencia (caja negra)

Ya hemos visto que necesitamos al menos un caso de prueba para cada camino, pero con eso no basta. Tenemos que añadir más casos de prueba, utilizando algunas otras técnicas como la de las clases de equivalencia.

Una **clase de equivalencia** es el conjunto de todos los valores para los que el programa debería comportarse del mismo modo. Cuando usamos clases de equivalencia, diseñamos un caso de prueba para un valor al azar de cada clase y damos por probados todos los valores.

Las clases de equivalencia se determinan a partir de cada condición:

- Si una condición especifica un **rango**, tendremos una clase válida (dentro del rango) y dos clases no válidas (una por encima del rango y otra por debajo).

Por ejemplo una condición que se aplica a valores enteros entre 0 y 100 genera una clase válida (todos los valores entre 0 y 100), una clase no válida con todos los valores menores que 0 y una clase no válida con todos los valores mayores que 100.

- Si una condición requiere un **valor específico** tendremos una clase válida (ese valor) y dos clases no válidas (una por encima del valor y otra por debajo).

Por ejemplo, el control de la división se aplica exclusivamente al valor cero. Tendremos una clase no válida (el cero), una clase válida con los valores menores que cero y una clase válida con los valores mayores que cero.

- Si una condición especifica un **conjunto**, tendremos una clase válida (valor perteneciente al conjunto) y una clase no válida (no perteneciente al conjunto).

Por ejemplo, una condición que se aplica a los números pares genera una clase válida (el conjunto de los números pares) y una clase no válida (el conjunto de los números impares).

- Si es una **condición lógica** tendremos una clase válida (valor que cumple la condición) y una clase no válida (valor que no cumple la condición).

**Nota:** Se llama clases de equivalencia **no válidas** a las que corresponden a valores que el programa debe rechazar y clases de equivalencia **válidas** a las que el programa debe aceptar.

#### Ejemplo 1: Clasificador de población

Tenemos un método que a partir de la edad y el género (hombre **-H-** o mujer **-M-**) devuelve el mensaje que corresponde según la tabla que se muestra a continuación.

Edad	Género	Respuesta
De 0 a 17	Cualquiera	menor
De 18 a 64	Hombre	trabajador
	Mujer	trabajadora
De 65 a 125	Hombre	jubilado
	Mujer	jubilada

El código **Java** es este:

```
public static String clasifica(int edad, char genero) throws
EdadErroneaException, GeneroErroneoException {
    String respuesta = "";
    genero = Character.toUpperCase(genero);

    if (edad < 0 || edad > 125) {
        throw new EdadErroneaException(edad);
    }
    if (edad < 18) {
        respuesta = "menor";
    } else {
        if (edad < 65) {
            switch (genero) {
                case 'H':
                    respuesta = "trabajador";
                    break;
                case 'M':
                    respuesta = "trabajadora";
                    break;
                default:
                    throw new GeneroErroneoException(genero);
            }
        } else {
            switch (genero) {
                case 'H':
                    respuesta = "jubilado";
                    break;
                case 'M':
                    respuesta = "jubilada";
                    break;
                default:
                    throw new GeneroErroneoException(genero);
            }
        }
    }
    return respuesta;
}
```

El valor de entrada **edad** genera 5 clases de equivalencia (3 válidas, 2 no válidas):

Valor del dato edad	Clase de equivalencia
edad < 0	NV1 (clase no válida 1)
0 <= edad < 18	V2 (clase válida 2)
18 <= edad < 65	V3
65 <= edad <=125	V4
125 < edad	NV5

El valor de entrada **genero** genera 3 clases de equivalencia (2 válidas, 1 no válida):

Valor del dato genero	Clase de equivalencia
genero = 'h' ó 'H'	V6
genero = 'm' ó 'M'	V7
Cualquier otro valor	NV8

Y la entrada completa del método, que incluye ambos datos, necesitará **15 casos de prueba** que son los que se obtienen de combinar las **5 clases de la edad** con las **3 clases del género** ( $5 \times 3 = 15$ ):

<i>Clases de equivalencia</i>	<i>Valores elegidos</i>	<i>Resultado esperado</i>
NV1 con V6	edad = -7 y genero = 'h'	Genera una excepción por edad
NV1 con V7	edad = -7 y genero = 'M'	Genera una excepción por edad
NV1 con NV8	edad = -7 y genero = 'a'	Genera una excepción por edad
V2 con V6	edad = 15 y genero = 'H'	"menor"
V2 con V7	edad = 15 y genero = 'm'	"menor"
V2 con NV8	edad = 15 y genero = 's'	"menor"
V3 con V6	edad = 50 y genero = 'H'	"trabajador"
V3 con V7	edad = 50 y genero = 'm'	"trabajadora"
V3 con NV8	edad = 50 y genero = '?'	Genera una excepción por genero
V4 con V6	edad = 100 y genero = 'h'	"jubilado"
V4 con V7	edad = 100 y genero = 'M'	"jubilada"
V4 con NV8	edad = 100 y genero = '9'	Genera una excepción por genero
NV5 con V6	edad = 200 y genero = 'h'	Genera una excepción por edad
NV5 con V7	edad = 200 y genero = 'M'	Genera una excepción por edad
NV5 con NV8	edad = 200 y genero = 'o'	Genera una excepción por edad

Nuestras pruebas deben incluir al menos estos 15 casos de prueba porque las clases de equivalencia nos dicen que son 15 situaciones claramente distintas. No obstante, se pueden crear más (varios casos de prueba para cada clase de equivalencia).

**Recuerda:** el uso de **clases de equivalencia** es independiente del uso del **conjunto básico de caminos**, de modo que a la tabla anterior habrá que añadir los casos de prueba que resulten del conjunto de caminos.

### 3.3.3 Valores límite

Este análisis se basa en el hecho de que los errores tienden a producirse con más frecuencia en los extremos de los rangos de entrada. Este análisis complementa la prueba anterior, eligiendo valores que están justo en los límites de las clases de equivalencia.

El análisis de valores límite debería buscar también probar los valores límite en la salida, es decir, buscar valores de entrada que provocan salidas en los extremos de rangos. Esto tiene una mayor dificultad.

- Si una condición especifica un **rango** (por ejemplo, la entrada de edad válida entre 0 y 125) cogemos los valores límite (interno y externo) de ambos extremos del rango.

Por ejemplo, para el **rango [0-125]**, tomaremos los valores **-1, 0, 125 y 126**.

Si aplicamos esta regla a las condiciones de salida (por ejemplo, un programa que devuelve la edad de una persona), se deberán diseñar casos de prueba que provoquen salidas en los límites del rango de los valores esperados, aunque no siempre será posible.

- Si una condición especifica un **número de entradas** (por ejemplo, un programa que requiere 5 números para sumarlos), se deben diseñar casos de prueba con los valores máximo y mínimo, uno justo por debajo del mínimo y uno justo por encima del máximo.

Por ejemplo, si se requieren entre **3 y 7 entradas**, diseñamos casos de prueba con **2, 3, 7 y 8** entradas.

Si aplicamos esta regla a las condiciones de salida, se deberán diseñar casos de prueba que provoquen salidas con un número de valores en los límites del número de valores esperado, aunque no siempre será posible.

- Si las estructuras de datos internas son limitadas, hay que diseñar pruebas que trabajen en sus límites.

Valor requerido	Casos de prueba
Entero entre 1 y 100 inclusive	0, 1, 100, 101
Número real de 0 a 25	-0.1, 0, 25, 25.1
Cadena de 1 a 4 caracteres	Cadenas de 0, 1, 4 y 5 caracteres

Volviendo al ejemplo anterior: ya habíamos determinado las clases de equivalencia y sus combinaciones. Ahora, aplicando la idea de los valores límite, añadimos varios casos de prueba para cada combinación (con fondo verde, los valores de los límites):

Clases de equivalencia	Valores elegidos	Resultado esperado
NV1 con V6	edad = -7 y genero = 'h'	Genera una excepción por edad
NV1 con V6	edad = -1 y genero = 'h'	Genera una excepción por edad
NV1 con V7	edad = -7 y genero = 'M'	Genera una excepción por edad

NV1 con V7	edad = -1 y genero = 'M'	Genera una excepción por edad
NV1 con NV8	edad = -7 y genero = 'a'	Genera una excepción por edad
NV1 con NV8	edad = -1 y genero = 'a'	Genera una excepción por edad
V2 con V6	edad = 15 y genero = 'H'	"menor"
V2 con V6	edad = 0 y genero = 'H'	"menor"
V2 con V6	edad = 17 y genero = 'H'	"menor"
V2 con V7	edad = 15 y genero = 'm'	"menor"
V2 con V7	edad = 0 y genero = 'm'	"menor"
V2 con V7	edad = 17 y genero = 'm'	"menor"
V2 con NV8	edad = 15 y genero = 's'	"menor"
V3 con V6	edad = 50 y genero = 'H'	"trabajador"
V3 con V6	edad = 18 y genero = 'H'	"trabajador"
V3 con V6	edad = 64 y genero = 'H'	"trabajador"
V3 con V7	edad = 50 y genero = 'm'	"trabajadora"
V3 con V7	edad = 18 y genero = 'm'	"trabajadora"
V3 con V7	edad = 64 y genero = 'm'	"trabajadora"
V3 con NV8	edad = 50 y genero = '?'	Genera una excepción por genero
V4 con V6	edad = 100 y genero = 'h'	"jubilado"
V4 con V6	edad = 65 y genero = 'h'	"jubilado"
V4 con V6	edad = 125 y genero = 'h'	"jubilado"
V4 con V7	edad = 100 y genero = 'M'	"jubilada"
V4 con V7	edad = 65 y genero = 'M'	"jubilada"
V4 con V7	edad = 125 y genero = 'M'	"jubilada"
V4 con NV8	edad = 100 y genero = '9'	Genera una excepción por genero
NV5 con V6	edad = 200 y genero = 'h'	Genera una excepción por edad
NV5 con V6	edad = 126 y genero = 'h'	Genera una excepción por edad
NV5 con V7	edad = 200 y genero = 'M'	Genera una excepción por edad
NV5 con V7	edad = 126 y genero = 'M'	Genera una excepción por edad
NV5 con NV8	edad = 200 y genero = 'o'	Genera una excepción por edad
NV5 con NV8	edad = 126 y genero = 'o'	Genera una excepción por edad

La tabla que hemos conseguido trata de escenificar la importancia de ocuparse de todas las situaciones distintas a las que se pueda enfrentar el código. Probablemente sea exagerada la cantidad de casos de prueba para un método tan simple y se podrían eliminar algunos de ellos, pero procurando que siempre quede algún caso de prueba en cada valor límite y en cada combinación de clases de equivalencia.

**Recuerda:** A esta tabla habría que añadir los casos de prueba que surjan del **conjunto básico de caminos**, pero se pueden desechar aquellos caminos que ya son recorridos con los casos de prueba ya contemplados.



### 3.4 Herramientas de depuración

La depuración consiste en ejecutar los casos de prueba, evaluar los resultados obtenidos y compararlos con los resultados esperados. En caso de discrepancia:

- Si se encuentra la causa del error: se corrige y se elimina.
- Si no se encuentra la causa del error: el encargado de la depuración deberá hacer hipótesis sobre las causas, diseñar casos de prueba para confirmar o descartar esas sospechas y volver a repetir las pruebas.



Hay errores (llamados de compilación) que son fáciles de detectar y solucionar sobre todo si usamos un **IDE**. Sin embargo, hay otros errores (errores lógicos también llamados **bugs**), que son más difíciles de detectar porque se completa la compilación sin problemas, pero a la hora de ejecutar devuelve resultados inesperados o erróneos.

Los **IDE** incluyen herramientas para depurar el código (depuradores o **debugger**) que permiten parar la ejecución en el punto que decidamos, analizar los resultados intermedios en ese punto y ejecutar paso a paso.

#### 3.4.1 Establecer breakpoints



Lo primero que debes hacer para depurar código con el **debugger** es indicar los puntos de ruptura (**breakpoints**), es decir, las líneas de código donde quieres que se detenga la ejecución. Sólo es posible establecer puntos de ruptura en las líneas donde hay código de verdad (no se puede hacer en las líneas en blanco o en aquellas que contienen exclusivamente símbolos de sintaxis como llaves de apertura o cierre).

Para establecer los **breakpoints**:



Eclipse	Netbeans
<b>dobles clic en el margen izquierdo de la línea</b> (a la izquierda del número de línea)	<b>clic en el número de línea</b>
Quedará marcado con un círculo azul 	Quedará marcado con un cuadrado rosa 

Los **breakpoints** se quitan del mismo modo que se ponen, es decir, haciendo clic o doble clic en el mismo sitio.

Cuando marcamos un **breakpoint** en la cabecera de la clase (**breakpoint** de clase) se señala de forma especial:

Eclipse	Netbeans
Círculo verde con una C en su interior 	Triángulo rosa invertido 


Cuando marcamos un **breakpoint** en la cabecera de método (**breakpoint** de método) se señala así:

Eclipse	Netbeans
Círculo azul con una flecha sobre él 	Triángulo rosa invertido 



### 3.4.2 Lanzar el depurador (debugger)

Una vez que se han establecido los puntos de ruptura podemos lanzar el depurador.

Para lanzar el depurador:

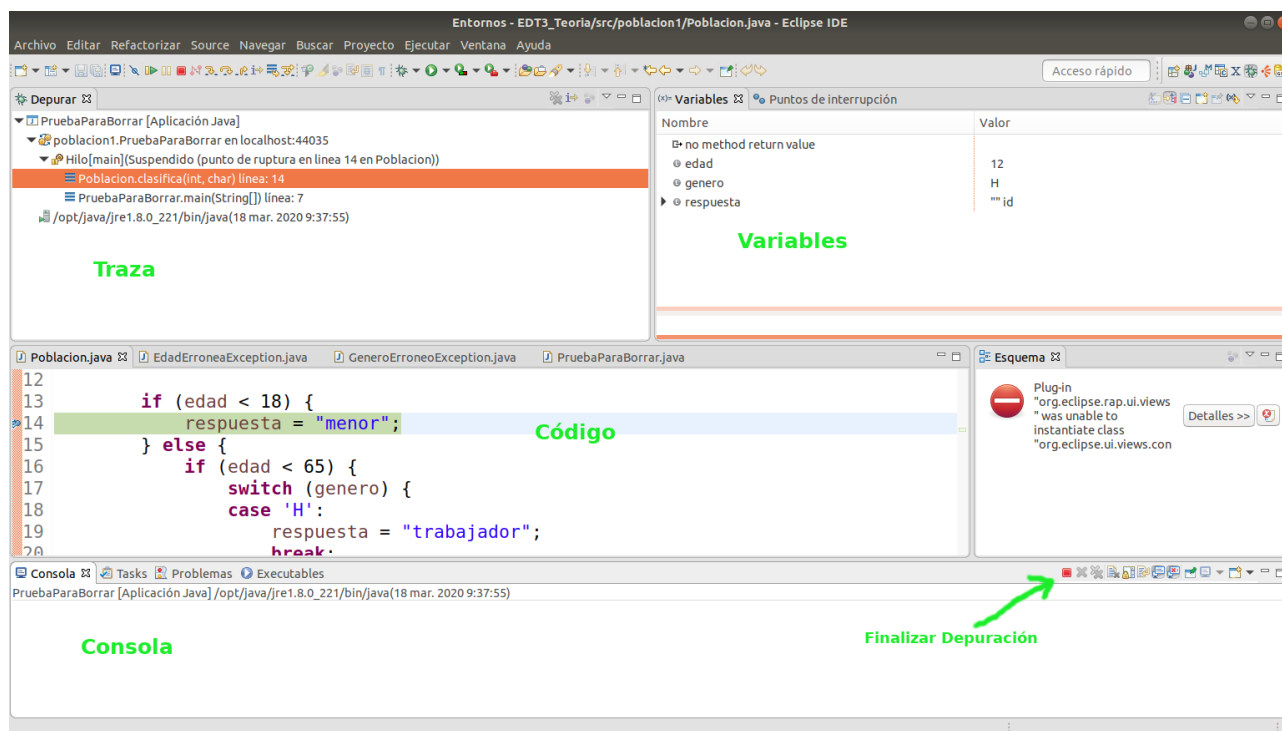
<i>Eclipse</i>	<i>Netbeans</i>
<b>Run/Debug o Ejecutar/Depurar</b> (o usando el menú botón del "escarabajo"  que está junto a la flecha verde de ejecución normal)	<b>Depurar/Debug Project</b> o <b>Debug/Debug Project</b> (o CTRL-F5)
Es conveniente abrir la vista de depuración con: <b>Ventana/Abrir perspectiva/Debug</b> (probablemente te lo sugiera el propio <i>Eclipse</i> )	

Para finalizar la ejecución:

<i>Eclipse</i>	<i>Netbeans</i>
<b>Botón del cuadrado rojo</b>	
	

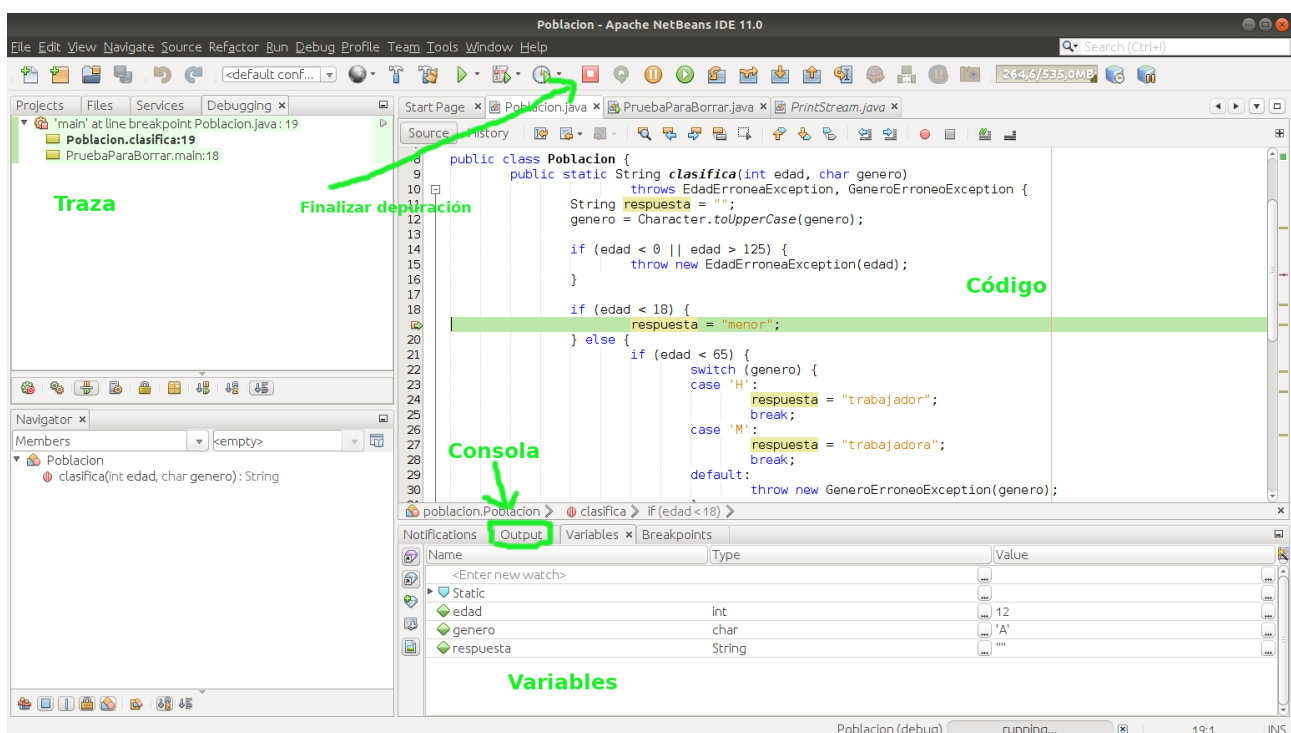
### 3.4.3 Herramientas del debugger

Durante la ejecución en modo depuración podremos ver en distintas ventanas. En la siguiente imagen aparece la distribución de estas ventanas en *Eclipse*:



- La **ventana de código**, y señalada, la siguiente línea a ejecutar.
- La **consola** donde se muestra la salida del programa e introducimos los datos de teclado que solicite el programa.
- Las **variables** con el valor que tienen en este punto de la ejecución. Los valores de las variables irán cambiando conforme avanza la ejecución del código.
- La **traza del punto de ejecución**, es decir, en que línea de cada uno de los métodos se encuentra. Si un método llama a otro que a su vez llama a un tercer método, la traza indica el número de la línea de ejecución en que nos encontramos en cada uno de ellos.

En la siguiente imagen se muestra la distribución de la pantalla en el **debugger de Netbeans**:



Durante la ejecución en modo depuración podemos realizar varias acciones:

- Consultar el valor de una variable.
- Ejecutar la línea en la que nos encontramos.
- Ejecutar desde la línea actual hasta el próximo punto de ruptura.
- Ejecutar desde la línea actual hasta la posición del cursor.
- Ejecutar la línea en que nos encontramos sin entrar dentro del método (suponiendo que en esta línea hay una llamada a un método).
- Ejecutar hasta salir del método actual.
- Modificar "manualmente" el valor de alguna variable.
- Configurar el comportamiento de un **breakpoint**.

	<i>Eclipse</i>	<i>Netbeans</i>
Ver el valor actual de una variable.	<i>pasando el ratón sobre ella</i>	
Paso a paso ( <i>Step Info</i> ).	<i>F5</i>	<i>F7</i>
Hasta próximo <i>breakpoint</i> .	<i>F8</i>	<i>F5</i>
Hasta la línea en la que está el cursor ( <i>Run to Line</i> ).	<i>CTRL-R</i>	<i>F4</i>
Paso a paso, sin entrar en métodos ( <i>Step Over</i> ).	<i>F6</i>	<i>MAY-F8</i>
Hasta salir del método ( <i>Step Return</i> ).	<i>F7</i>	<i>CTRL-F7</i>
Finalizar la ejecución en depuración ( <i>Terminate</i> ).	<i>CTRL-F2</i> (o cuadro rojo)	<i>MAY-F5</i> (o cuadro rojo)

**Nota:** Para ver un ejemplo de depuración de código *Java* con *Eclipse* puedes ver en *Youtube* la *píldora informática 150*: <https://www.youtube.com/watch?v=ymV7IUUhkUU>

Otra de las acciones que permite el *debugger* es configurar los puntos de interrupción de modo que estén activados/desactivados o que sean condicionales. Supongamos que tenemos un bucle que hace 1000 iteraciones (desde *i=0* hasta *i=999*) y queremos observar lo que ocurre dentro del bucle en las dos últimas iteraciones. Establecemos un *breakpoint* dentro del bucle y el *debugger* se detendrá en él las 1000 veces, de modo que tendremos que usar la opción "*continuar hasta el próximo breakpoint*" 1000 veces.

Sin embargo podemos establecer ese *breakpoint* como condicional para que sólo esté activo si se cumple la condición "*i==998*". De esta manera, la ejecución se detendrá antes de las dos últimas iteraciones que es lo que necesitamos.

Para configurar los *breakpoints*:

<i>Eclipse</i>	<i>Netbeans</i>
<i>clic-dcho en el círculo del punto de interrupción/propiedades del punto de interrupción</i>	<i>clic-dcho en el cuadrado del punto de interrupción/breakpoint/properties.</i>
También podemos acceder a la configuración de los <i>breakpoints</i> en la ventana <i>puntos de interrupción</i> (visibilizable en <i>Ventana/Mostrar vista/Otras/Depurar</i> )	También podemos acceder a la configuración de los <i>breakpoints</i> en la pestaña adecuada de la ventana inferior. (Visibilizable en <i>Windows/Debugging/Breakpoints</i> )
Marcamos la casilla condicional y seleccionamos " <i>parar cuando true</i> " o " <i>parar cuando cambie el valor</i> ". A continuación escribimos la condición.	Activamos la casilla de la condición y escribimos la condición con la que queremos que se active el punto de ruptura

**Nota:** Para un ejemplo de *breakpoints* condicionales con *Eclipse* y algunas cosas más puedes ver en *Youtube* la *píldora informática 151*: <https://www.youtube.com/watch?v=qvNEQ2nAEVE>

### 3.5 JUnit

**JUnit** es una herramienta que permite hacer pruebas unitarias automatizadas. Está integrada tanto en **Eclipse** como en **Netbeans** por lo que no es necesario descargar ni instalar nada. Las pruebas unitarias se aplican sobre una clase de forma aislada, aunque a veces es imposible por su dependencia de otras clases.

**Nota:** La última versión de **JUnit** en el momento de escribir este documento es la versión 5, pero es habitual hablar de **JUnit Jupiter** que es el subproyecto de **JUnit 5** que nos interesa.

#### 3.5.1 Creación de una clase de prueba

Para cada clase de nuestro proyecto crearemos una clase que la prueba. Para crearla, primero nos situamos sobre la clase a probar y:

<b>Eclipse</b>	<b>Netbeans</b>
<b>click-dcho sobre el nombre de la clase/Nueva/Otras.../Java/JUnit/Caso de prueba JUnit</b>	<b>click-dcho sobre el nombre del proyecto/Nuevo/Otro/Unit Tetsts/Test for Existing Class/Siguiente</b>
Escribimos un nombre (aunque puedes dejar el que aparece por defecto).	
Al pulsar <b>siguiente</b> , nos permite seleccionar los métodos de la clase que queremos probar y <b>Finalizar</b> .	Seleccionamos en el desplegable ( <b>browse</b> ) la clase a testear.
Antes de completar la operación, nos pedirá que aceptemos la inclusión de la librería <b>JUnit</b> en nuestro proyecto, que debemos aceptar.	Pulsamos <b>Terminar</b> .

Tras seguir estos pasos se habrá creado una nueva clase que se usa exclusivamente para hacer pruebas a la clase original. Por tanto la nueva clase no formará parte del proyecto final.

Si estás trabajando con **Eclipse**, la nueva clase se genera en el mismo paquete que la original. Si estás trabajando con **Netbeans**, la nueva clase se genera en una estructura paralela que, en lugar de colgar de la carpeta **src**, cuelga de la carpeta **test**. En realidad, la ubicación de la nueva clase no es importante y, seguramente, la podrás cambiar de sitio. Recuerda que las clases de prueba no formarán parte del proyecto final.

En el ejemplo siguiente, para probar la clase **Enteros** creamos una nueva clase (**EnterosTest**) en la que aparece el esqueleto de los métodos que probarán nuestros métodos originales. Si en la clase **Enteros** hay un método que se llama **factorial()**, en la clase **EnterosTest** habrá un método que se llama **testFactorial()**.

**Ejemplo: Clase generada (en Eclipse) con JUnit Jupiter para testear la clase *Enteros.java*.**

**Nota:** La clase *Enteros* la tienes disponible entre los ejemplos de código del tema. Esa clase incluye 11 métodos que podremos usar como ejemplos. Al crear la clase de pruebas podemos decidir cuáles de los métodos queremos probar. El código siguiente corresponde a una situación en la que el usuario ha seleccionado solamente tres de los once métodos.

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class EnterosTest {

    @Test
    final void testFactorial() {
        fail("No implementado aun");
    }

    @Test
    final void testEsPrimo() {
        fail("No implementado aun");
    }

    @Test
    final void testDivide() {
        fail("No implementado aun");
    }
}
```

El resultado que obtendremos será una clase con tantos métodos como hayamos seleccionado y con el mismo nombre, pero precedidos de la palabra "**test**". Serán métodos que no devuelven ningún valor (**void**) y no reciben argumentos. Delante del método aparece la anotación **@Test** que indica al compilador que es un método de prueba.

**Nota:** En anteriores versiones de **JUnit** las clases y métodos de pruebas tenían que ser públicos. En la versión **5** no es necesario, lo cual supone una ventaja, porque no podrán ser invocados desde fuera de **JUnit 5**.

Cada uno de los métodos tendrá, originalmente, una llamada al método **fail()** con un mensaje que indica que no se ha implementado el método. Esta llamada hará que el test genere un fallo con ese mensaje. Habrá que sustituirla por el código correspondiente a nuestros casos de prueba.

**Nota:** Si usas **Netbeans** el código generado automáticamente será ligeramente distinto, pero en realidad no tiene importancia porque el cuerpo de los métodos tendrás que escribirlo tú, tanto en un **IDE** como en el otro.

### 3.5.2 Ejecución de casos de prueba

Algunos de los métodos que podemos usar en **JUnit** son los siguientes:

Método	Acción	Ejemplo
<b>assertTrue</b> (expresión booleana)	Comprueba que la expresión devuelve un valor true.	<b>assertTrue</b> (esPrimo(7))
<b>assertFalse</b> (expresión booleana)	Comprueba que la expresión devuelve un valor false.	<b>assertFalse</b> (esPrimo(4))
<b>assertEquals</b> (esperado, obtenido)	Comprueba si el valor esperado coincide con el valor obtenido.	<b>assertEquals</b> (24, factorial(4)) <b>assertEquals</b> (1.5, divide(3,2))
<b>assertEquals</b> (esperado, obtenido, delta)	Comprueba si el valor esperado coincide con el valor obtenido, dentro de la variación indicada. (para reales)	<b>assertEquals</b> (2.23, sqrt(5), 0.01) <b>assertEquals</b> (0.33, divide(1,3), 0.01)

Supongamos que tras analizar el método **esPrimo()** hemos concluido que los casos de prueba necesarios son estos dos (es un ejemplo, en realidad se necesitarán más casos de prueba):

<i>n</i>	resultado esperado
4	false
7	true

Entonces el método para testear el método **esPrimo()** será el siguiente:

```
@Test
final void testEsPrimo() {
    assertFalse(Enteros.esPrimo(4));
    assertTrue(Enteros.esPrimo(7));
}
```

Que viene a decir algo así como "comprueba que al llamar al método **esPrimo()** con el valor **4** devolverá **false** y que al llamar al método **esPrimo()** con el valor **7** devolverá **true**".

Por otra parte, hemos visto que hay dos métodos con el mismo nombre (**assertEquals**):

- El primero de ellos (dos argumentos) nos permite comprobar el resultado de un método que podemos expresar de forma exacta. Al dividir **3 entre 2** se debe obtener **1'5** que es un valor que podemos expresar de forma exacta:

```
assertEquals (1.5, Enteros.divide(3,2));
```

- El segundo de ellos (tres argumentos) nos permite comprobar el resultado de un método que **NO** podemos expresar de forma exacta. Al dividir **1 entre 3** se debe obtener **0,3333....** que es un valor que NO podemos expresar de forma exacta. Por tanto, expresamos una aproximación del número (**0,33**) y la diferencia (entre el resultado del método y la aproximación) que estamos dispuestos a admitir (**0,01**):

```
assertEquals (0.33, Enteros.divide(1,3), 0.01);
```

Comprueba que al dividir **1 entre 3** se obtiene aproximadamente **0'33** con una variación máxima de **0'01**.

**Nota:** El valor **delta** (**delta** es la letra con la que se suele expresar una variación en matemáticas) no puede ser negativo.

Otros métodos que nos resultarán muy útiles:

Método	Acción
<b>assertArrayEquals</b> (array esperado, array obtenido)	Para <b>arrays</b> de valores de cualquier tipo.
<b>assertArrayEquals</b> (array esperado, array obtenido, delta)	Para <b>arrays</b> de valores <b>float</b> o <b>doubles</b> .
<b>assertNotEquals</b> (inesperado, obtenido)	Comprueba que el valor inesperado no coincida con el valor obtenido.
<b>assertNotEquals</b> (inesperado, obtenido, delta)	Comprueba que el valor inesperado no coincida con el valor obtenido, dentro de la variación indicada. (para reales)
<b>assertNull</b> (objeto)	Comprueba que el objeto sea <b>null</b> .
<b>assertNotNull</b> (objeto)	Comprueba que el objeto no sea <b>null</b> .
<b>assertSame</b> (esperado, obtenido)	Comprueba si el objeto esperado y el obtenido son el mismo. Ojo! No que sean iguales, sino el mismo objeto.
<b>assertNotSame</b> (esperado, esperado)	Comprueba que el objeto esperado y el obtenido no sean el mismo.
<b>fail</b> ()	Genera un "fallo de prueba".

Puedes consultar la documentación completa de **JUnit** con todos sus paquetes, clases y métodos en <https://junit.org/junit5/docs/current/api/>. Y más concretamente, para la clase **Assertions** en <https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html>, donde se encuentran los métodos que estamos comentando.

Tienes la guía de usuario de **JUnit5** en inglés: <https://junit.org/junit5/docs/current/user-guide/>

También podemos usar todos estos métodos con un mensaje como argumento final. En ese caso cuando se produzca un fallo se lanzará el mensaje:

```
assertTrue(esPrimo(7), "Fallo en esPrimo(7)")
```

**Ejemplo: Pruebas sobre métodos de la clase Enteros.**

```
class EnterosTest {

    @Test
    final void testFactorial() {
        assertEquals(120, Enteros.factorial(4));
    }

    @Test
    final void testDivisible() {
        assertTrue(Enteros.divisible(15, 0));
    }

    @Test
    final void testEsPrimo() {
        assertFalse(Enteros.esPrimo(4));
        assertTrue(Enteros.esPrimo(7), "Fallo en esPrimo(7)");
    }

    @Test
    final void testPrimosHasta() {
        int[] esperado={2,3,5,7};
        assertEquals(esperado, Enteros.primosHasta(10));
    }

}
```



En este ejemplo estamos ejecutando:







- Un caso de prueba para el método **factorial()** (el factorial de 4 no es 120 (sino 24), se ha incluido a propósito para generar un **fallo**).
- Un caso de prueba para el método **divisible()** que generará un **error** (**ten en cuenta que un error es distinto a un fallo**) porque el método no controla la división por 0.
- Dos casos de prueba para **esPrimo()**, uno de ellos con mensaje en caso de error.
- Un caso de prueba para el método **primosHasta()**.

La distribución de los casos de prueba en métodos de prueba es libre. Podemos agrupar todos los casos en un método de prueba, o crear un método de prueba para cada método original o incluso varios métodos de prueba para cada método original, cada uno con su grupo de casos de prueba.

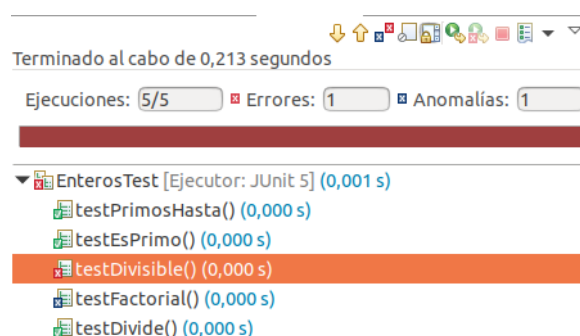
Para ejecutar las pruebas, nos situamos en la clase de pruebas de **JUnit** y hacemos:



<b>Eclipse</b>	<b>Netbeans</b>
<b>Ejecutar/Ejecutar como prueba JUnit</b>	<b>Ejecutar/archivo o Run/Run file o Run Test File</b>

Cada una de las pruebas puede generar:

- **Resultado correcto**: Se representa con un **stick verde** ( en **Eclipse** o  en **Netbeans**). El resultado esperado coincide con el resultado obtenido.
- **Fallo**: Se representa con un **aspa azul** (**Eclipse**) () o un **triángulo amarillo** (**Netbeans**) (). El resultado esperado no coincide con el resultado obtenido. Se ofrece una pequeña explicación que incluirá el mensaje que hemos incluido en el código (si es que lo incluimos).
- **Error**: Se representa con un **aspa roja** (**Eclipse**) () o una **admiración roja** (**Netbeans**) (). Un error indica que se ha generado una excepción inesperada.

Al lanzar la ejecución se ofrecerá un informe con el resultado de cada método de prueba. Si un método de prueba tiene un fallo o un error no continua su ejecución, se pasa al siguiente método. El informe para el ejemplo anterior tendría este aspecto en **Eclipse**:



En **Eclipse**, asegúrate que no están marcados los botones “**mostrar solo anomalías**” () y tests saltados “**mostrar skipped test**”. ()

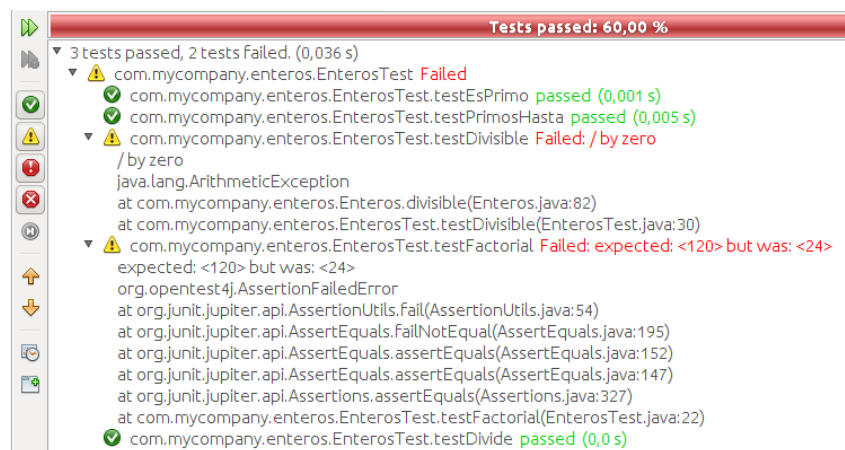
Hay que tener en cuenta que **JUnit** solo informa de una anomalía por cada método de prueba. Si en un mismo método de prueba hay dos o más casos de prueba que presentan anomalías (fallo o error) nos informa del primero de ellos.

El informe de errores es el siguiente:

- Se han ejecutado cinco pruebas con tres éxitos, un fallo y un error.
- La pruebas de los métodos **divide()**, **primosHasta()** y **esPrimo()** han sido exitosas (stick verde).
- La prueba del método **factorial()** ha terminado en fallo, es decir, ha dado un resultado distinto al esperado (aspa azul en **Eclipse** o triángulo amarillo en **Netbeans**).
- La prueba del método **divisible()** ha provocado un error (aspa o admiración roja). En este método no se ha programado correctamente la división por cero.

El informe para el ejemplo anterior tendría este aspecto en **Netbeans**:

**Nota Importante:** **JUnit 5** solo funciona desde **Netbeans 11** si el proyecto ha sido creado de **Java+Maven**, pero no funciona con **Java+Ant**. Además, como puedes ver en la imagen, muestra la excepción que genera el método **divisible()** como **fallo** y no como **error**. Quizá se deba a un mal funcionamiento de **Netbeans+JUnit5**, porque con **JUnit4** se mostraba como **error** en ambos IDE.



En **Netbeans**, bloquea los botones de lo que quieres mostrar: éxitos, fallos, errores y abortos.

### 3.5.3 Tests de prueba para métodos con excepciones

En algunas ocasiones, un método puede estar diseñado para que lance una excepción (por ejemplo, frente a datos de entrada incorrectos) que será manejada desde algún otro método que hace la llamada. En estos casos el lanzamiento de la excepción es correcto y lo que tenemos que probar con **JUnit** es que se lanza la excepción en la situación esperada.

Método	Acción
<b>assertThrows</b> (excepción, ejecutable)	Comprueba que el ejecutable (método) genera la excepción indicada.
<b>assertDoesNotThrow</b> (ejecutable)	Comprueba que el ejecutable (método) no genera excepciones.

En la clase **Enteros** tenemos un método (**divide()**) que genera una excepción cuando se le pasa un divisor igual a cero. Entre los casos de prueba deberá haber uno que contemple esa situación: "**cuando se reciba un divisor igual a cero se debe generar la excepción ArithmeticException**".

dividendo	divisor	resultado esperado
7	0	ArithmeticException

A continuación mostramos el que podría ser el método de prueba resaltando en amarillo el caso de prueba que genera la excepción:

```
@Test
final void testDivide() {
    assertEquals (1.5, Enteros.divide(3,2));
    assertEquals (0.33, Enteros.divide(1,3), 0.01);
    assertThrows (ArithmeticException.class, ()->Enteros.divide(7,0));
}
```

La llamada al método **assertThrows** tiene algunas cosas que seguramente te resultarán novedosas:

- El primer argumento que recibe es una **excepción** que, como bien sabes, es una clase **Java**. Hasta ahora hemos usado argumentos con valores o con variables o con objetos. En este caso pasamos la clase entera al método. Para hacer eso escribimos el nombre de la clase seguido de **.class**.
- El segundo argumento que recibe es un **ejecutable**, es decir, un trozo de código. Puede ser un trozo de código que se escribe ahí directamente (véase el ejemplo siguiente) o puede ser una llamada a un método. En ambos casos, para indicar que lo que se pasa al método no es el valor que devuelve el código, sino el código mismo, se le antepone el símbolo **()->**

En el siguiente ejemplo estamos comprobando que la llamada a **println()** no generará ninguna excepción:

```
assertDoesNotThrow (()->{System.out.println("aaa");});
```

### Ejemplo: Método clasifica de la clase Poblacion

Recordamos aquí el método **clasifica()** que ya hemos visto en este tema:

```
public static String clasifica(int edad, char genero) throws
EdadErroneaException, GeneroErroneoException {
    String respuesta = "";
    genero = Character.toUpperCase(genero);
    if (edad < 0 || edad > 125) {
        throw new EdadErroneaException(edad);
    }
    if (edad < 18) {
        respuesta = "menor";
    } else {
        if (edad < 65) {
            switch (genero) {
                case 'H':
                    respuesta = "trabajador";
                    break;
                case 'M':
                    respuesta = "trabajadora";
                    break;
                default:
                    throw new GeneroErroneoException(genero);
            }
        } else {
            switch (genero) {
                case 'H':
                    respuesta = "jubilado";
                    break;
                case 'M':
                    respuesta = "jubilada";
                    break;
                default:
                    throw new GeneroErroneoException(genero);
            }
        }
    }
    return respuesta;
}
```

En este método se pueden generar dos excepciones distintas. Fíjate en la cabecera del método:

```
public static String clasifica(int edad, char genero) throws
EdadErroneaException, GeneroErroneoException
```

Ya habíamos calculado los casos de prueba que necesita este método. Puedes verlos en la página 16. Esos 33 casos de prueba los podemos distribuir en un solo método o en varios, como queramos. En este caso hemos creado un solo método de prueba con los 33 casos de prueba:

```
public class PoblacionTest1 {

    @Test
    public void testClasifica() throws EdadErroneaException,
        GeneroErroneoException {
        assertEquals("menor", Poblacion.clasifica(15, 'H'));
        assertEquals("menor", Poblacion.clasifica(0, 'H'));
        assertEquals("menor", Poblacion.clasifica(17, 'H'));
        assertEquals("menor", Poblacion.clasifica(15, 'm'));
        assertEquals("menor", Poblacion.clasifica(0, 'm'));
        assertEquals("menor", Poblacion.clasifica(17, 'm'));
        assertEquals("menor", Poblacion.clasifica(15, 's'));
        assertEquals("trabajador", Poblacion.clasifica(50, 'H'));
        assertEquals("trabajador", Poblacion.clasifica(18, 'H'));
        assertEquals("trabajador", Poblacion.clasifica(64, 'H'));
        assertEquals("trabajadora", Poblacion.clasifica(50, 'm'));
        assertEquals("trabajadora", Poblacion.clasifica(18, 'm'));
        assertEquals("trabajadora", Poblacion.clasifica(64, 'm'));
        assertEquals("jubilado", Poblacion.clasifica(100, 'h'));
        assertEquals("jubilado", Poblacion.clasifica(65, 'h'));
        assertEquals("jubilado", Poblacion.clasifica(125, 'h'));
        assertEquals("jubilada", Poblacion.clasifica(100, 'M'));
        assertEquals("jubilada", Poblacion.clasifica(65, 'M'));
        assertEquals("jubilada", Poblacion.clasifica(125, 'M'));
        assertThrows(EdadErroneaException.class, ()->Poblacion.clasifica(-7, 'h'));
        assertThrows(EdadErroneaException.class, ()->Poblacion.clasifica(-1, 'h'));
        assertThrows(EdadErroneaException.class, ()->Poblacion.clasifica(-7, 'M'));
        assertThrows(EdadErroneaException.class, ()->Poblacion.clasifica(-1, 'M'));
        assertThrows(EdadErroneaException.class, ()->Poblacion.clasifica(-7, 'a'));
        assertThrows(EdadErroneaException.class, ()->Poblacion.clasifica(-1, 'a'));
        assertThrows(EdadErroneaException.class, ()->Poblacion.clasifica(200, 'h'));
        assertThrows(EdadErroneaException.class, ()->Poblacion.clasifica(126, 'h'));
        assertThrows(EdadErroneaException.class, ()->Poblacion.clasifica(200, 'M'));
        assertThrows(EdadErroneaException.class, ()->Poblacion.clasifica(126, 'M'));
        assertThrows(EdadErroneaException.class, ()->Poblacion.clasifica(200, 'o'));
        assertThrows(EdadErroneaException.class, ()->Poblacion.clasifica(126, 'o'));
        assertThrows(GeneroErroneoException.class, ()->Poblacion.clasifica(50, '?'));
        assertThrows(GeneroErroneoException.class, ()->Poblacion.clasifica(100, '9'));
    }
}
```

**Muy importante:** Al generar los métodos de prueba, como ya sabes, se hace una llamada al método que estamos probando. El **IDE** alertará que esa llamada puede generar excepciones y nos ofrecerá como siempre dos alternativas: rodear con **try/catch** o propagar la excepción. En los métodos de prueba **siempre se elegirá propagar la excepción**.

### 3.5.4 Anotaciones JUnit

Las **anotaciones Java** son metadatos que podemos añadir a cualquier elemento **Java** (tales como clases, paquetes, atributos, métodos, etc). Se escriben justo delante del elemento al que quieren modificar y comienzan por el símbolo arroba (@). Las anotaciones son un elemento general de **Java**, pero aquí nos centraremos en algunas de las anotaciones que usa **JUnit**. Puedes ver un listado exhaustivo en <https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations>.

- **@Test**: Se escribe justo antes de la cabecera de un método e indica que se trata de un método de prueba (sin parametrizar).
- **@ParameterizedTest**: Se escribe justo antes de la cabecera de un método e indica que se trata de un método de prueba parametrizado.
- **@ValueSource**: Se escribe entre **@ParameterizedTest** y la cabecera de un método de prueba parametrizado. Indica que la fuente de datos que se suministrarán al método será un flujo de valores individuales que se escriben justo tras la propia anotación. Ver ejemplos en el apartado siguiente y en el código del tema.
- **@CsvSource**: Se escribe entre **@ParameterizedTest** y la cabecera de un método de prueba parametrizado. Indica que la fuente de datos que se suministrarán al método será un flujo de conjuntos de valores (**Comma Separated Values**) que se indican justo tras la anotación. Ver ejemplos en el apartado siguiente y en el código del tema.
- **@MethodSource**: Se escribe entre **@ParameterizedTest** y la cabecera de un método de prueba parametrizado. Indica que la fuente de datos que se suministrarán al método será un flujo de conjuntos de argumentos que generará un método. El nombre del método se escribe justo tras la propia anotación. Ver ejemplos en el apartado siguiente y en el código que se proporciona con este tema.
- **@DisplayName**: Se escribe entre la anotación **@Test** y la cabecera de un método de prueba. Define el nombre con el que se anunciarán los resultados de este método en el informe de **JUnit**. Si se omite, usará el nombre del método.
- **@BeforeEach**: Se escribe delante de la cabecera de un método e indica que debería ejecutarse antes de cada método de prueba (**@Test** o **@ParameterizedTest**) Puede haber varios métodos con esta anotación. Por ejemplo, si necesitamos probar un método que lee datos de un fichero, quizás necesitamos un método que previamente cree ese fichero e introduzca datos. Ese método de preparación que debe ejecutarse antes de cada método de prueba irá precedido de **@BeforeEach**.
- **@AfterEach**: Se escribe delante de la cabecera de un método e indica que debería ejecutarse tras cada método de prueba (**@Test** o **@ParameterizedTest**) Puede haber varios métodos con esta anotación. Siguiendo el ejemplo anterior, quizá deseamos eliminar el fichero una vez terminada la prueba. Pues escribimos el método que elimina el fichero precedido de **@AfterEach**.
- **@BeforeAll**: Se escribe delante de la cabecera de un método e indica que debería ejecutarse antes de comenzar todas las pruebas. Solo puede haber un método con esta etiqueta. Siguiendo el ejemplo, si las tareas de preparación para todas las pruebas son las mismas, creamos el fichero que hace esas tareas precedido de **@BeforeAll**.
- **@AfterAll**: Se escribe delante de la cabecera de un método e indica que se ejecutará tras finalizar todas las pruebas. Solo puede haber un método con esta etiqueta.

### 3.5.5 Pruebas parametrizadas

Escribir una llamada al método de prueba para cada uno de los casos puede ser engorroso. Con las pruebas parametrizadas podemos incluir una batería de pruebas para cada método.

Las pruebas parametrizadas posibilitan ejecutar un mismo test varias veces con argumentos distintos (por ejemplo, para cada caso de prueba). Los métodos para pruebas parametrizadas van precedidos de la anotación **@ParameterizedTest** en lugar de **@Test**. Hay que declarar la fuente que proporciona los argumentos que se usarán en las distintas invocaciones al método. En la documentación puedes ver que hay múltiples posibilidades de fuentes desde las que alimentar los casos de prueba. Aquí veremos tres de ellas: **@ValueSource**, **@CsvSource** y **@MethodSource**.

#### @VALUESOURCE

Con **@ValueSource** la fuente de los argumentos es una lista de valores individuales. **@ValueSource** es una de las fuentes más simples. Permite especificar un array lineal de valores literales y solo vale para proporcionar un argumento simple en la invocación del test parametrizado.

**Ejemplo: @ValueSource. Valores que devuelven TRUE en el método esPrimo()**

```
@ParameterizedTest
@DisplayName("Los que sí son primos")
@ValueSource(ints = {7, 17, 19, 23})
final void testEsPrimo2(int candidato) {
    assertTrue(Enteros.esPrimo(candidato));
}
```

Este método se ejecutará 4 veces (la fuente de valores tiene 4 datos) y en cada invocación, el argumento **candidato** tomará sucesivamente los valores del array.

**Nota:** La palabra **ints** es un atributo de la anotación **@ValueSource** que indica que los datos proporcionados son un array de enteros. Otros atributos que podemos usar aquí son **floats**, **chars**, **strings**, **doubles**, etc. En <https://junit.org/junit5/docs/current/api/org.junit.jupiter.params/org.junit.jupiter.params.provider/ValueSource.html> tienes la lista completa.

**Nota:** Puedes ver los ejemplos de **@ValueSource** en la clase **EnterosTestParametrizados\_MUESTRARIO.java** en el código que se proporciona con este tema.

#### @CSVSOURCE

Si usamos **@CsvSource** la fuente que proporciona los argumentos es una lista de conjuntos de valores. Cada conjunto de valores es un **String** con los valores separados por comas (CSV-Comma Separated Values). **JUnit** hará implícitamente el casting de cada valor a su tipo.

**Ejemplo: @CsvSource. Conjuntos de valores de entrada y salida en el método clasifica()**

```
@ParameterizedTest
@DisplayName("Pruebas clasifica SIN GENERAR EXCEPCIONES")
@CsvSource({"menor, 15, H",
            "trabajador, 18, H",
            "trabajador, 64, H",
            "jubilada, 65, M",})
final void testClasifica(String salida, int edad, char genero) throws
    EdadErroneaException, GeneroErroneoException {
    assertEquals(salida, Poblacion.clasifica(edad, genero));
}
```



En este ejemplo vemos la ejecución de algunos casos de prueba que no deben generar excepciones. La fuente de valores CSV es un array de **Strings**. Cada **String** es un "**conjunto de valores separados por comas**". Como puedes ver en cada uno de esos conjuntos tenemos un **String**, un **int** y un **char** en ese orden. Observa que no se usan comillas para los **String** ni los **char**. **JUnit** hace el casting de forma implícita aunque debes preocuparte que estén en el orden adecuado coincidiendo con el orden que tienen los argumentos en el método **testClasifica()**.

**Nota:** Puedes ver ejemplos de **@CsvSource** en la clase **PoblacionTestParametrizados\_CSV.java**. Ambas clases se proporcionan en el código de este tema.

### **@METHODSOURCE**

Si usamos **@MethodSource** la fuente que proporciona los datos es un método que genera una lista de conjuntos de argumentos. Cada conjunto de argumentos se usará en una invocación al método de prueba.

Así pues por cada método de prueba parametrizado con **@MethodSource** habrá que escribir un método generador de datos que genere un flujo de argumentos para el método de prueba.

Cada método generador de datos debe generar un flujo de argumentos, y cada conjunto de argumentos de ese flujo será proporcionado al método de prueba parametrizado para invocaciones individuales. En general esto significa que el método generador tendría que devolver el tipo **Stream<Arguments>**, pero no necesariamente, porque en este contexto, un flujo es cualquier cosa que **JUnit** pueda convertir en **Stream**, tales como los propios **Stream** o colecciones, arrays, etc. Cada uno de los elementos de ese flujo serán los argumentos de una invocación individual del método de prueba. Esos argumentos pueden ser una instancia de **Arguments**, un array de objetos o un valor simple si el método **@ParameterizedTest** acepta un argumento simple.

En la siguiente tabla hay ejemplos en los que se muestra que tipo de dato debería devolver un método generador de datos, según la cabecera del método de prueba parametrizado, es decir, según los argumentos que recibe.

	método <b>@ParameterizedTest</b>	método generador de datos
1	void test(int)	static int[] factory()
2	void test(int)	static IntStream factory()
3	void test(String)	static String[] factory()
4	void test(String)	static List<String> factory()
5	void test(String)	static Stream<String> factory()
6	void test(String, String)	static String[][] factory()
7	void test(String, int)	static Object[][] factory()
8	void test(String, int)	static Stream<Object[]> factory()
9	void test(String, int)	static Stream<Arguments> factory()
10	void test(int[])	static int[][] factory()
11	void test(int[])	static Stream<int[]> factory()
12	void test(int[][])	static Stream<int[][]> factory()
13	void test(Object[][])	static Stream<Object[][]> factory()



Si el flujo proporciona arrays unidimensionales de objetos, cada uno de los elementos del array será uno de los argumentos del método de prueba. Fíjate en la fila 8 de la tabla. El método de prueba parametrizado está esperando un **String** y un **int**, y el método generador de datos proporciona un flujo de arrays de objetos, donde cada array tendrá exactamente dos objetos: un **String** y un **int**.

Sin embargo, si el flujo proporciona arrays multidimensionales, cada array multidimensional en bloque será considerado como un argumento individual. Fíjate en las filas 12 y 13 de la tabla. Los métodos de prueba parametrizados están esperando un argumento individual del tipo **int[][]** y **Object[][]** respectivamente.

**Ejemplo: Método de prueba parametrizado con @MethodSource para el método esPrimo()**

```
@ParameterizedTest
@DisplayName("Los que sí son primos -method-")
@MethodSource("datosEsPrimo4")
final void testEsPrimo4(int candidato) {
    assertTrue(Enteros.esPrimo(candidato));
}

static int[] datosEsPrimo4() {
    return new int[] {7, 17, 19, 23};
}
```

El método de prueba recibirá los datos de un método llamado **datosEsPrimo4()** tal y como se indica en la línea señalada con fondo amarillo.

A continuación se ha definido el método generador que devolverá una lista de 4 enteros. Cada uno de esos enteros se usará como argumento (**candidato**) en una invocación del método de prueba.

**Nota:** Este ejemplo se corresponde con la fila 1 de la tabla.

**Ejemplo: Método de prueba parametrizado con @MethodSource para el método alreves()**

```
@ParameterizedTest
@DisplayName("Pruebas alreves() -method-")
@MethodSource("datosAlreves3")
final void testAlreves3(String entrada, String salida) {
    assertEquals(salida, Enteros.alreves(entrada));
}

static String[][] datosAlreves3() {
    return new String[][] {{ "abc", "cba" }, { "a", "a" }, { "Hola", "aloH" }};
}
```

El método de prueba recibirá los datos de un método llamado **datosAlreves3()** tal y como se indica en la línea señalada con fondo amarillo.

A continuación se ha definido el método generador que devolverá una lista de 3 arrays de **String**. Cada uno de esos arrays es un conjunto de argumentos (**entrada**, **salida**) para una invocación del método de prueba.

**Nota:** Este ejemplo se corresponde con la fila 6 de la tabla.

**Ejemplo: Método de prueba parametrizado con @MethodSource para el método clasifica()**

```

@ParameterizedTest
@DisplayName("Pruebas clasifica SIN GENERAR EXCEPCIONES -method-")
@MethodSource("datosClasifica")
final void testClasifica(String salida, int edad, char genero)
throws EdadErroneaException, GeneroErroneoException {
    assertEquals(salida, Poblacion.clasifica(edad, genero));
}

static Object[][] datosClasifica() {
    return new Object[][] {{"menor", 15, 'H'},
                           {"trabajadora", 50, 'm'},
                           {"jubilado", 125, 'h'}
    };
}

```

El método de prueba recibirá los datos de un método llamado **datosClasifica()** tal y como se indica en la línea señalada con fondo amarillo.

A continuación se ha definido el método generador que devolverá una lista de 3 arrays de **Object**. Cada uno de esos arrays es un conjunto de argumentos (**salida**, **edad**, **genero**) para una invocación del método de prueba. Observa que cada uno de los argumentos es de un tipo distinto: **String**, **int**, **char**. Por eso el tipo tiene que ser **Object**, que es el más genérico y los admite a los tres.

**Nota:** Este ejemplo se corresponde con la fila 7 de la tabla, aunque en este ejemplo hay un argumento más.

**3.5.6 Pruebas parametrizadas con excepciones**

Además del método de prueba con todos los casos en los que no se generan excepciones, será necesario crear un método de prueba más para cada una de las posibles excepciones. Así, puesto que el método **clasifica()** puede generar dos excepciones distintas, hay que crear tres métodos: uno para los casos que no generan excepción, otro para los casos que generan la excepción **GeneroErroneoException**, y otro para los casos que generan la excepción **EdadErroneaException**.

En el siguiente ejemplo se muestra la clase de pruebas para la excepción **GeneroErroneoException**, faltaría una clase similar para la excepción **EdadErroneaException**.

```

@ParameterizedTest
@DisplayName("Pruebas clasifica GENERANDO EXCEPCIÓN GENERO -csv-")
@CsvSource({"50, ?", "100, 9"})
final void testClasificaExcGenero(int edad, char genero ) {
    assertThrows(GeneroErroneoException.class, ()->Poblacion.clasifica(edad, genero));
}

```

**Nota:** En la clases **PoblacionTestParametrizados\_CSV** y **PoblacionTestParametrizados\_METHOD** puedes ver dos ejemplos distintos en los que se contemplan todos los casos de prueba que se han hallado en este tema para el método **clasifica()**.