

PROGRAMACIÓN

## Excepciones y depuración

---

# ÍNDICE

<b>/ 1. Introducción y contextualización práctica</b>	<b>3</b>
<b>/ 2. Definición y control de las excepciones</b>	<b>4</b>
2.1. Control de excepciones	4
<b>/ 3. Caso práctico 1: “Manejando muchas excepciones”</b>	<b>5</b>
<b>/ 4. Lanzar excepciones</b>	<b>6</b>
<b>/ 5. Propagar excepciones</b>	<b>6</b>
<b>/ 6. Creando excepciones propias</b>	<b>7</b>
<b>/ 7. Documentación de excepciones</b>	<b>8</b>
<b>/ 8. Depuración de programas</b>	<b>8</b>
8.1. El depurador de NetBeans	9
8.2. Puntos de ruptura	10
<b>/ 9. Caso práctico 2: “¿Dónde está el error?”</b>	<b>11</b>
<b>/ 10. Pruebas de programa</b>	<b>11</b>
10.1. Tipos de pruebas	12
<b>/ 11. Resumen y resolución del caso práctico de la unidad</b>	<b>13</b>
<b>/ 12. Bibliografía</b>	<b>13</b>

# OBJETIVOS



*Conocer y gestionar excepciones.*

*Propagar excepciones.*

*Crear nuevas excepciones.*

*Conocer el uso de un depurador de código.*

*Conocer diferentes tipos de pruebas.*



## / 1. Introducción y contextualización práctica

En este tema, hablaremos sobre los errores en los programas y qué son las excepciones. Veremos cómo tratar un error o excepción para que el programa siga funcionando después de que ocurra.

También, aprenderemos a crear nuestras propias excepciones para manejar errores personalizados y cómo propagarlas correctamente.

Por otro lado, veremos cómo depurar un programa para localizar dónde están los errores.

Y por último, conoceremos algunos tipos de pruebas que podremos aplicar a nuestros programas para verificar su funcionamiento.

Son conceptos relativamente nuevos, pero de gran utilidad a la hora de aplicarlos a lo que ya sabemos.

Escucha el siguiente audio en el que planteamos el caso práctico que iremos resolviendo a lo largo de esta unidad.



Fig. 1. Errores en los programas.



Audio intro. "Errores en el código"

<https://bit.ly/2F4UDhn>



## / 2. Definición y control de las excepciones

A lo largo de todas las unidades anteriores, nos hemos encontrado con que nuestros programas han provocado errores tras hechos tan simples como olvidarnos de terminar una instrucción con un punto y coma, cerrar unas comillas, etc. Errores que el propio compilador nos avisa y no nos deja compilar hasta que los arreglemos.

Existen otro tipo de errores, concretamente, **los errores en tiempo de ejecución**. Estos son los que se dan durante la ejecución del programa, es decir, el compilador no nos avisa que vamos a tenerlos, sino que los descubriremos por nosotros mismos.

Este tipo de errores son los más “peligrosos”. Cuando tenemos un error en tiempo de ejecución, el programa terminará de forma abrupta, cerrándose sin dar la oportunidad de guardar las acciones que se hayan realizado, forzando al usuario a volver a ejecutarlo, y a rehacerlo todo de nuevo.

El manejo de las excepciones va a hacer que los programadores tengan que crear un bloque exclusivo para el tratamiento de error. Este provoca que el programa, cuando lo detecte, vaya directo a ejecutar ese bloque, consiguiendo que no se cierre de forma abrupta, y por tanto, permitiendo al usuario volver a realizar la acción que ha provocado el error de una forma segura.

Con el tratamiento de excepciones, vamos a conseguir que los usuarios no noten que el programa ha fallado, aunque lo más recomendable es avisar al usuario de que ha ocurrido un fallo. No obstante, el programa se ha recuperado de forma exitosa de ello.

Algunos ejemplos que pueden provocar una excepción son: una división entre cero, el intentar acceder a una posición no válida de un array, un desbordamiento aritmético, un acceso fallido a disco, etc.

Las excepciones en Java heredan todas de la clase *Exception*.



Fig. 2. Las excepciones son parte de los programas.

### 2.1. Control de excepciones

En Java, el control de excepciones lo vamos a realizar mediante los **bloques try-catch**.

Estos bloques nos van a permitir “aislar” un fragmento de código que sea susceptible de producir un error, e indicar el código que queremos que se ejecute en caso de que ese error se produzca.

En el **bloque try**, vamos a escribir todo el código que sea susceptible de producir uno o varios errores. Este código irá en el cuerpo del *try*, que estará definido mediante llaves ( { } ).

En el **bloque catch**, vamos a escribir todo el código que queremos que se ejecute para tratar el error que se capture.

Hay excepciones que las detectará automáticamente el propio compilador y nos avisará de que tenemos que poner un bloque try-catch, pero en otras ocasiones, no lo hará y lo descubriremos nosotros mismos cuando el programa falle.

Cuando se produce una excepción, NetBeans nos mostrará dónde ha ocurrido y el tipo de excepción que se ha generado, pudiendo así, de una forma sencilla, encapsular ese fragmento de código en un bloque try-catch y controlar esa excepción.

**Un try puede tener uno o varios catch, pero un catch no puede estar sin un try.** Cuando un *try* tiene varios *catch* podemos agruparlos en un **multicatch**, tratando todas esas excepciones de igual forma. Para separar las diferentes excepciones en un *multicatch*, utilizaremos el símbolo ‘|’ entre los nombres de las mismas.



En el *catch*, podemos ver que tenemos un objeto de una clase que representa una excepción. Mediante este objeto, podremos saber cuál ha sido el motivo del error.

Una vez que se produzca un error, se pasará el control directamente al *catch*, provocando que todo el código que se encuentre por debajo de la línea que ha provocado el error no se ejecute.

En la siguiente figura, podemos ver un ejemplo de bloque *try-catch*.

```
public static void main(String[] args) {  
    int numero1 = 3, numero2 = 0;  
  
    try  
    {  
        // La división entre 0 provoca un error  
        double division = numero1 / numero2;  
        System.out.println("La división es: " + division);  
    }  
    catch(ArithmeticException error)  
    {  
        // Mostramos la causa del error  
        System.out.println("Error: " + error.toString());  
    }  
}
```

Código 1. Ejemplo de try catch.

## / 3. Caso práctico 1: “Manejando muchas excepciones”

**Planteamiento:** Pilar y José están realizando un programa en el que tienen que tratar varias excepciones, concretamente, han de tratar 6 excepciones diferentes que pueden ocurrir en el fragmento de código que están desarrollando.

Pilar le dice a José: “Con tantas excepciones a tratar, vamos a tener más código en los bloques *catch* que código a escribir en la parte *try*”. José se queda pensativo y asiente: “Tienes razón, seguro que lo podemos arreglar de alguna forma”, le dice a Pilar.

**Nudo:** ¿Qué piensas sobre ello? ¿Cómo crees que se puede elaborar el código de forma más ordenada?

**Desenlace:** Cuando estemos tratando varias excepciones, podemos tener la misma situación de Pilar y José, y por raro que parezca, es más habitual de lo creemos.

Cuando hay una única excepción a tratar, o incluso dos, podemos poner uno o dos bloques *catch* en nuestro *try*, pero cuando ya son muchas, el desarrollo del código se puede tornar un poco lioso a la hora de gestionar las excepciones.

En estas situaciones, lo más habitual y recomendable es el uso del *multicatch*, ya que gracias a él, podremos agrupar en un único *catch* varias excepciones, consiguiendo así reducir de forma considerable la cantidad de líneas de código de nuestro programa.

Hay que tener cuidado, ya que, como sabemos, en programación no es bueno generalizar. Cuando tenemos varias excepciones, podremos usar el *multicatch* cuando queramos darles el mismo tratamiento a todas, pero cuando queramos hacer alguna distinción, tendremos que usar más de un *catch*, aunque siempre podremos utilizar varios *multicatch* para mitigar en lo posible esto.

```
public static void main(String[] args) {  
    try {  
        double resultado = 9 / 7;  
        System.out.println(resultado);  
    }  
    catch (NullPointerException e) {  
        System.out.println("Error");  
    }  
    catch (ArithmeticException e) {  
        System.out.println("Error");  
    }  
}
```

Código 2. Ejemplo de multicatch.



## / 4. Lanzar excepciones

Ya hemos visto cómo podemos tratar un fragmento de código que provoca excepciones mediante un bloque *try-catch*.

Puede ocurrir que, en algún punto, necesitemos lanzar una excepción por nosotros mismos, ya sea porque estemos ante una excepción personalizada, o por especificaciones del proyecto que estemos desarrollando.

Imaginemos, por ejemplo, que estamos realizando una aplicación donde estamos realizando un registro de personas. Estas personas, entre muchos atributos, tendrán una edad, y un requisito para poder llevar a cabo el registro es que esas personas sean mayores de edad, es decir, que la edad sea mayor o igual que 18. Podemos gestionar nuestro código para que cuando esto ocurra, se lance una excepción de forma automática.

Para lanzar excepciones, podemos usar la instrucción *throw*, a la cual deberemos indicar la excepción que queremos que lance, utilizando su constructor con parámetros e indicando el mensaje que queremos que aparezca cuando se consulte el motivo del error.

La instrucción *throw* solo puede lanzar una excepción a la vez.

Cuando lanzamos una excepción, rompemos la ejecución secuencial que pudiera venir trayendo el programa, es decir, se cede el control al bloque *catch*. Al terminar de ejecutarse éste, el programa continuará por la primera instrucción que haya a continuación del bloque.

Recaltar que en un programa podremos lanzar todas las excepciones que sean necesarias.

Un ejemplo de lanzamiento de excepciones lo podemos ver en la siguiente figura:

```
public static void main(String[] args) {  
    Scanner teclado_int = new Scanner(System.in);  
    int edad;  
  
    try  
    {  
        System.out.println("Introduce la edad de la persona");  
        edad = teclado_int.nextInt();  
        if( edad < 18 )  
        {  
            throw new Exception("La persona ha de ser mayor de edad");  
        }  
    }  
    catch(Exception error)  
    {  
        // Mostramos la causa del error  
        System.out.println("Error: " + error.toString());  
    }  
}
```

Código 3. Lanzando una excepción.

## / 5. Propagar excepciones

Una vez aprendido cómo podemos controlar las excepciones y cómo podemos lanzarlas según nuestras necesidades, vamos a ver cómo podemos propagar una excepción por nuestro programa de forma correcta para que pueda ser tratada en el programa principal.

En la propagación de excepciones, entran en juego los métodos. Ya sabemos qué es un método y cómo funciona, pero ¿qué ocurre cuando se produce un error en un método?

Cuando se produce una excepción en un método, podemos propagarla para que el bloque de código donde este fue invocado se encargue de ella.



Para propagar una excepción en un método, usamos la instrucción *throws*, a la cual podremos indicar cuál o cuáles excepciones lanzan el método.

Para indicar que queremos propagar más de una excepción en un método, pondremos una lista de ellas separadas por comas.

La sintaxis de un método que propaga excepciones es la siguiente:

```
[public / private] [tipo devuelto / void] nombreMetodo ([parámetros]) throws Excepcion1, Excepcion2
```

Una vez que un método tiene la sentencia *throws* para indicar que puede propagar una o más excepciones, cuando lo invoquemos desde cualquier parte de nuestro programa, el compilador nos mostrará un error que nos dirá que el método llamado provoca una excepción, teniendo que usar un bloque *try-catch* para su uso.

Puede ocurrir que haya excepciones que ya contengan implícitamente a otras, es decir, que en el abanico de posibilidades de error que abarca una excepción, estén cubiertas todas las posibilidades que abarca otra. En estos casos, tendremos que eliminar la excepción contenida, ya que con la excepción que la contiene, es más que suficiente para el tratamiento de los posibles errores.

```
public double division(double dividendo, double divisor) throws ArithmeticException
{
    if( divisor != 0 )
        return dividendo / divisor;
    else
        throw new ArithmeticException("El divisor no puede ser 0");
}
```

Código 4. Propagando excepciones.



Audio 1. "Propagando excepciones de forma correcta"  
<https://bit.ly/3IOU2B4>



## / 6. Creando excepciones propias

En Java, al igual que en muchos otros lenguajes de programación, además de las excepciones definidas en el propio lenguaje, vamos a poder crear nuestras propias excepciones para tratar casos específicos.

Ya hemos visto que todas las excepciones heredan de la clase *Exception*. Esto quiere decir que si en un bloque *try-catch* atrapamos la excepción *Exception*, atraparemos todas y cada una de las excepciones que se produzcan en el bloque *try*.

Cuando creamos una excepción, a la hora de nombrarla, esta deberá terminar en **Exception**, no como forma obligatoria, sino como parte de un estándar para que, al verla, se sepa de inmediato que representa un posible error en nuestro código. Por ejemplo, si queremos crear una excepción para que se lance cuando un DNI introducido no sea válido, podríamos llamarla *DNIException*.

Una excepción va a ser una clase nueva que tendrá que heredar de la clase *Exception*, pero solo con esto no basta, tendremos que implementar el constructor con parámetros de esta nueva clase, al que le llegará como parámetro un objeto de tipo *String*, que será el mensaje que propagará nuestra nueva excepción.

Estas nuevas excepciones podrán ser utilizadas en los bloques *catch*, pudiendo lanzarlas cuando lo necesitemos mediante la instrucción *throw*, y propagarlas en los métodos mediante la instrucción *throws*. A fin y al cabo, lo que tenemos es una excepción totalmente funcional.



En un programa, podremos crear tantas excepciones como necesitemos.

A la hora de crear nuestras propias excepciones es importante asegurarse bien de que no estén ya cubiertas por las que Java nos ofrece de forma interna, para así evitar la duplicación innecesaria de código.

La lista de excepciones internas que nos proporciona Java la podemos consultar en el siguiente enlace:

<https://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html>



Vídeo 1. "Creación y uso de una excepción"

<https://bit.ly/31WS0XD>



## / 7. Documentación de excepciones

En unidades anteriores, ya vimos la necesidad de documentar los programas, ya que así, será mucho más fácil la tarea de leerlos para entender qué hacen.

Vimos que se tenían que documentar:

- Métodos.
- Clases.
- Código mediante comentarios de una línea o multilínea.

Las excepciones no son menos, y también habrá que documentarlas, ya que así será más fácil saber qué tipo de errores tratan. Cuando creamos una excepción propia, la documentación será la misma que se utiliza al documentar una clase, ya que, al fin y al cabo, lo que creamos es una clase.

Cuando tenemos un método que lanza una o varias excepciones, tendremos que agregar la etiqueta `@throws` por cada una de las excepciones que lancemos.

La descripción que debemos poner en cada `@throws` es el caso en el que se lanzará esa excepción. Por ejemplo, imaginemos que tenemos un método que lanza una excepción cuando se divide por cero, entonces, deberíamos indicar que esa excepción se va a lanzar al producirse una división entre cero.

Si documentamos correctamente todas las excepciones lanzadas, al generar la documentación con *Doxygen*, aparecerán todas y cada una de esas excepciones bien explicadas en los métodos, siendo así mucho más fácil de usar, entender y mantener nuestro código.

```
/**
 * Este método calcula la media de los elementos de un array
 * @param array Array para calcular la media
 * @return Media de los elementos del array
 * @throws ArithmeticException Se lanzará si el array está vacío
 */
public double mediaArray(ArrayList<Integer> array) throws ArithmeticException
{
    double media = 0;

    for (int i = 0; i < array.size(); i++) {
        media += array.get(i);
    }

    media /= array.size();

    return media;
}
```

Código 5. Documentación excepciones.

## / 8. Depuración de programas

Ya hemos adelantado que vamos a tener muchos errores de código en nuestros programas; desde errores en tiempo de compilación, de los que nos avisará el propio compilador, hasta errores en tiempo de ejecución, de los que el compilador no nos avisará y la única forma que vamos a tener de descubrir dónde están es que se provoquen.





Puede ser que estos errores no aparezcan en una primera ejecución, sino que lo hagan pasado un tiempo y al introducir unos datos específicos en nuestros programas, o puede ser que sí se presenten desde un inicio, pero que se produzcan unas veces sí y otras no, siendo muchísimo más complicada su detección, y, por ende, el poder solucionarlos.

Para ayudarnos a lidiar con estos errores, aparte de la experiencia que podamos ir adquiriendo, tenemos a los depuradores de código.

Todos los entornos de desarrollo medianamente completos suelen traer un depurador de código ya integrado, aunque siempre podemos usar un depurador externo a nuestro IDE. En el caso de NetBeans, ya trae su propio depurador integrado cuyo funcionamiento es bastante preciso.

Las operaciones que nos va a permitir realizar un depurador son las siguientes:

- **Rastreo:** Nos va a permitir ver el código fuente que se está depurando.
- **Punto de ruptura:** Un punto de ruptura es un punto donde vamos a parar la ejecución del programa para poder depurar ese punto.
- **Ventanas de seguimiento:** En estas ventanas, el depurador nos va a permitir ver el valor de las variables que hay en memoria en ese instante.
- **Reinicialización:** Reinicia la ejecución del programa.
- **Modificación del valor de las variables:** El depurador permitirá cambiar el valor de las variables implicadas en la ejecución del programa.

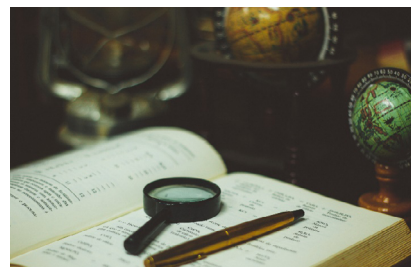


Fig. 3. La depuración es una ardua tarea pero vital.

## 8.1. El depurador de NetBeans

Como hemos adelantado, el IDE NetBeans ya trae integrado su propio depurador, siendo este bastante **completo y fácil de usar**. Para ejecutar el depurador de NetBeans, debemos pulsar el botón de depurador, el cual se muestra en la siguiente figura.



Fig. 4. Botón para lanzar el depurador de NetBeans.

Esta opción se encuentra justo a la derecha del botón de compilar programas. Una vez pulsemos el botón del depurador, comenzará el proceso y el programa se compilará y ejecutará, igual que cuando pulsamos el botón de compilar, solo que, esta vez, tendremos el depurador abierto.

Con este abierto, podremos controlar la línea de código que se va ejecutando, pasar a la siguiente línea, a la anterior, o al siguiente bloque que se vaya a ejecutar, habilitándose los siguientes botones:

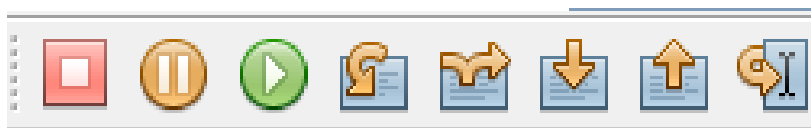


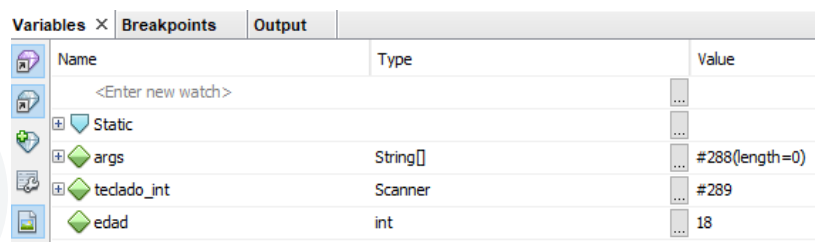
Fig. 5. Botones del depurador.



En orden de izquierda a derecha, tenemos:

- Botón para terminar el proceso de depuración.
- Botón para pausar el proceso de depuración.
- Botón para continuar el proceso de depuración.
- Botón para ir a la siguiente línea de código.
- Botón para ir a la siguiente función que se ejecutará.

También, nos aparecerá la ventana de control de variables, donde visualizaremos todas las variables que están implicadas en la ejecución del código con sus respectivos valores. Podremos cambiar esos valores desde esta ventana si es necesario.



Variables	Breakpoints	Output
Name	Type	Value
<Enter new watch>		...
Static		...
args	String[]	#288(length=0)
teclado_int	Scanner	#289
edad	int	18

Fig. 6. Ventana de control de variables.

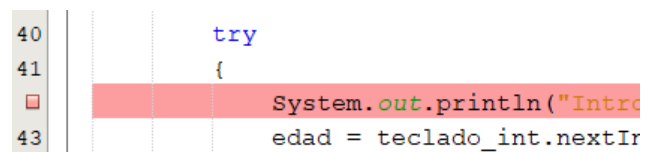
## 8.2. Puntos de ruptura

Ya hemos visto que los puntos de ruptura son una de las muchas herramientas que nos va a proporcionar cualquier depurador de código, entre ellos, el de NetBeans.

Indicando un punto de ruptura, vamos a conseguir que cuando estemos depurando un programa, el depurador se pare justo en esa línea y podremos observar la ventana de control de variables para ver qué valores tienen, pudiendo **deducir posibles errores**.

Podremos poner tantos puntos de ruptura como líneas tenga nuestro programa. Para la detección de errores, una buena estrategia es poner un punto de ruptura justo en la línea anterior a la del error, así podremos ver los valores de las variables implicadas y será más fácil ver qué puede estar provocando el error.

Para colocar un punto de ruptura en un programa con NetBeans, debemos pulsar en la parte izquierda sobre el número de la línea donde lo queremos poner. Una vez hecho esto, la línea se pondrá de color rojo, indicando que hay un punto de ruptura. Para poder eliminar dicho punto, basta con volver a pulsar sobre el número de la línea y dejará de ser rojo.



```
40  
41  
42 try  
43 {  
    System.out.println("Introduzca su edad:");  
    edad = teclado_int.nextInt();  
}
```

Fig. 7. Punto de ruptura en la línea 42.

Cuando ejecutemos el depurador, se parará automáticamente en la línea 42, mostrando la ventana de control de variables y podremos ver qué variables hay implicadas y cuáles son sus valores en ese preciso momento. Podemos poner puntos de ruptura en diferentes métodos de clases de nuestro programa, y cuando el depurador vaya llegando a ellos, se irá parando automáticamente. Si ejecutamos normalmente un programa con puntos de ruptura no ocurrirá nada especial, sino que el programa se ejecutará como si nada.



Vídeo 2. "Ejemplo de depuración de un programa"  
<https://bit.ly/2GtOkVg>





## / 9. Caso práctico 2: “¿Dónde está el error?”

**Planteamiento:** Pilar y José están realizando una aplicación bastante complicada que consiste en la gestión de una barra de un bar.

En este proyecto, tendrán que poder gestionar todos los pedidos de la barra por clientes, y obtendrán las cuentas a pagar de cada pedido. En toda esta gestión, pueden producirse errores, como es lógico en cualquier programa, aún más si entraña cierta complicación.

Pilar y José, en su implementación de la práctica, están teniendo un caso que no les había ocurrido nunca. A veces, al insertar un producto de un pedido ocurre un error sin sentido, y otras veces no. Nuestros amigos están bastante confundidos con ello y no saben a qué se debe este extraño error.

**Nudo:** ¿Te ha ocurrido esto a ti también? ¿Cómo crees que pueden solucionarlo?

**Desenlace:** Cuando realizamos un programa, estamos expuestos a los errores, eso no hay forma de evitarlo. Debido a esto, tenemos que tener mucho cuidado e ir controlando todas las excepciones posibles que podamos, pero, como siempre, se nos puede escapar alguna, hecho muy normal, sobre todo, cuando estamos empezando. Puede ocurrir que aparezcan errores que se produzcan por circunstancias que no entendamos, como es el caso de Pilar y José, que al introducir un producto de un pedido, a veces, tienen un error y otras no.

Cuando esto ocurre, nuestro mejor aliado es el depurador de código. Lo más recomendable es intentar provocar el error de nuevo, pero, esta vez, utilizando el depurador y ejecutando línea a línea para poder controlar el valor de las variables del programa. De esta forma, el error acabará por aparecer y sabremos cuál es su causa.

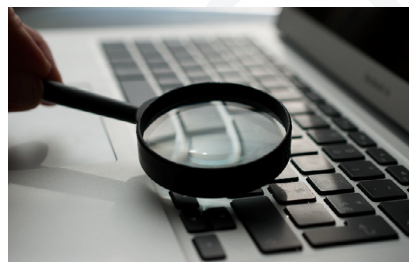


Fig. 8. Hay que tener paciencia buscando los errores.

## / 10. Pruebas de programa

En este punto que nos encontramos, ya sabemos perfectamente que los programas no van a funcionar a la primera, y que tarde o temprano van a aparecer fallos en nuestro código. Unos fallos serán más fáciles de encontrar y otros, no tanto.

Para encontrar los fallos, tenemos a nuestra disposición una herramienta muy poderosa que **engloba los elementos de depuración** estudiados hasta ahora: las pruebas de código.

Estas pruebas son importantes, hasta el punto de existir una fase del proceso de creación de software dedicada a ellas. Realizar pruebas en nuestros programas va a consistir, básicamente, en probar a **ejecutarlos cerciorándonos que estos no fallan al proporcionar los valores esperados para unas entradas específicas**.

Mientras sea mayor la cantidad de pruebas a las que sometamos nuestros programas, tendremos más seguridad de que estos funcionan a la perfección. Aunque siempre puede ocurrir que haya errores que no se detecten con estas pruebas y salgan a la luz a largo plazo, teniendo que modificar nuestro código para tratar dichos errores y volver a realizar pruebas.

Las pruebas y la depuración de código están íntimamente ligadas, ya que, con las pruebas, podemos detectar los errores, y gracias a la depuración, podremos encontrarlos y solucionarlos.

Cuando realizamos pruebas, estamos verificando que nuestro programa funciona correctamente, es decir, hace lo que se espera según su especificación. A mayor cantidad y variedad de pruebas realizadas, más verificado estará nuestro programa.

Normalmente, en los equipos de desarrollo, quienes realizan las pruebas son un equipo diferente al de codificación para tener la máxima independencia unos de otros.

Un ejemplo de prueba muy conocida son las versiones alfa y beta de los videojuegos. Las versiones alfa son una primera versión del videojuego que se entrega a una serie de testers para que lo prueben e informen de los fallos encontrados, como pueden ser fallos en texturas, en movimientos, etc., mientras que las betas, son una versión ya terminada del videojuego, pero no comercializada, entregada a los testers para el mismo propósito.



Fig. 9. Testeando código.

## 10.1. Tipos de pruebas

Existen varios tipos de pruebas de código, entre las que podemos destacar:

- Pruebas exhaustivas.
- Pruebas de cubrimiento.
- Pruebas de caja negra.
- Pruebas de caja blanca.
- Pruebas de valores límite.
- Pruebas de clases de equivalencia.

Las pruebas **exhaustivas** son pruebas en las que seleccionamos todas las posibles combinaciones de valores de entrada y se las pasamos a nuestro programa para que opere con ellas, comprobando si produce los resultados esperados.

Las pruebas de **cubrimiento** son un tipo de pruebas en las que comprobaremos que todos los posibles caminos que se puedan dar en un bloque de código, normalmente un método, se pueden llegar a dar y no hay ninguno que quede aislado.

Las pruebas de **caja negra** son un tipo de pruebas en las que se comprueba que el programa obtenga los resultados que se esperan, sin llegar a mirar el código.

Con las pruebas de **caja blanca** se comprueba que el código funciona correctamente, que se han recorrido todos los caminos, no hay variables que no se usan, etc.

Las pruebas de **valores límite** son pruebas en las que los valores que se toman para la prueba son los que hay en los límites, por ejemplo, si tenemos un método que compruebe que una persona es mayor de edad, sus pruebas límite se harán con los valores 17 y 18, ya que son los límites para que sea o no mayor de edad.

Las pruebas de **clases de equivalencia** son un tipo de prueba donde cada caso que se prueba intentará cubrir el mayor número de entradas posible.



Fig. 10. Tester.



## / 11. Resumen y resolución del caso práctico de la unidad

A lo largo de esta unidad, hemos visto que podemos manejar de una forma relativamente simple los fallos en nuestros programas, o como se conoce a estos, las **excepciones**, mediante el bloque *try-catch*.

También, hemos aprendido cómo **propagar una excepción** ocurrida de forma correcta, así **como a crear nuestras propias excepciones** de una forma muy sencilla, pudiéndolas utilizar como una excepción más.

Hemos conocido, además, el **depurador de código** que trae integrado NetBeans, donde podemos seguir la traza de una excepción para poder averiguar cuál es su causa mediante los **puntos de ruptura** y la ventana de control de variables.

Por último, hemos estudiado qué son las **pruebas de código**, para qué sirven, su gran importancia y algunos de los tipos de pruebas que existen y podemos aplicar a nuestros programas.

### Resolución del caso práctico de la unidad

Cuando los programas empiezan a ser más complicados, es normal que aparezcan errores de código que no sabemos de dónde vienen.

Las excepciones nos van a ayudar en esto, ya que según la excepción que se produzca, podremos saber qué tipo de error ha ocurrido, y sumando a esto que NetBeans nos da una estimación de dónde ha ocurrido el error, nos será un poco más sencilla su localización.

Con el tratamiento de excepciones, vamos a conseguir que cuando una de ellas se produzca, podamos tratarla y hacer que el programa no se cierre inesperadamente, pudiendo volver a su ejecución normal.



Fig. 11. Buscando el error.

## / 12. Bibliografía

Colaboradores de Wikipedia. (2020, abril 17). *Manejo de excepciones*. Wikipedia, la enciclopedia libre. Recuperado 26 de mayo de 2020, de [https://es.wikipedia.org/wiki/Manejo\\_de\\_excepciones](https://es.wikipedia.org/wiki/Manejo_de_excepciones)

Colaboradores de Wikipedia. (2020b, mayo 18). *Depurador*. Wikipedia, la enciclopedia libre. Recuperado 26 de mayo de 2020, de <https://es.wikipedia.org/wiki/Depurador>

Colaboradores de Wikipedia. (2020, marzo 28). *Pruebas de software*. Wikipedia, la enciclopedia libre. Recuperado 26 de mayo de 2020, de [https://es.wikipedia.org/wiki/Pruebas\\_de\\_software](https://es.wikipedia.org/wiki/Pruebas_de_software)

Turnes, Y. (s. f.). Combo de definiciones. *Versiones de desarrollo*. Recuperado 26 de mayo de 2020, de <http://www.gamerdic.es/combo/versiones-de-desarrollo/>