

MP0485
Programación
UF8. Excepciones

8.1. Captura de excepciones

Índice

☰	Objetivos	3
☰	Uso de objetos de excepción	4
☰	Provocando excepciones no comprobadas	13
☰	Varios catch	16
☰	Polimorfismo y excepciones	21
☰	Relanzamiento o propagación de excepciones	23
☰	Resumen de las excepciones más habituales	28
☰	Resumen	30

Objetivos

Con esta unidad perseguimos los siguientes objetivos:

1

Desarrollar programas Java que controlen situaciones de excepción.

2

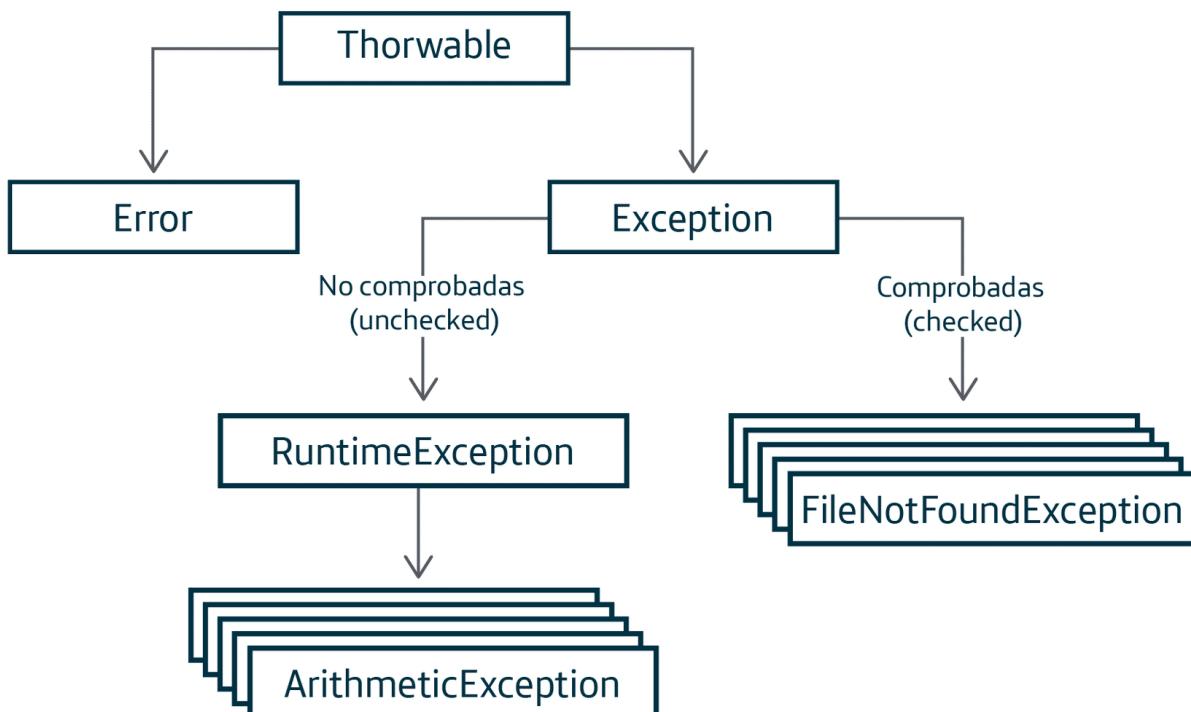
Relacionar los conceptos de polimorfismo y control de excepciones en la programación Java.

¡Ánimo y adelante!

Uso de objetos de excepción

Cuando se produce una condición de error (excepción para Java), la máquina virtual genera un tipo de objeto especial con información sobre el suceso ocurrido; un objeto de tipo *Exception*.

La jerarquía de clases para la gestión de excepciones en Java se puede resumir con la siguiente imagen:



Jerarquía de clases de excepción en Java.

La clase principal de la que heredan todas las clases que representan excepciones es **Throwable**.

Observando la estructura jerárquica de la imagen, puedes comprobar que *Throwable* se descompone en dos ramas, que representan dos clases de situaciones de error:

1

Error: representan situaciones que se escapan al control del programador, por lo que no deberíamos hacer nada con ellas. Por ejemplo, un error grave en el funcionamiento de la máquina virtual de Java que escapa a nuestro control.

2

Exception: representan situaciones que el programador sí puede gestionar y por lo tanto, será de este tipo de clases de las que nos ocupemos en esta lección. Los objetos de este tipo son los que realmente denominamos excepciones.

Prueba a ejecutar este pequeño programa, donde realizamos una operación claramente incorrecta; una división cuyo divisor es un cero:

```
public class Principal {  
    public static void main(String args[]) {  
        int cero=0;  
        int resul=6/cero;  
        System.out.println(resul);  
    }  
}
```

Programa que genera una situación de error o excepción.

Como resultado has obtenido un mensaje de error o excepción como el siguiente:

 Exception in thread "main" java.lang.ArithmetiException: / by zero
at DivCero.main(DivCero.java:4)

La máquina virtual de Java (JVM) ha generado un objeto de la clase *ArithmetiException* y ha abortado el programa. A lo largo de la unidad aprenderás cómo puedes capturar dicho objeto para recuperar el control sobre la ejecución del programa y evitar que aborte la ejecución.

Tipos de excepciones (clases derivadas de *Exception*)

Volviendo de nuevo a observar la imagen, puedes comprobar que existen dos tipos de clases que derivan de *Exception*:

1

Excepciones no comprobadas que derivan de *RuntimeException*: no estamos obligados a controlar estas excepciones. Tienen que ver con situaciones en que existe un mal planteamiento en el código. Puede evitarse que lleguen a ocurrir y no hay necesidad de tener que controlarlas, aunque podemos hacerlo. Nuestro ejemplo anterior de la división entre cero ha provocado una *exception* de tipo *ArithmetiException*, este es un ejemplo de excepción que no es obligatorio controlar. También está claro que podría evitarse esta situación antes de realizar la división.

2

Excepciones comprobadas que derivan directamente de *Exception*: son situaciones de error de las que podemos recuperarnos, pero que no son fruto de un mal planteamiento del código, sino de una situación inesperada, tal como cuando abrimos un fichero de texto y de repente alguien ha borrado dicho fichero del disco. Debemos controlar este tipo de excepciones para que el programa pueda recuperarse de la situación de error sin necesidad de abortarlo.

¿Y cómo se controlan las excepciones?

Un bloque de código susceptible de producir una excepción debe encerrarse en un bloque *try* {...} seguido de un bloque *catch* {...}, que capturará el objeto de excepción. Debes utilizar el siguiente formato:

```
try {  
    // Sentencias que pueden provocar excepción  
}  
catch(Exception e){  
    // Respuesta a la situación de excepción  
}  
finally {  
    // Sentencias que se ejecutan incondicionalmente  
}
```

Formato *try ... catch ... finally*.

- **Bloque *try*:** donde se encierran las instrucciones que pueden provocar una situación de excepción, ya sea comprobada o no comprobada. La diferencia está en que si se trata de excepciones comprobadas, es obligatorio usar un bloque *try* y si se trata de excepciones no comprobadas, el uso de *try* es opcional.
- **Bloque *catch*:** donde se captura el objeto de excepción. Si dentro del bloque *try* ocurre una situación de excepción, el control pasa al siguiente bloque *catch*, capaz de recoger dicha excepción.
- **Bloque *finally*:** este bloque es opcional y contiene sentencias que se ejecutarán de manera incondicional ocurra o no una excepción.

```
public class Principal {  
    public static void main(String args[]) {  
        int cero=0;  
        int resul;  
        try {  
            resul=6/cero;  
            System.out.println(resul);  
        }  
        catch (ArithmetricException e) {  
            System.out.println("Se ha producido una excepción")  
            System.out.println(e.getMessage());  
        }  
        finally {  
            System.out.println("Hasta pronto");  
        }  
    }  
}
```

Vamos ver paso a paso lo que ocurre con el programa anterior:

1

La sentencia `resul=6/cero;` provoca una excepción, por lo que la máquina virtual de Java genera un objeto de tipo *ArithmetricException*. Al encontrarse dentro de un bloque *try*, el control pasará al primer bloque *catch* que pueda recoger un objeto de tipo *ArithmetricException* (pueden existir varios bloques *catch*). En el ejemplo, el objeto es recogido en la variable *e*.

2

Dentro del bloque *catch*, el programa informa de lo ocurrido y utiliza el método *getMessage()* de la clase *Exception* para mostrar la descripción del error o excepción. Recuerda que *ArithmetricException* hereda de *Exception*, luego el objeto *e* es tanto una *ArithmetricException* como una *Exception*.

3

La sentencia `System.out.println(resul);` será pasada por alto, no llegará a ejecutarse, ya que tras ocurrir la excepción en la línea anterior, el control pasó al bloque *catch*.

4

Por último se ejecutará el bloque *finally*.

¿Y para qué sirve el bloque *finally*? Vas a descubrirlo con una pequeña práctica.

Pega este pequeño programa en un proyecto Eclipse y ejecútalo:

```
import java.util.Scanner;

public class Principal {
    public static void main(String[] args) {
        Scanner lector = new Scanner(System.in);
        System.out.println("Introduce radio de la circunferencia: ");
        String num = lector.nextLine();
        lector.close();
        int radio;
        try {
            radio = Integer.parseInt(num);
        } catch (NumberFormatException e) {
            System.out.println("Ha ocurrido una excepción de tipo: " + e);
            System.out.println(e.getMessage());
            return;
        }
        System.out.println("Longitud: " + (2*Math.PI*radio));
        System.out.println("Área: " + (Math.PI*radio*radio));
        System.out.println("Fin del programa");
    }
}
```

Introducimos por teclado el radio de una circunferencia como un texto, dentro de un bloque *try* lo convertimos a valor numérico. Al final del programa estamos calculando y mostrando en pantalla la longitud y el área de la circunferencia, pero como no tiene sentido que estas sentencias se ejecuten si hay error en la entrada de datos, hemos colocado un *return* al final del bloque *catch*, lo que provoca que termine la ejecución del programa si se desencadenó la excepción y no ejecuta el resto del código que queda en la función. Sin embargo, nos gustaría que pase lo que pase aparezca el mensaje de "*Fin del programa*", y es aquí donde viene a salvarnos el bloque *finally*.

```
import java.util.Scanner;

public class Principal {
    public static void main(String[] args) {
        Scanner lector = new Scanner(System.in);
        System.out.println("Introduce radio de la circunferencia: ");
        String num = lector.nextLine();
        lector.close();
        int radio;
        try {
            radio = Integer.parseInt(num);
        } catch (NumberFormatException e) {
            System.out.println("Ha ocurrido una excepción de tipo: " + e);
            System.out.println(e.getMessage());
            return;
        } finally {
            System.out.println("Fin del programa");
        }
        System.out.println("Longitud: " + (2*Math.PI*radio));
        System.out.println("Área: " + (Math.PI*radio*radio));
    }
}
```

El bloque *finally* se ejecuta siempre, independientemente del código que esté encerrado en cada uno de los *catch*.

¿Y cómo se trabaja con las excepciones comprobadas?

Las excepciones comprobadas son las que pueden producir determinados métodos, cuyas llamadas deben estar obligatoriamente encerradas dentro de un bloque *try* o, en su defecto, estas excepciones también pueden ser relanzadas para que las gestione el método superior dentro de la cadena de llamadas.

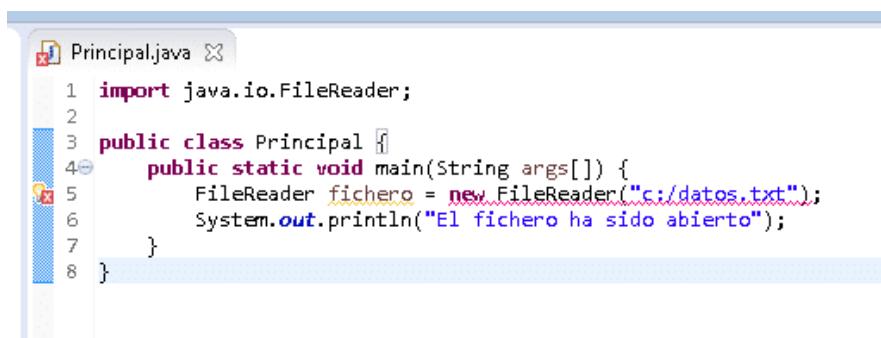
En nuestro ejemplo de la división entre 0 no estábamos obligados a poner un bloque *try*. El programa no tenía errores de compilación, aunque al ejecutar, nos lanzaba la excepción, pero hay determinadas situaciones en que estamos obligados a gestionar las situaciones de error. Vamos a ver un pequeño ejemplo para que puedas comprenderlo mejor.

Prueba a crear este pequeño programa dentro de Eclipse:

```
import java.io.FileReader;

public class Principal {
    public static void main(String args[]) {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}
```

Verás que ni siquiera has intentado ejecutar y ya estás teniendo problemas, Eclipse te está indicando que tienes errores de compilación, tu programa no puede ni siquiera ser ejecutado.



El error de compilación viene porque el método constructor de *FileReader* abre un fichero para lectura cuya ruta y nombre se especifica como argumento. El fichero especificado podría no existir, y eso es algo que escapa a nuestro control, porque no podemos predecir si el usuario va a borrar el archivo o lo va a mover de sitio. Por esta razón, estamos obligados a controlar dicha excepción.

Para poder ejecutar el programa tienes que usar obligatoriamente un bloque *try*. Cambia ahora el código de esta manera:

```
import java.io.FileNotFoundException;
import java.io.FileReader;

public class Principal {
    public static void main(String args[]) {
        try {
            FileReader fichero = new FileReader("c:/datos.txt")
            System.out.println("El fichero ha sido abierto");
        } catch (FileNotFoundException e) {
            System.out.println("Error al abrir el fichero");
            System.out.println(e.getMessage());
        }
    }
}
```

En resumen:

1

Excepciones comprobadas: *try ... catch* obligatorio.

2

Excepciones no comprobadas: *try ... catch* opcional.

Provocando excepciones no comprobadas

En este apartado tendrás oportunidad de provocar excepciones de tipo "no comprobadas". Como no es obligatorio usar `try ... catch` en este tipo de excepciones, no lo usaremos, solo vamos a familiarizarnos con más tipos de excepciones, a parte de la ya conocida `ArithmeticException`.

Ve probando con Eclipse los pequeños programas que te vamos a proponer. Ejecuta cada ejemplo para que veas con tus propios ojos lo que ocurre.

Salida de los límites de un `array`

```
public class Principal {  
    public static void main(String args[]) {  
        int nums[] = new int[3];  
        nums[4]=25;  
    }  
}
```

Estamos accediendo a una posición inexistente del `array` y se produce la excepción `ArrayIndexOutOfBoundsException`.

Salida de los límites de una colección

```
import java.util.ArrayList;

public class Principal {
    public static void main(String args[]) {
        ArrayList<Integer> nums = new ArrayList<Integer>();
        nums.add(25);
        nums.add(56);
        nums.add(18);
        System.out.println(nums.get(4));
    }
}
```

Estamos intentando recuperar un elemento de la colección de una posición inexistente. Se produce la excepción *IndexOutOfBoundsException*.

Error en la conversión entre tipos de datos

```
public class Principal {
    public static void main(String args[]) {
        String texto = "pepe";
        int num = Integer.parseInt(texto);
    }
}
```

Estamos intentando convertir el valor de tipo *String* "pepe" a formato numérico, cosa que no es posible y provoca una excepción de tipo *NumberFormatException*.

Intento de utilización de una referencia con valor *null*

```
public class Principal {  
    public static void main(String args[]) {  
        String texto=null;  
        System.out.println(texto.toString());  
    }  
}
```

Estamos intentando utilizar una referencia a un *String* que en realidad no apunta a ningún objeto, tiene el valor *null*. Se produce una excepción de tipo *NullPointerException*.

Recuerda que aunque no tienes obligación de controlar estas excepciones con un *try ... catch*, aunque puedes hacerlo si lo consideras necesario.

Varios catch

En esta ocasión probaremos con un pequeño programa que puede producir dos tipos distintos de excepciones.

Este pequeño programa permite al usuario introducir por teclado los valores de un dividendo y un divisor, almacenándolos en variables tipo *String*. Luego, convierte ambos valores a datos numéricos de tipo *int* y por último calcula la división y el resto, mostrándolos en pantalla.

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        Scanner lector = new Scanner(System.in);
        System.out.println("Introduce dividendo: ");
        String texto = lector.nextLine();
        int dividendo = Integer.parseInt(texto);
        System.out.println("Introduce divisor: ");
        texto = lector.nextLine();
        int divisor = Integer.parseInt(texto);
        int resultado = dividendo/divisor;
        int resto = dividendo%divisor;
        System.out.println("Resultado división: " + resultado);
        System.out.println("Resto: " + resto);
        lector.close();
    }
}
```

Esto puede dar lugar a tres situaciones diferentes:

1

Que el usuario introduzca correctamente los dos valores y el programa funcione sin excepciones.

```
<terminated> Principal (16) [Java App]
Introduce dividendo:
16
Introduce divisor:
3
Resultado división: 5
Resto: 1
```

2

Que el usuario introduzca un valor de texto que no pueda ser convertido a número.

```
Introduce dividendo:
16
Introduce divisor:
pepe
Exception in thread "main" java.lang.NumberFormatException: For input string: "pepe"
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
at java.lang.Integer.parseInt(Integer.java:580)
at java.lang.Integer.parseInt(Integer.java:615)
at Principal.main(Principal.java:11)
```

3

Que el usuario introduzca 0 como divisor.

```
Introduce dividendo:
16
Introduce divisor:
0
Exception in thread "main" java.lang.ArithmetricException: / by zero
at Principal.main(Principal.java:12)
```

¿Cómo podemos tener controladas todas estas situaciones?

Tras un bloque *try* pueden existir tantos bloques *catch* como sean necesarios:

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        try {
            Scanner lector = new Scanner(System.in);
            System.out.println("Introduce dividendo: ");
            String texto = lector.nextLine();
            int dividendo = Integer.parseInt(texto);
            System.out.println("Introduce divisor: ");
            texto = lector.nextLine();
            int divisor = Integer.parseInt(texto);
            int resultado = dividendo/divisor;
            int resto = dividendo%divisor;
            System.out.println("Resultado división: " + resultado);
            System.out.println("Resto: " + resto);
            lector.close();
        }
        catch (ArithmaticException e1) {
            System.out.println("Se ha producido una ArithmaticException");
            System.out.println(e1.getMessage());
        }
        catch (NumberFormatException e2) {
            System.out.println("Se ha producido un NumberFormatException");
            System.out.println(e2.getMessage());
        }

        System.out.println("El programa sigue aquí, no se ha abortado");
    }
}
```

El objeto de excepción será recogido por el bloque *catch* cuyo tipo de argumento coincide con el tipo de excepción que se ha producido.

Cuando colocamos varios *catch* es importante tener en cuenta el orden en que se colocan. Deben ir situados en orden de las clases más específicas a las más genéricas, o lo que es lo mismo, desde las inferiores en el árbol jerárquico a las superiores.

Observa este código:

```
try {  
    FileReader f = new FileReader("C:/datos.txt");  
    int x = f.read();  
    System.out.println(x);  
    f.close();  
} catch (FileNotFoundException e1) {  
    System.out.println("El fichero no se encuentra");  
} catch (IOException e2) {  
    System.out.println(e2.getMessage());  
} catch (Exception e3) {  
    System.out.println(e3.getClass().getName());  
    System.out.println(e3.getMessage());  
}
```

Dentro del bloque *try* intentamos abrir un fichero para lectura y después el programa realiza alguna operación de lectura. No te preocupes por comprender el código encerrado en el *try*, lo único que nos ocupa ahora es la comprensión de las excepciones. Si el fichero no puede abrirse por cualquier circunstancia, irá recorriendo secuencialmente los bloques *catch* en busca de la primera referencia que pueda capturar la excepción que se ha producido. Si el primer bloque *catch* fuese el de *Exception*, hubiera capturado la excepción sea cual sea, ya que un objeto *FileNotFoundException* o *IOException* también es *Exception*.

Una de las novedades a partir de la versión 8 de Java son los llamados *multicatch*, capaces de recoger con un solo bloque *catch* distintos tipos de excepciones.

Para separar cada uno de los tipos de excepción se utiliza el carácter |.

El programa anterior utilizando *multicatch* quedaría así:

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        try {
            Scanner lector = new Scanner(System.in);
            System.out.println("Introduce dividendo: ");
            String texto = lector.nextLine();
            int dividendo = Integer.parseInt(texto);
            System.out.println("Introduce divisor: ");
            texto = lector.nextLine();
            int divisor = Integer.parseInt(texto);
            int resultado = dividendo/divisor;
            int resto = dividendo%divisor;
            System.out.println("Resultado división: " + resultado);
            System.out.println("Resto: " + resto);
            lector.close();
        }
        catch (NumberFormatException | ArithmeticException e) {
            System.out.println("Se ha producido una excepción de tipo: " + e.getClass().getName());
            System.out.println(e.getMessage());
        }
        System.out.println("El programa sigue aquí, no se ha abortado");
    }
}
```

Utilizamos *e.getClass().getName()* para averiguar cuál de las dos excepciones ocurrió.

Polimorfismo y excepciones

Recuerda que todas las clases que representan excepciones heredan directa o indirectamente de la clase *Exception*.

Esto significa que los objetos de tipo *NumberFormatException*, *ArithmetricException*, *IndexOutOfBoundsException*, *FileNotFoundException*, etc., son a la vez objetos *Exception*. Esto nos permite aplicar las ventajas del polimorfismo, es decir, tratar todos estos objetos de manera genérica como *excepciones*.

Sea cual sea el tipo de excepción que produzca nuestro programa, podrá ser capturada de esta manera:

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        try {
            Scanner lector = new Scanner(System.in);
            System.out.println("Introduce dividendo: ");
            String texto = lector.nextLine();
            int dividendo = Integer.parseInt(texto);
            System.out.println("Introduce divisor: ");
            texto = lector.nextLine();
            int divisor = Integer.parseInt(texto);
            int resultado = dividendo/divisor;
            int resto = dividendo%divisor;
            System.out.println("Resultado división: " + resultado);
            System.out.println("Resto: " + resto);
            lector.close();
        }
        catch (Exception e) {
            System.out.println("Se ha producido una Excepción d
                System.out.println(e.getMessage());
        }
    }

    System.out.println("El programa sigue aquí, no se ha aborta
}
}
```

Una referencia de tipo *Exception* puede apuntar a cualquier tipo de objeto de excepción. Utilizamos *g.getClass().getName()* para averiguar qué excepción ocurrió.

Aunque este sistema resulta muy cómodo, no hay que abusar de su uso, ya que evita el hecho de poder tratar cada tipo de excepción de manera más especializada y profesional.

Relanzamiento o propagación de excepciones

Aunque estamos obligados a controlar las excepciones de tipo "comprobadas", también podemos propagarlas o relanzarlas hacia arriba para que sean controladas en el método superior en la pila de llamadas.

Observa este ejemplo y prueba a añadirlo a un proyecto Eclipse:

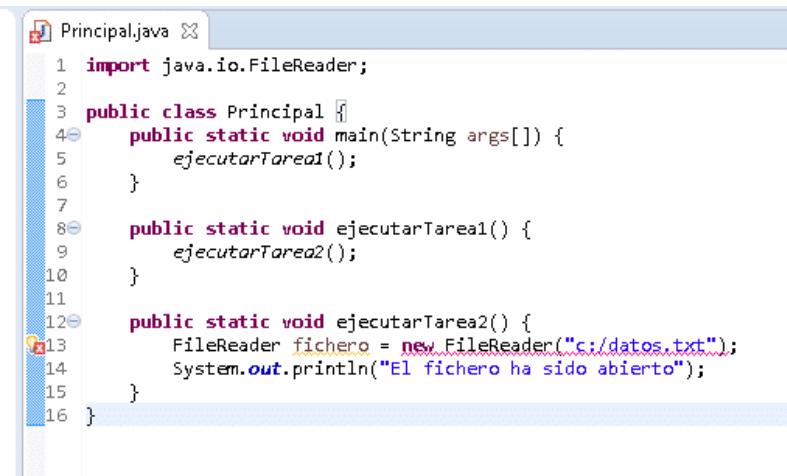
```
import java.io.FileReader;

public class Principal {
    public static void main(String args[]) {
        ejecutarTarea1();
    }

    public static void ejecutarTarea1() {
        ejecutarTarea2();
    }

    public static void ejecutarTarea2() {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}
```

El método `ejecutarTarea2()` es invocado desde el método `ejecutarTarea1()`, que es llamado desde el método `main()`. Si ya has pegado el código en Eclipse estarás comprobando que de nuevo te da error de compilación porque es obligatorio controlar la posible excepción de tipo `FileNotFoundException` que puede provocar el constructor de `FileReader`. Deberíamos añadir el bloque `try ... catch` en el método `ejecutarTarea2()`.



```
Principal.java
1 import java.io.FileReader;
2
3 public class Principal {
4     public static void main(String args[]) {
5         ejecutarTarea1();
6     }
7
8     public static void ejecutarTarea1() {
9         ejecutarTarea2();
10    }
11
12    public static void ejecutarTarea2() {
13        FileReader fichero = new FileReader("c:/datos.txt");
14        System.out.println("El fichero ha sido abierto");
15    }
16 }
```

Sin embargo, también es posible que el método `ejecutarTarea2()` propague o relance la excepción hacia arriba para responsabilizar al método superior. Esto se hace con una declaración `throws` en la cabecera del método, indicando las excepciones que vayan a relanzarse.

```
import java.io.FileNotFoundException;
import java.io.FileReader;

public class Principal {
    public static void main(String args[]) {
        ejecutarTarea1();
    }

    public static void ejecutarTarea1() {
        ejecutarTarea2();
    }

    public static void ejecutarTarea2() throws FileNotFoundException {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}
```

Propagación de la excepción desde `ejecutarTarea2()` hasta `ejecutarTarea1()`.

En este caso le hemos pasado la responsabilidad al método `ejecutarTarea1()`, que es el que ahora tiene el problema, tal y como puedes apreciar en la siguiente imagen:

```
1@ import java.io.FileNotFoundException;
2 import java.io.FileReader;
3
4 public class Principal {
5@     public static void main(String args[]) {
6         ejecutarTarea1();
7     }
8
9@     public static void ejecutarTarea1() {
10        ejecutarTarea2();
11    }
12
13@    public static void ejecutarTarea2() throws FileNotFoundException {
14        FileReader fichero = new FileReader("c:/datos.txt");
15        System.out.println("El fichero ha sido abierto");
16    }
17 }
```

Por otro lado, el método *ejecutarTarea1()* también podría "escurrir el bulto" pasando el problema al método *main()* de la misma forma:

```
import java.io.FileNotFoundException;
import java.io.FileReader;

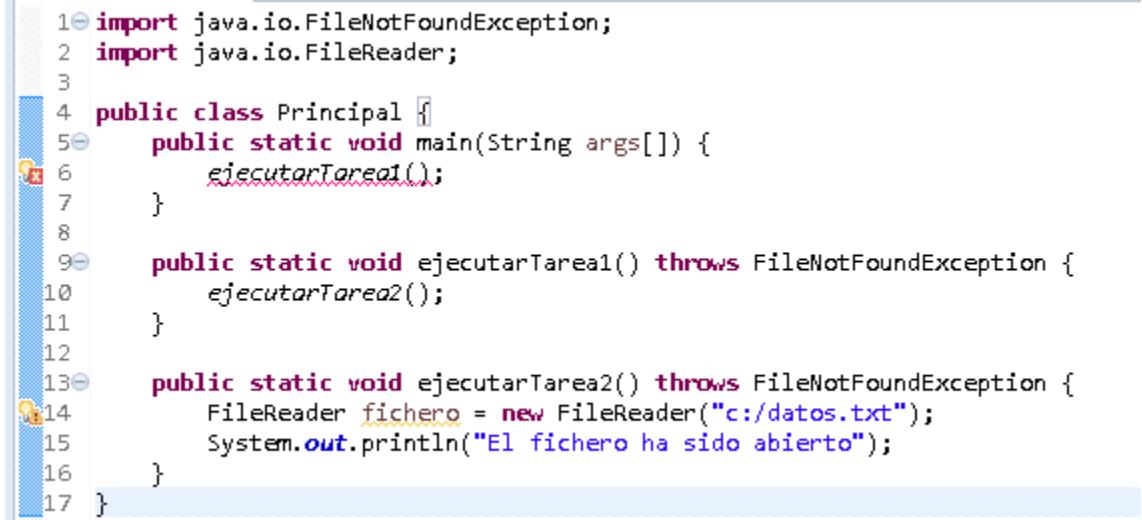
public class Principal {
    public static void main(String args[]) {
        ejecutarTarea1();
    }

    public static void ejecutarTarea1() throws FileNotFoundException {
        ejecutarTarea2();
    }

    public static void ejecutarTarea2() throws FileNotFoundException {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}
```

Propagación de la excepción desde *ejecutarTarea1()* hasta *main()*.

Ahora es el método `main()` el que tiene el problema.



```
1 import java.io.FileNotFoundException;
2 import java.io.FileReader;
3
4 public class Principal {
5     public static void main(String args[]) {
6         ejecutarTarea1();
7     }
8
9     public static void ejecutarTarea1() throws FileNotFoundException {
10        ejecutarTarea2();
11    }
12
13    public static void ejecutarTarea2() throws FileNotFoundException {
14        FileReader fichero = new FileReader("c:/datos.txt");
15        System.out.println("El fichero ha sido abierto");
16    }
17 }
```

The screenshot shows a Java code editor with the following code. Lines 13 and 14 are highlighted with orange markers. Line 13 has a red 'X' icon indicating an error. Line 14 has an orange exclamation mark icon. The code is as follows:

En el método `main()` podemos quitar los problemas de compilación de dos formas:

1

De nuevo propagando la excepción hacia arriba, pero como ya no hay más métodos para recogerla, será recibida por la máquina virtual de Java, que abortará el programa en el caso de que realmente se produzca la excepción, es decir, en el caso de que no exista el fichero que estamos intentando abrir.

```
import java.io.FileNotFoundException;
import java.io.FileReader;

public class Principal {
    public static void main(String args[]) throws FileNotFoundException
        ejecutarTarea1();

    public static void ejecutarTarea1() throws FileNotFoundException {
        ejecutarTarea2();
    }

    public static void ejecutarTarea2() throws FileNotFoundException {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}
```

2

Colocar un *try ... catch*:

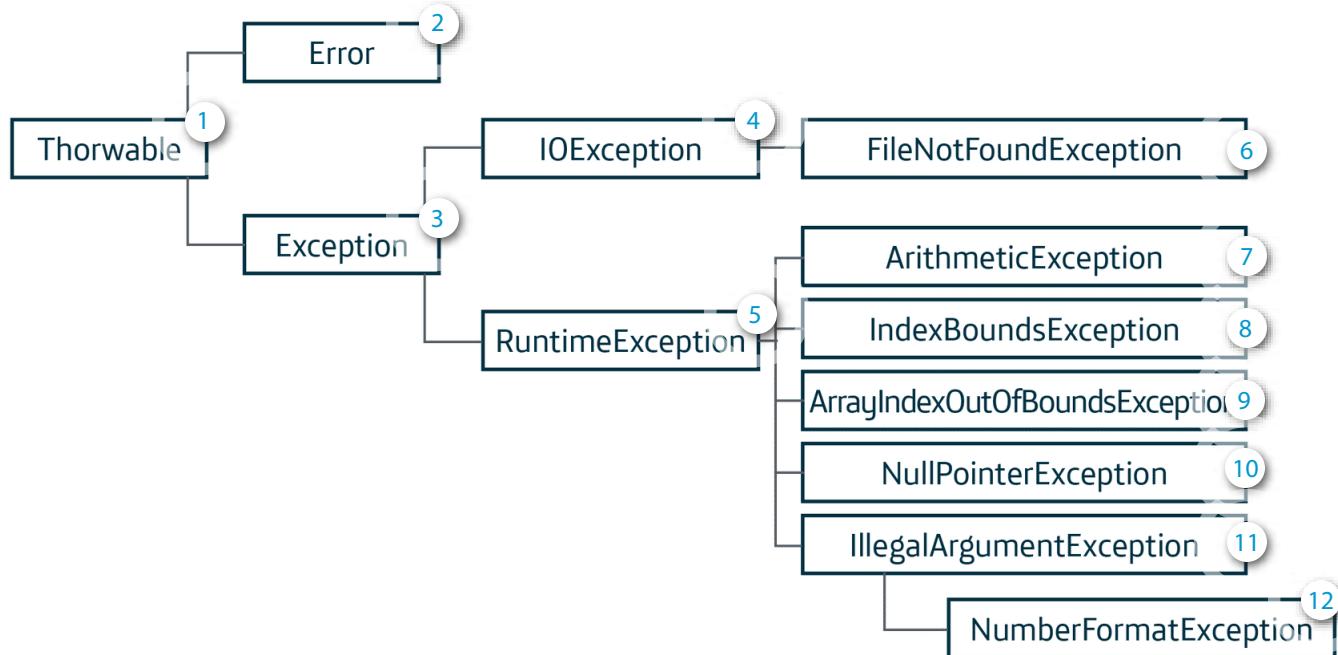
```
import java.io.FileNotFoundException;
import java.io.FileReader;

public class Principal {
    public static void main(String args[]) {
        try {
            ejecutarTarea1();
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
            return;
        }
    }

    public static void ejecutarTarea1() throws FileNotFoundException {
        ejecutarTarea2();
    }

    public static void ejecutarTarea2() throws FileNotFoundException {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}
```

Resumen de las excepciones más habituales



1 **Throwable**

Clase base de la que derivan todas las demás clases de excepción.

2 **Error**

Errores graves que escapan a nuestro control, tal como un fallo de la máquina virtual de Java.

3 **Exception**

Excepciones que sí podemos controlar y que se clasifican en:

- Comprobadas: las que derivan directamente de `Exception`. Es obligatorio gestionarlas.
- No comprobadas: las que derivan de `RuntimeException`. No es obligatorio gestionarlas.

4 **IOException**

Excepciones producidas durante operaciones de entrada y/o salida.

5 **RuntimeException**

Clase base de la que derivan todas las excepciones no comprobadas.

6 FileNotFoundException

Se está intentando acceder a un fichero que no existe.

7 ArithmeticException

Errores en operaciones aritméticas, como por ejemplo una división entre 0.

8 IndexOutOfBoundsException

Intento de acceso a una posición inexistente de una colección.

9 ArrayIndexOutOfBoundsException

Intento de acceso a un elemento de un *array* que existe, es decir, intento de salirse de los límites del *array*.

10 NullPointerException

Intento de acceder a una propiedad o a un método a partir de una referencia nula.

11 IllegalArgumentException

Se pasa un argumento a una función que resulta inválido para la operación que la función debe realizar.

12 NumberFormatException

Formato de número incorrecto. Ocurre cuando realizamos una conversión de texto a número y el texto no puede interpretarse como un número.

Resumen

Has terminado la lección, vamos a ver los puntos más importantes que hemos tratado.

- Cuando se produce una condición de error (excepción para Java), la máquina virtual genera un tipo de objeto especial con información sobre el suceso ocurrido; un objeto de tipo *Exception*.
- El método que provocó la excepción puede controlarla con un bloque *try ... catch* o propagarla hacia arriba con una declaración *throws* en la cabecera del método.
- Existen dos tipos de excepciones: **comprobadas**, donde es obligatorio usar *try ... catch* o propagarla, y **no comprobadas**, donde es opcional. Las excepciones controladas derivan directamente de *Exception*, mientras que las no controladas derivan de *RuntimeException* y este a su vez de *Exception*.

**PROEDUCA**