

Excepciones en Java

Al ejecutar código Java pueden ocurrir diferentes **errores**: errores de codificación cometidos por el programador, errores por entradas incorrectas u otras cosas imprevisibles. Cuando ocurre un error, el programa **se detendrá** y generará un **mensaje de error**.

Esto es lo que se conoce como **generar una excepción**.

Cuando programamos tenemos que tener en cuenta estos errores. En ocasiones, sobre todo cuando usamos librerías hechas por terceros, ciertos métodos vienen ya preparados para gestionar estos errores. En otras ocasiones, tenemos que **conocer** qué errores genera un método para gestionarlos nosotros. Finalmente, también tenemos que considerar **dónde** vamos a tratar un error en caso de que se produzca.

```
try {  
    // Block of code to try  
  
} catch (Exception e) {  
    // Block of code to handle errors  
}
```

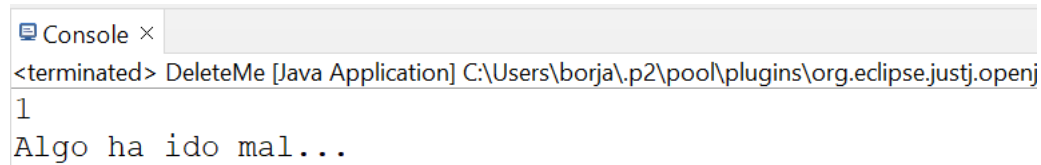
Una **excepción** en java se define en un bloque similar al anterior. Tiene dos partes:

- Un bloque en el que creemos que **podría generarse** un error
- Un bloque en el que **tratamos** el error, si ocurre.

Observa el siguiente trozo de código:

```
try {  
    int[] numeros = { 1, 2, 3 };  
    System.out.println(numeros[0]);  
    System.out.println(numeros[5]);  
    System.out.println(numeros[2]);  
    System.out.println(numeros[3]);  
} catch (Exception e) {  
    System.out.println("Algo ha ido mal...");  
}
```

Este código **genera un error** al intentar acceder a una posición del array que no existe. El resultado por consola es el siguiente.



```
Console ×
<terminated> DeleteMe [Java Application] C:\Users\borja\.p2\pool\plugins\org.eclipse.justj.openj
1
Algo ha ido mal...
```

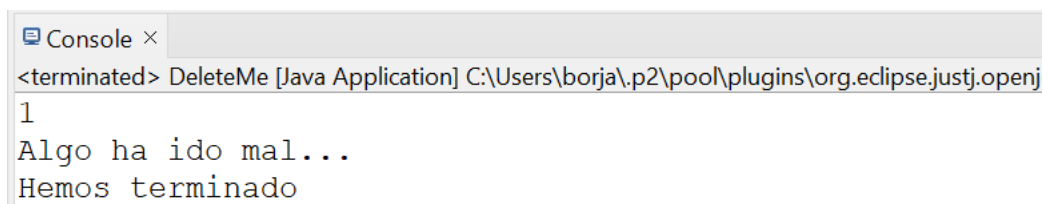
Si te fijas, el código del bloque try se ejecuta normalmente. Sin embargo, en cuanto se genera un error, la ejecución se detiene inmediatamente y **se salta** al catch. A partir del punto en el que se produce el error, el código no se ejecuta.

Si queremos que un código se ejecute siempre, independientemente de si se ha generado un error, deberemos usar la sentencia **finally**.

Como **buena práctica**, el bloque try – catch debería de ocupar siempre todo el cuerpo de un método, aunque sepamos que no todo el código del try puede generar errores.

```
public void mostrar() {
    try {
        int[] numeros = { 1, 2, 3 };
        System.out.println(numeros[0]);
        System.out.println(numeros[5]);
        System.out.println(numeros[2]);
        System.out.println(numeros[3]);
    } catch (Exception e) {
        System.out.println("Algo ha ido mal...");
    } finally {
        System.out.println("Hemos terminado");
    }
}
```

El resultado por consola de este método es:



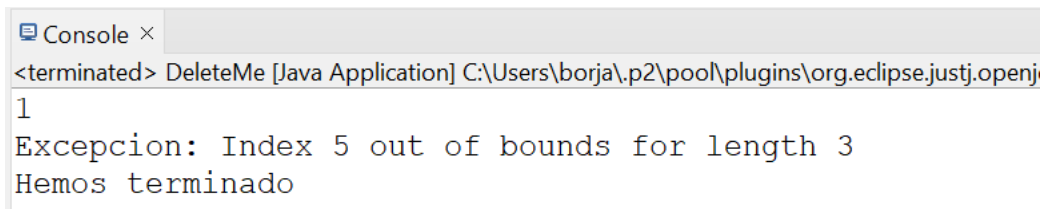
```
Console ×
<terminated> DeleteMe [Java Application] C:\Users\borja\.p2\pool\plugins\org.eclipse.justj.openj
1
Algo ha ido mal...
Hemos terminado
```

El objeto Exception

Cuando en Java se produce una excepción, **se crear un objeto** que contiene la información sobre el error producido. Las clases de estos objetos tienen como clase padre la clase **Throwable** y, por tanto, se mantiene una jerarquía en las excepciones. Podemos acceder a dicho objeto en el catch, y utilizarlo para obtener la información que necesitamos.

```
public void mostrar() {  
    try {  
        int[] numeros = { 1, 2, 3 };  
        System.out.println(numeros[0]);  
        System.out.println(numeros[5]);  
        System.out.println(numeros[2]);  
        System.out.println(numeros[3]);  
    } catch (Exception e) {  
        System.out.println("Excepcion: " + e.getMessage());  
    } finally {  
        System.out.println("Hemos terminado");  
    }  
}
```

El resultado por consola de este método es:



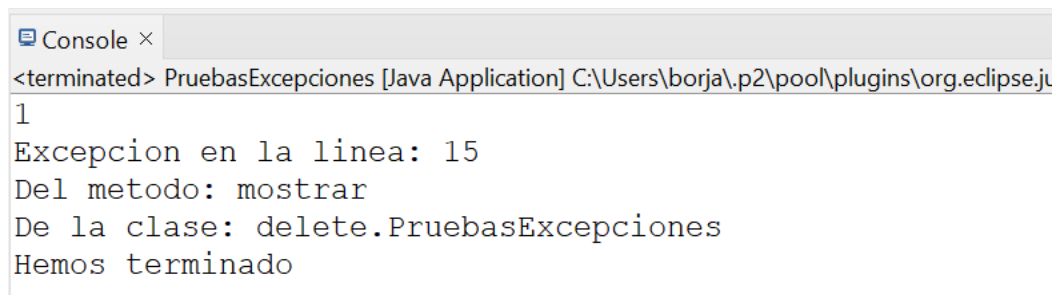
```
<terminated> DeleteMe [Java Application] C:\Users\borja\.p2\pool\plugins\org.eclipse.justj.openj  
1  
Excepcion: Index 5 out of bounds for length 3  
Hemos terminado
```

Exception dispone de diferentes métodos que aportan diferente información sobre el error. Obviamente, usaremos el más conveniente para obtener la información que deseamos mostrar. Dicho esto, **el tratamiento** de una excepción no siempre consiste en mostrar información. En ocasiones es posible arreglar el error y proseguir ejecutando el programa.

El siguiente código muestra diferentes informaciones del objeto **Exception**.

```
try {
    int[] numeros = { 1, 2, 3 };
    System.out.println(numeros[0]);
    System.out.println(numeros[5]);
    System.out.println(numeros[2]);
    System.out.println(numeros[3]);
} catch (Exception e) {
    System.out.println("Excepcion en la linea: " + e.getStackTrace()[0].getLineNumber());
    System.out.println("Del metodo: " + e.getStackTrace()[0].getMethodName());
    System.out.println("De la clase: " + e.getStackTrace()[0].getClassName());
} finally {
    System.out.println("Hemos terminado");
}
```

El resultado por consola de este método es:



```
<terminated> PruebasExcepciones [Java Application] C:\Users\borja\.p2\pool\plugins\org.eclipse.j...
1
Excepcion en la linea: 15
Del metodo: mostrar
De la clase: delete.PruebasExcepciones
Hemos terminado
```

Tipos de Excepciones

Al objeto **Exception** se le suele llamar **excepción genérica**. En realidad, dado que los errores son muy diversos, existen diferentes tipos de excepciones. Por ejemplo, **FileNotFoundException** es una excepción que se genera cuando la clase `File` intenta acceder a un fichero que no existe. **IndexOutOfBoundsException** es una excepción que se lanza cuando intentamos acceder a una posición del array que no existe.

Estas excepciones en realidad son **subclases** de la clase **Exception**. Se suelen organizar en forma de árbol, de forma que las excepciones de un nivel **agrupan** a sus descendientes en categorías. Por ejemplo, **FileNotFoundException** hereda de **IOException**, que es la excepción genérica que agrupa a las excepciones de Entrada / Salida. Existen múltiples niveles en el árbol de excepciones, existiendo la posibilidad de añadir nuevos nodos y hojas al mismo según nuestras necesidades.

Múltiples excepciones

Cuando programamos una función, nos podemos encontrar con la necesidad de atender a **varias excepciones diferentes**. Existen varias formas de hacer esto, cada una con sus ventajas e inconvenientes. Depende de nosotros escoger la más conveniente.

Capturar todo con Exception

La solución más simple consiste en capturarlo todo como si fuese una Exception. En este caso, no importa qué excepción se genere: todas son capturadas por el mismo catch. Lamentablemente, esto nos impide hacer tratamientos individualizados de los errores y sus posibles soluciones.

```
public void mostrarFichero() {
    try {
        MyFile myFile = new MyFile ();
        String text = myFile.readFile ();
        System.out.println("Contenido del fichero: " + text );
    } catch (Exception e) {
        System.out.println("Excepcion: " + e.getMessage());
    } finally {
        System.out.println("Hemos terminado");
    }
}
```

Capturar cada excepción por separado

La solución estándar consiste en añadir un catch por cada excepción, de la siguiente forma:

```
public void mostrarFichero() {
    try {
        MyFile myFile = new MyFile ();
        String text = myFile.readFile ();
        System.out.println("Contenido del fichero: " + text );
    } catch (FileNotFoundException e) {
        System.out.println("FileNotFoundException: " + e.getMessage());
    } catch (IOException e) {
        System.out.println("IOException: " + e.getMessage());
    } catch (Exception e) {
        System.out.println("Excepcion: " + e.getMessage());
    } finally {
        System.out.println("Hemos terminado");
    }
}
```

Esto permite hacer un tratamiento específico para cada excepción, a costa de tener que escribir mucho más código. **El orden** en el que se capturan las excepciones **es importante**. Deben de colocarse de las más específicas a las más generales. Dado que **FileNotFoundException** pertenece a **IOException**, se coloca encima; y **Exception** se pone la de más abajo (es la superclase de todas las demás).

Esto es así porque cuando se lanza una excepción, Java recorre los catch de arriba abajo. Supongamos que se genera una **IOException**. Se hace lo siguiente:

- ¿**IOException** es **FileNotFoundException** o una subclase suya? No, luego pasa a comprobar la siguiente excepción.
- ¿**IOException** es **IOException** o una subclase suya? Si, luego esta es la rama que tiene que ejecutar.

Fíjate que **se pregunta** por la clase o por las **subclases**. Esto es deliberado, y el motivo de por qué capturar **Exception** te permite tratar todas las excepciones posibles; porque todas las excepciones son subclases de **Exception**.

Si el tratamiento de varias excepciones va a ser similar, pero no el de todas, es posible agruparlas de la siguiente forma:

```
public void mostrarFichero() {
    try {
        MyFile myFile = new MyFile ();
        String text = myFile.readFile ();
        System.out.println("Contenido del fichero: " + text );
    } catch (SocketException | FileNotFoundException e) {
        System.out.println("Exception: " + e.getMessage());
    } catch (IOException e) {
        System.out.println("IOException: " + e.getMessage());
    } catch (Exception e) {
        System.out.println("Excepcion: " + e.getMessage());
    } finally {
        System.out.println("Hemos terminado");
    }
}
```

Como convenio, es recomendable capturar **Exception siempre**, para evitar que una excepción no controlada que desconocíamos se propague y detenga la ejecución del programa.

Propagar excepciones

En Java también existe la posibilidad de dejar que **las excepciones generadas** por un trozo de código **sean tratadas en otro** trozo de código. A esto se le llama generalmente **propagar la excepción**, y se usa habitualmente para centralizar el tratamiento de errores.

En el siguiente ejemplo, `mostrarArray ()` genera una excepción, pero en lugar de ser tratada en esa misma función, la excepción se propaga a `controlarErrores ()`. En esta, el proceso se repite: se genera la misma excepción, pero como hay un try-catch, se captura normalmente.

```
public void controlarErrores() {
    try {
        mostrarArray();
    } catch (Exception e) {
        System.out.println("Exception: " + e.getMessage());
    } finally {
        System.out.println("Hemos terminado");
    }
}

public void mostrarArray() {
    int[] numeros = { 1, 2, 3 };
    System.out.println(numeros[0]);
    System.out.println(numeros[5]);
    System.out.println(numeros[2]);
    System.out.println(numeros[3]);
}
```

El resultado de la ejecución de este código es:

```
<terminated> PruebasExcepciones [Java Application] C:\Users\borja\.p2\pool\plugins\org.eclipse.ji
1
Exception: Index 5 out of bounds for length 3
Hemos terminado
```

Es posible propagar una excepción hasta donde queramos; no obstante, al final siempre debería de haber un bloque try-catch que capture todas las excepciones. Si una excepción no se captura, entonces llegará hasta el `main ()`, y de allí hasta la Máquina Virtual de Java, provocando un **volcado de error** por la consola. Esto nunca debería ocurrir.

```
<terminated> PruebasExcepciones [Java Application] C:\Users\borja\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_20.0.1.v20230425-1611\jre\bin\javaw.exe [2
1
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3
    at MyProjectsForProgram/delete.PruebasExcepciones.mostrarArray(PruebasExcepciones.java:60)
    at MyProjectsForProgram/delete.PruebasExcepciones.main(PruebasExcepciones.java:12)
```

Propagando excepciones: throws

En el ejemplo anterior, estamos haciendo una propagación de excepciones genéricas, es decir, que sin que importe la excepción generada en **mostrarArray ()** ésta se va a capturar y tratar de forma general en **controlarErrores ()**. Pero esto no es correcto siempre.

La palabra reservada **throws** añadida a un método indica qué excepciones pueden ser **lanzadas y propagadas** por un método. Por defecto, todos los métodos pueden lanzar **Exception**; pero no hace falta escribirlo. Para todas las demás, **hace falta** especificarlo.

En este ejemplo, indicamos con el **throws** que **mostrarArray ()** genera específicamente la excepción **ArrayIndexOutOfBoundsException**. Esto nos permite que en **controlarErrores ()** podamos hacer un tratamiento específico del error.

```
public void controlarErrores() {
    try {
        mostrarArray();
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("ArrayIndexOutOfBoundsException: " + e.getMessage());
    } catch (Exception e) {
        System.out.println("Exception: " + e.getMessage());
    } finally {
        System.out.println("Hemos terminado");
    }
}

public void mostrarArray() throws ArrayIndexOutOfBoundsException{
    int[] numeros = { 1, 2, 3 };
    System.out.println(numeros[0]);
    System.out.println(numeros[5]);
    System.out.println(numeros[2]);
    System.out.println(numeros[3]);
}
```

El resultado de la ejecución es:

```
<terminated> PruebasExcepciones [Java Application] C:\Users\borja\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.j
1
ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3
Hemos terminado
```

Nótese que **Exception** siempre se propaga, aunque no se indique. Tampoco hay límite en cuanto al número de excepciones que propaguemos; pero normalmente en estos casos se agrupan en un solo tipo. Por ejemplo, si se tienen que propagar tres excepciones lo que se hace es propagar en su lugar la excepción ‘padre’ de todas ellas.

Tratamiento obligado

Añadir un **throws** a un método **obliga** a que todos los métodos que llamen a ese método tengan que, **o bien tratar** esa excepción, **o bien propagarla** a otra parte. No puedes ignorar ni debe ignorarse un throws distinto de Exception.

Como puedes ver, Java te obliga a tratar la excepción que podría lanzarte **abrirFichero ()**.

```
public void mostrarFichero() {  
    abrirFichero();  
}  
  
private void abrirFichero() throws IOException {  
  
}
```

Lanzar excepción

Es posible **lanzar** una excepción **manualmente** como forma de indicar que se ha producido un error. Esto suele hacerse para cortar la ejecución de un código y no utilizar el return del método para enviar el error. Para lanzar una excepción de esta forma, se usa **throw**.

```
public void mostrarFichero() {  
    try {  
        abrirFichero("file.dat");  
    } catch (Exception e) {  
        System.out.println("Exception: " + e.getMessage());  
    }  
}  
  
private void abrirFichero(String name) throws IOException {  
    if (notExist (name)) {  
        throw new IOException();  
    }  
}
```

Al generar una nueva excepción mediante throws, es posible añadir información adicional utilizando los métodos que dispone la propia clase.

```
private void abrirFichero(String name) throws IOException {  
    if (notExist (name)) {  
        throw new IOException("Fichero no encontrado");  
    }  
}
```

Crear nuevas excepciones

Cuando intentamos entender qué ha ido mal en mi programa, la información no la obtenemos por el mensaje de error que escribimos por consola únicamente. Eso es algo totalmente secundario y pensado para el **usuario** del programa, no para el programador. Para nosotros, lo importante es la **excepción misma**. No es lo mismo una **IOException** que una **DateTimeException**, dado que cada cuál se genera bajo unas condiciones diferentes. El propio nombre de la clase de hecho nos da información sobre el error producido.

Por tanto, es posible que nosotros generemos **nuestras propias excepciones**.

```
public class MyException extends Exception {  
  
    private static final long serialVersionUID = -6784516171993559200L;  
  
    public MyException () {  
        super();  
    }  
  
    public MyException (String message) {  
        super(message);  
    }  
}
```

Este ejemplo crea una excepción **MyException** dentro de mi proyecto. Extiende de la clase **Exception**, lo que significa que hereda sus métodos (y su significado). Obviamente, podemos hacer que nuestras excepciones hereden de cualquier otra clase que queramos. A partir de aquí, podemos lanzar esta excepción cuando queramos mediante **throw**.

Un uso habitual de este tipo de prácticas es cuando **enmascaras** una excepción de un método de acceso a Base de Datos por otra que sea comprensible para otra parte del código. Por ejemplo, podríamos considerar que, ocurra lo que ocurra con el método **buscarAlumno ()** lo que se va a enviar es una excepción propia llamada **UserNotFoundException**, que sí tiene sentido para el Controlador (MVC).

```
private void buscarAlumno () throws UserNotFoundException {  
    try {  
  
        // Acceso a BBDD...  
  
    } catch (Exception e) {  
        // Si hay un error, suponemos que usuario no encontrado  
        throw new UserNotFoundException(e.getMessage());  
    }  
}
```

Malas prácticas

Cosas que **no** hay que hacer con excepciones

Try-catch no ocupa toda la función

Meter bloques de código encima y debajo del try-catch es jugar con fuego. Date cuenta de que, si hay excepción y pasamos por el catch, el segundo bloque de código se va a ejecutar siempre, y puede que esto nos despiste. Es más prudente que el try-catch ocupe toda la función por completo.

```
public void malaIdea () {  
  
    // Block of code  
  
    try {  
  
        // Block of code  
  
    } catch (Exception e) {  
        // Some code to handle error  
    }  
  
    // Block of code  
  
}
```

Capturar una excepción y lanzarla

Si le capturas una excepción con un catch, y lo único que haces es lanzar una nueva o propagarla con un throw... ¿para qué la capturas? Y encima, si haces una excepción nueva, pierdes la información de la excepción original...

```
public void controlarErrores() throws ArrayIndexOutOfBoundsException, Exception {  
    try {  
        mostrarArray();  
    } catch (ArrayIndexOutOfBoundsException e) {  
        throw new ArrayIndexOutOfBoundsException ();  
    } catch (Exception e) {  
        throw new Exception ();  
    } finally {  
        System.out.println("Hemos terminado");  
    }  
}
```