

EXCEPCIONES

Cuando sucede un evento anormal en la ejecución de un programa y lo detiene decimos que se ha producido una excepción. Cualquier programa bien escrito debe ser capaz de tratar las posibles excepciones que pueden ocurrir en su ejecución de manera inteligente y de recuperarse, si es posible, de ellos.

Ya hemos tenido algún contacto con las excepciones. Por ejemplo, cuando vimos el método *readLine()* de la clase *BufferedReader* para la lectura de cadenas por teclado, tuvimos que declarar la excepción *IOException* en el método *main()* para poder compilar el programa.

Con **los mecanismos de recuperación ante excepciones construimos programas robustos** y con capacidad de recuperación ante errores más o menos previsibles que se pueden producir en el momento en que se ejecuta el programa.

En Java una excepción es un objeto que avisa que ha ocurrido alguna condición inusual. Existen muchos objetos de excepción predefinidos, y también podremos crear los nuestros propios.

Cuando se capturan excepciones en Java las sentencias que pueden causar un error se deben insertar en un bloque formado por *try* y *catch*, a continuación, en *catch* debemos tratar esos posibles errores. También existe la posibilidad de lanzar una excepción cuando se produzca una determinada situación en la ejecución del programa, para hacerlo utilizaremos la instrucción *throw*

Durante esta unidad se estudiarán con detalle las excepciones. Analizaremos su funcionamiento y se presentarán las principales clases de excepciones existentes, además de conocer los mecanismos para su captura, propagación y creación.

1 EXCEPCIONES Y ERRORES

Una excepción es una situación anómala que puede producirse durante la ejecución de un programa.

Ejemplos: intento de división entera entre 0, un acceso a posiciones de un array fuera de los límites del mismo o un fallo durante la lectura de datos de la entrada/salida.

Mediante la **captura de excepciones**, Java proporciona un mecanismo que permite al **programa sobreponerse a estas situaciones, pudiendo el programador decidir las acciones a realizar para cada tipo de excepción** que pueda ocurrir.

Además de excepciones, en un programa Java pueden producirse errores. Un **error representa una situación anormal irreversible.**

Cada tipo de excepción está representada por una subclase de **Exception**, mientras que los errores son subclases de **Error**. Ambas clases, Exception y Error, son subclases de **Throwable**

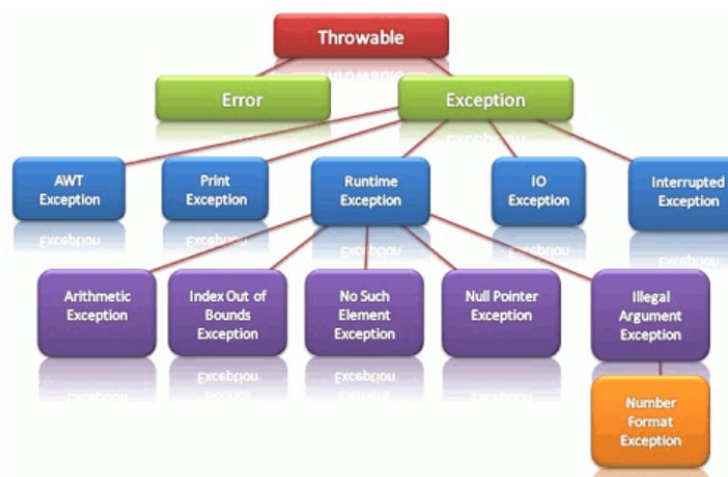


Ilustración 1 Subclases Error y Exception

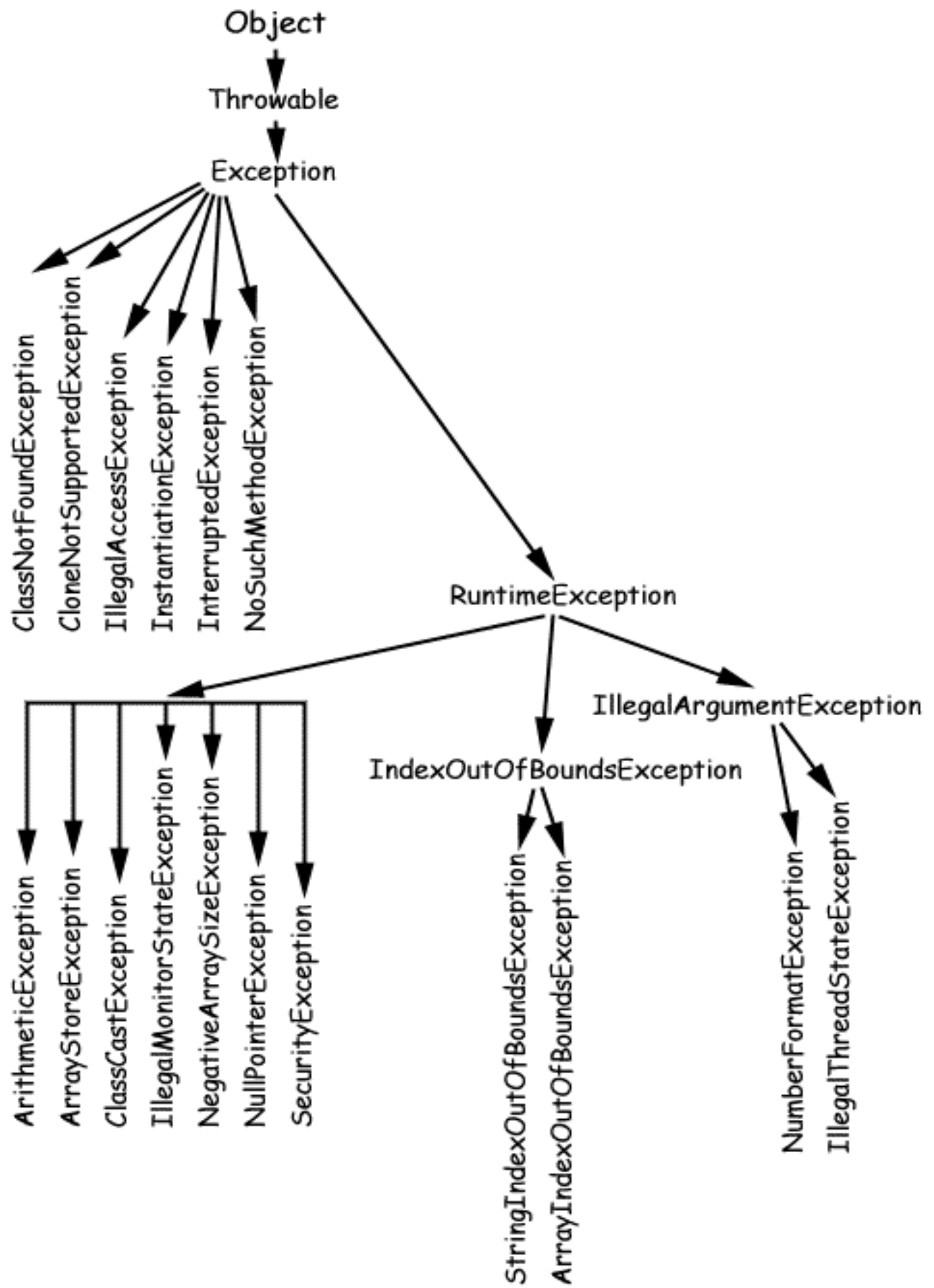
En Java las excepciones forman una **completa jerarquía de clases**, cuya clase base es **Throwable**. La clase **Throwable** tiene dos subclases: **Error** y **Exception**.

Un **Error** indica que se ha producido un fallo del que **no se puede recuperar la ejecución normal del programa**, por lo tanto, este tipo de errores no se pueden tratar.

Una **Exception** indicará una **condición anormal que puede ser resuelta** para evitar la terminación de la ejecución del programa. Hay varias **subclases** de la clase **Exception** conocidas como *excepciones predefinidas*, y una de ellas **RuntimeException**, a su vez, tiene numerosas subclases.

El usuario puede definirse sus propias excepciones, tan solo tendrá que extender la clase **Exception** y proporcionar la funcionalidad extra que requiera el tratamiento de esa excepción.

Todas las excepciones deben llevar un **mensaje asociado** a ellas al que se puede acceder utilizando el método **getMessage()**, que presentará un mensaje describiendo el error o la excepción que se ha producido. El método **toString()** aplicado a una excepción produce un resultado similar.



Excepciones predefinidas más comunes:

- ***ArithmeticException***

Las excepciones aritméticas son típicamente el resultado de una división por 0:

```
int i = 12 / 0;
```

- ***NullPointerException***

Se produce cuando se intenta acceder a una variable o método antes de ser definido.

- ***ClassCastException***

El intento de convertir un objeto a otra clase que no es válida.

```
y = (Prueba)x; // donde x no es de tipo Prueba
```

- ***NegativeArraySizeException***

Puede ocurrir si se intenta definir el tamaño de un array con un número negativo.

- ***ArrayIndexOutOfBoundsException***

Se intenta acceder a un elemento de un array que está fuera de sus límites.

- ***NoClassDefFoundException***

Se referenció una clase que el sistema es incapaz de encontrar.

Ejemplo de un programa que no trata excepciones:

```
public class EjExcepcion {  
    public static void main (String args[]) {  
        String cadenas[] = {  
            "Cadena 1",  
            "Cadena 2",  
            "Cadena 3",  
            "Cadena 4"  
        };  
  
        for (int i=0; i<=4; i++) System.out.println(cadenas[i]);  
    }  
}
```

Al ejecutar:

```
> java EjExcepcion  
Cadena 1  
Cadena 2  
Cadena 3  
Cadena 4  
java.lang.ArrayIndexOutOfBoundsException  
    at EjExcepcion.main(EjExcepcion.java, Compiled Code)  
Exception in thread "main" Process Exit...
```

2 CLASES DE EXCEPCIÓN

Al producirse una excepción en un programa, se crea un objeto de la subclase de *Exception* a la que pertenece la excepción. Como veremos más adelante, este objeto puede ser utilizado por el programa durante el tratamiento de la excepción para obtener información de esta.

En la siguiente imagen se muestra la jerarquía de clases con algunas de las excepciones más habituales que podemos encontrar en un programa.

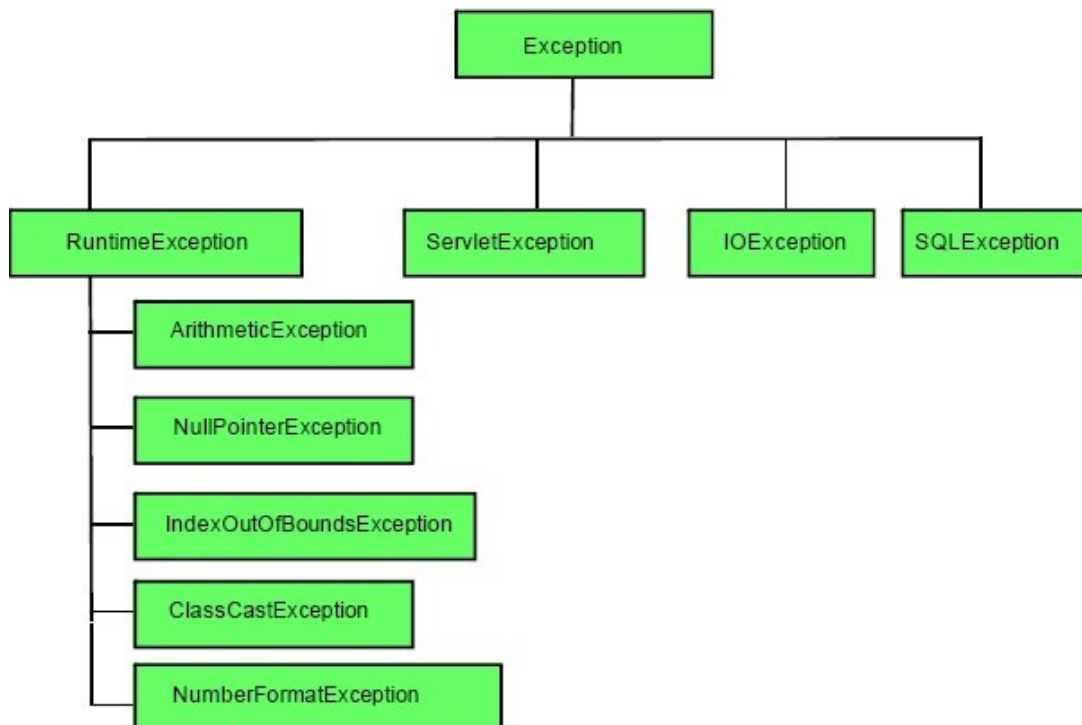


Ilustración 2 Jerarquía de clases de excepción

3.TIPOS DE EXCEPCIONES

Desde el punto de vista del tratamiento de una excepción dentro de un programa, hay que tener en cuenta que **todas estas clases de excepción se dividen en dos grandes grupos:**

- **Excepciones marcadas**
- **Excepciones no marcadas**

3.1 Excepciones marcadas

Se entiende por **excepciones** marcadas aquéllas **cuya captura es obligatoria**. Normalmente, este tipo de excepciones se producen al invocar a ciertos métodos de determinadas clases y son generadas (lanzadas) desde el interior de dichos métodos como consecuencia de algún fallo durante la ejecución de estos.

Todas las clases de excepciones, salvo **RuntimeException** y sus subclases, pertenecen a este tipo.

Un ejemplo de excepción marcada es **IOException**. Esta excepción es lanzada por el método *readLine()* de la clase **BufferedReader** cuando se produce un error durante la operación de lectura, lo que obliga al programa que va a hacer uso de dicho método a capturar la excepción, tal y como veremos más adelante.

Si en un bloque de código se invoca a algún método que puede provocar una excepción marcada y ésta no se captura, **el programa no compilará**.

DECLARACIÓN DE UNA EXCEPCIÓN

Los métodos que pueden provocar excepciones marcadas deben declarar éstas en la definición del método.

Para declarar una excepción **se utiliza** la palabra **throws**, seguida de la lista de excepciones que el método puede provocar:

```
public String readLine() throws IOException

public void service(...) throws ServletException, IOException
```

Así, siempre que vayamos a utilizar algún método que tenga declaradas excepciones, hemos de tener presente que estamos obligados a capturar dichas excepciones.

3.2 Excepciones no marcadas

Pertenecen a este grupo **todas las excepciones de tiempo de ejecución**, es decir, *RuntimeException* y todas sus subclases.

No es obligatorio capturar dentro de un programa Java una excepción no marcada, el motivo es que gran parte de ellas (*NullPointerException*, *ClassCastException*, etc.) se **producen como consecuencia de una mala programación**, por lo que la solución **no debe pasar por preparar el programa para que sea capaz de recuperarse ante una situación como esta, sino por evitar que se produzca**. Tan sólo las excepciones de tipo *ArithmeticException* es recomendable capturarlas.

Si durante la ejecución de un programa Java se produce una excepción y ésta no es capturada, la máquina virtual de Java provoca la finalización inmediata del mismo, enviando a la consola el volcado de pila con los datos de la excepción. Estos volcados de pila permiten al programador detectar fallos de programación durante la depuración de este.

Ejemplo: código Java que divide el número 20 por 0. Al ejecutar, nos devuelve la pila de datos del error por consola.

```
public class Division{
    public static void main (String[] argumentos){
        int numero=20;
        System.out.print(numero/0);
    }
}
```

Al ejecutar

```
C:\Curso22_33\DAM\01-PR\08_excepciones\code_exception> cmd /C
""C:\Program Files\Java\jdk-11.0.12\bin\java.exe" -cp
C:\Users\Usuario\AppData\Roaming\Code\User\workspaceStorage\e707c570d50915f8
778c894af62ce2ae\redhat.java\jdt_ws\code_exception_2c1c4f54\bin ejemplos.Division "
```

Exception in thread "main" java.lang.ArithmeticException: / by zero

at ejemplos.Division.main(Division.java:5)

Efecto de una excepción no controlada

4 CAPTURA DE EXCEPCIONES

Cuando se produce una excepción en un programa, se crea un objeto de la clase de excepción correspondiente y se "lanza" a la línea de código donde la excepción tuvo lugar.

El mecanismo de captura de excepciones de Java permite "atrapar" el objeto de excepción lanzado por la instrucción e indicar las diferentes acciones a realizar según la clase de excepción producida.

A diferencia de las excepciones, los errores representan fallos de sistema de los cuales el programa no se puede recuperar. Esto implica que no es obligatorio tratar un error en una aplicación Java, de hecho, aunque se pueden capturar al igual que las excepciones con los mecanismos que vamos a ver a continuación, lo recomendable es no hacerlo.

4.1 Los bloques *try...catch...finally*

Las instrucciones *try*, *catch* y *finally*, proporcionan una forma elegante y estructurada de capturar excepciones dentro de un programa Java, evitando la utilización de instrucciones de control que dificultarían la lectura del código y lo harían más propenso a errores.

La **sintaxis** de este bloque es la siguiente:

```
try{
    // código sensible que puede generar excepciones
    // en este ejemplo las TipoExcepcion1 y TipoExcepcion2
    // se pondrán tantos bloques catch como tipo excepciones queramos tratar

}catch (TipoExcepcion1 nombre_objeto){
    // los pasos que queremos hacer cuando se produce
    // la excepción TipoExcepcion1

}catch(TipoExcepcion2 nombre_objeto){
    // los pasos que queremos hacer cuando se produce
    // la excepción TipoExcepcion2

}finally{
    // código que siempre queremos ejecutar al finalizar
}

}
```

A continuación, se explica cada una de estas partes:

TRY

El bloque ***try*** delimita aquella o aquellas **instrucciones donde se puede producir una excepción**.

Cuando se produce la excepción, **el control del programa se transfiere al bloque *catch* definido para el tipo de excepción que se ha producido**, pasándole como parámetro la excepción lanzada. **Opcionalmente**, se puede disponer de bloque ***finally*** en el que definir un **grupo de instrucciones de obligada ejecución**.

CATCH

Un bloque ***catch*** **define las instrucciones que deberán ejecutarse en caso de que se produzca un determinado tipo de excepción**.

Sobre la utilización de los bloques ***catch***, se debe tener en cuenta siguiente:

- Se pueden definir tantos bloques ***catch*** como se considere necesario. Cada bloque ***catch*** servirá para tratar un determinado tipo de excepción, **no pudiendo haber dos o más *catch* que tengan declarada la misma clase de excepción**.
- Un bloque ***catch*** sirve para capturar cualquier excepción que se corresponda con el tipo declarado o cualquiera de sus subclases. Por ejemplo, un ***catch*** como el siguiente:

```
catch (RuntimeException e) {  
    .  
    .  
    .  
}
```

se ejecutaría al producirse cualquier excepción de tipo `NullPointerException`, `ArithmeticException`, etc.

Esto significa que una excepción podría ser tratada en diferentes ***catch***.

Ejemplo: la excepción `NullPointerException` podría ser tratada en un ***catch*** que capturase directamente dicha excepción y otro bloque ***catch*** capturase la excepción `RuntimeException`.

- Aunque haya varios posibles ***catch*** que puedan capturar una excepción, **sólo uno de ellos será ejecutado cuando ésta se produzca**. La búsqueda del bloque ***catch*** para el tratamiento de la excepción lanzada se realiza de forma secuencial, de modo que **el primer *catch* coincidente será el que se ejecutará**.

Una vez terminada la ejecución de este, el control del programa se transferirá al bloque ***finally*** o, si no existe, a la instrucción siguiente al último bloque ***catch***, independientemente de que hubiera o no más ***catch*** coincidentes.

EJEMPLO: siguiendo con el ejemplo anterior, en el que se intenta dividir un número por cero se captura la excepción que se mostró por consola `ArithmeticException`

```
public class Mecanismo1a {
    public static void main(String[] argumentos) {
        try{
            int s=4/0;
            System.out.println("El programa sigue");
        }catch (ArithmeticException e){
            System.out.println("Se ha producido un error al intentar dividir
por 0");
        }catch (Exception e){
            System.out.println("Excepción general");
        }

        System.out.println("Final del main");
    }
}
```

El resultado de ejecución es el siguiente:

Se ha producido un error al intentar dividir por 0

Final del main

- Del listado anterior se deduce otro punto importante para tener en cuenta en el tratamiento de excepciones: tras la ejecución de un bloque *catch*, **el control del programa nunca se devuelve al lugar donde se ha producido la excepción.**
- **En el caso de que existan varios *catch* cuyas excepciones están relacionadas por la herencia, los *catch* más específicos deben estar situados por delante de los más genéricos.** De no ser así, se producirá un error de compilación puesto que los bloques *catch* más específicos nunca se ejecutarán.

Compila correctamente	No compila
<pre>try{ }catch(IOException e){ }catch(Exception e){ }</pre>	<pre>try{ }catch(Exception e){ }catch(IOException e){ }</pre>

- Si se produce una excepción no marcada para la que no se ha definido bloque *catch*, ésta será propagada por la pila de llamadas hasta encontrar algún punto en el que se trate la excepción. De no existir un tratamiento para la misma, la máquina virtual abortará la ejecución del programa y enviará un volcado de pila a la consola.

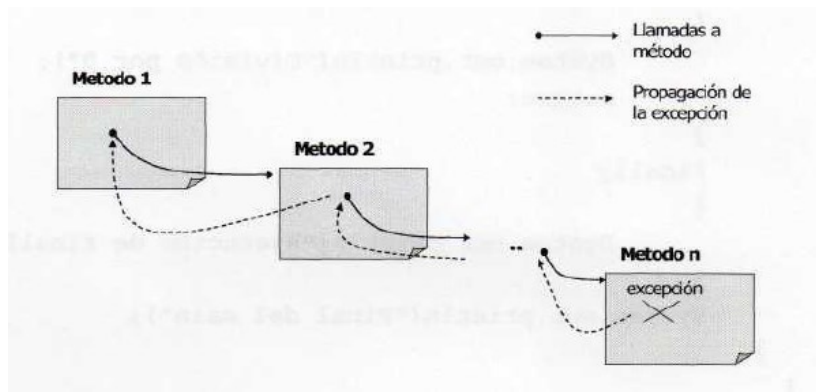


Ilustración 3 Propagación de una excepción en la pila de llamadas

- Los bloques *catch* son opcionales. **Siempre que exista un bloque *finally*, la creación de bloques *catch* después de un *try* es opcional.** Si no se cuenta con un bloque *finally*, entonces es obligatorio disponer de, al menos, un bloque *catch*.

FINALLY

Su uso es opcional. El bloque *finally* **se ejecutará tanto si se produce una excepción como si no**, garantizando así que un determinado conjunto de instrucciones siempre sea ejecutadas.

Si se produce una excepción en *try*, el bloque *finally* se ejecutará después del *catch* para tratamiento de la excepción. En caso de que no hubiese ningún *catch* para el tratamiento de la excepción producida, el bloque *finally* se ejecutaría antes de propagar la excepción.

Si no se produce excepción alguna en el interior de *try*, el bloque *finally* se ejecutará tras la última instrucción del *try*.

El siguiente código de ejemplo muestra el funcionamiento de *finally*.

```
public class Mecanismo1b {
    public static void main(String[] argumentos) {
        try{
            int s=4/0;
            System.out.println("El programa sigue");
        }catch (ArithmeticException e){
            System.out.println("Se ha producido un error al intentar dividir
por 0");
        }finally{
            System.out.println ("Aquí pondremos lo que queremos ejecutar
siempre");
        }

        System.out.println ("Final del main");
    }
}
```

Tras la ejecución se mostrará en pantalla lo siguiente:
Se ha producido un error al intentar dividir por 0
Aquí pondremos lo que queremos ejecutar siempre
Final del main

Esto demuestra que, **aun existiendo una instrucción para la salida del método (return), el bloque finally se ejecutará antes de que esto suceda.**

Retomamos el ejemplo del inicio EjExcepcion en el cual no se trataba la excepción `ArrayIndexOutOfBoundsException` y lo modificamos para hacerlo más robusto.

```
public class EjExcepcionBien {
    public static void main (String args[]) {
        String cadenas[] = {"Cadena 1","Cadena 2","Cadena 3","Cadena
4"};
        try {
            for (int i=0; i<=4; i++) System.out.println(cadenas[i]);
        } catch( ArrayIndexOutOfBoundsException aie ) {
            System.out.println("\nError: Fuera del índice del array\n");
        } catch( Exception e ) {
            // Captura cualquier otra excepción
            System.out.println("Excepción: " + e.toString() );
        } finally {
            System.out.println( "Esto se imprime siempre." );
        }
    }
}
```

Al ejecutar:

```
> java EjExcepcionBien
Cadena 1
Cadena 2
Cadena 3
Cadena 4
```

```
Error: Fuera del índice del array
Esto se imprime siempre.
```

Vamos a modificar el programa para que se produzca otro tipo de excepción, en este caso una división por cero y se ejecute el bloque del segundo *catch*:

```

public class EjExcepcionBien {
    public static void main (String args[]) {
        int valor = 0;
        String cadenas[] = {
            "Cadena 1",
            "Cadena 2",
            "Cadena 3",
            "Cadena 4"
        };
        try {
            for (int i=0; i<=4; i++) {
                System.out.println(cadenas[i]);
                valor = i/0;
            }
        } catch( ArrayIndexOutOfBoundsException aie ) {
            System.out.println("\nError: Fuera del índice del array\n");
        } catch( Exception e ) {
            // Captura cualquier otra excepción
            System.out.println("Excepción: " + e.toString() );
        } finally {
            System.out.println( "Esto se imprime siempre." );
        }
    }
}

```

- Como resultado de la ejecución del programa obtendremos:

```

> java EjExcepcionBien
Cadena 1
Excepción: java.lang.ArithmeticException: / by zero
Esto se imprime siempre.

```

- En este caso el programa ha ejecutado:

```

catch( Exception e ) {
    // Captura cualquier otra excepción
    System.out.println("Excepción: " + e.toString() );
}

```

que funciona como un mecanismo general de captura de excepciones, es decir, cualquier excepción no tratada anteriormente es tratada aquí.

4.2 Propagación de una excepción

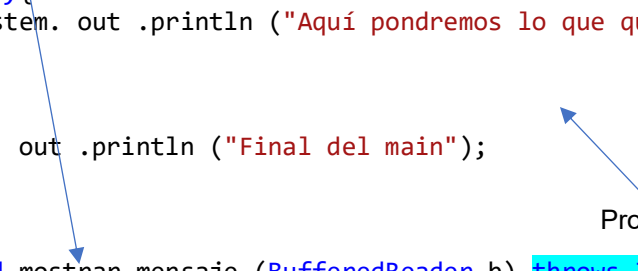
Anteriormente indicábamos que **si en el interior de un método se produce una excepción que no es capturada**, bien porque no está dentro de un *try* o bien porque no existe un *catch* para su tratamiento, ésta **se propagará por la pila de llamadas**.

En el caso de las excepciones marcadas, hemos visto cómo éstas deben ser capturadas obligatoriamente en un programa. Sin embargo, en el caso de que no tenga previsto ninguna acción particular para el tratamiento de una determina excepción de este tipo, es posible propagar la excepción sin necesidad de capturarla, dejando que sean otras partes del programa las encargadas de definir acciones para su tratamiento.

Para propagar una excepción sin capturarla, basta con declararla en la cabecera del método en cuyo interior puede producirse.

En el siguiente ejemplo el método *main()* es el que captura la excepción producida en *mostrar_mensaje()*. Si en *main()* se hubiera optado por propagar también la excepción, al ser el último método de la pila de llamadas ésta se propagará a la máquina virtual, cuyo comportamiento por defecto será, como ya sabemos, interrumpir la ejecución del programa y generar un volcado de pila en la consola.

```
public class Mecanismo2 {  
  
    public static void main(String[] argumentos) {  
        BufferedReader lector = new BufferedReader (new InputStreamReader  
(System.in));  
        try{  
            mostrar_mensaje(lector);  
        }catch (IOException e){  
            System.out.println("Fallo de lectura");  
        }finally{  
            System.out.println ("Aquí pondremos lo que queremos ejecutar  
siempre");  
        }  
  
        System.out.println ("Final del main");  
    }  
  
    static void mostrar_mensaje (BufferedReader b) throws IOException{  
        //el método declara la excepción para que sea propagada.  
        System.out.print("Escribe mensaje: ");  
        String str=b.readLine(); //esta línea provoca una excepción  
        System.out.println (str);  
    }  
}
```



The diagram consists of two blue arrows. One arrow originates from the line `mostrar_mensaje(lector);` inside the `try` block of the `main` method and points to the `throws IOException` declaration in the signature of the `mostrar_mensaje` method. A second arrow originates from the `throws IOException` declaration and points to the right, towards the text 'Propaga la excepción'.

Propaga la excepción

Otro ejemplo: en el siguiente ejemplo se capturan dos posibles excepciones

```
public static double capturanum() throws NumberFormatException, IOException
{
    BufferedReader entrada=
        new BufferedReader(new InputStreamReader(System.in));

    String txt = entrada.readLine();
    double num = Double.parseDouble(txt);
    return num;
}
}
```

5 LANZAMIENTO DE UNA EXCEPCIÓN

En determinados casos puede resultar útil generar y lanzar una excepción desde el interior de un determinado método. Esto puede utilizarse como un **medio para enviar un aviso a otra parte del programa, indicándole que algo está sucediendo y no es posible continuar con la ejecución normal del método.**

Para lanzar una excepción desde código utilizamos la expresión:

throw objeto _excepcion;

Donde ***objeto_excepcion*** es un objeto de alguna subclase de **Exception**.

El flujo de la ejecución se detiene inmediatamente después de la sentencia `throw`, y nunca se llega a la sentencia siguiente, ya que el control sale del bloque `try` y pasa a un manejador `catch` cuyo tipo coincide con el del objeto. Si se encuentra, el control se transfiere a esa sentencia. Si no, se inspeccionan los siguientes bloques hasta que el gestor de excepciones más externo detiene el programa.

El siguiente ejemplo muestra un caso práctico en el que un método, encargado de realizar una operación de extracción de dinero en una cuenta bancaria lanza una excepción cuando no se dispone de saldo suficiente para realizar la operación:

```

    Fichero Cuenta.java
public class Cuenta {
    private double saldo;
    public Cuenta(){
        saldo = 0;
    }
    public void ingresar (double c)
    {
        saldo += c;
    }

    //el método declara la excepción que puede provocar
    public void extraer (double c)throws Exception
    {
        if (saldo<c) {
            //creación y lanzamiento de la excepción
            throw new Exception();
        }else{
            saldo -= c;
        }
    }
    public double getSaldo(){
        return saldo;
    }
}

    Fichero Cajero.java

public class Cajero{
    public static void main (String [] args){
        Cuenta c = new Cuenta();
        try{
            c.ingresar(100);
            c.extraer (20);
            c.extraer (120);
        }catch (Exception e){
            System.out.println ("La cuenta no puede " + " quedar en
números rojos");
        }
        System.out.println("Fin del programa");
    }
}

```

Al ejecutar el ejemplo anterior:

```

    La cuenta no puede  quedar en números rojos
    Fin del programa

```

En el código del ejemplo, cuando se lanza una excepción marcada desde un método (todas lo son salvo `RuntimeException` y sus subclases) ésta **debe ser declarada en la cabecera del método** para que se pueda propagar al punto de llamada al mismo.

Las excepciones también se pueden relanzar desde un *catch*:

```
catch (IOException e)
{
    throw e;
}
```

En este caso, a pesar de estar capturada por un *catch* y dado que vuelve a ser *lanzada*, la excepción `IOException` también deberá declararse en la cabecera del método.

A continuación, se muestra otro ejemplo de tratamiento de excepciones con `throw`. El programa solicita dos números (numerador y denominador) para hacer una división. Se controlan las excepciones de división por cero (`ArithmeticException`), la excepción de que el usuario no teclee números (`NumberFormatException`), la lectura que siempre lanza `IOException` y cualquier excepción general (`Exception`).

```
import java.io.*;
public class ExcepcionThrow {
    public static void main(String args[]) {
        double op1, op2, resd;
        String resp = "";
        BufferedReader entrada = new BufferedReader(new
InputStreamReader(System.in));
        //bucle infinito aunque se produzca una excepción
        while (true) {
            try {
                System.out.println("\n##### Nueva División
#####");
                System.out.print("\nNumerador: ");
                op1 = Double.parseDouble(entrada.readLine());
                System.out.print("\nDenominador: ");
                op2 = Double.parseDouble(entrada.readLine());
                if (op2 == 0) {
                    throw new ArithmeticException("División por
cero");
                }
                resd = op1 / op2;
                System.out.println("\nResultado: " +
Double.toString(resd));
                System.out.println("\nPulse s para salir.");
                resp = entrada.readLine().toUpperCase();
                if (resp.equals("S")) { return; }
            } catch (ArithmeticException aie) {
                System.out.println("\nError aritmético: " +
aie.toString());
            } catch (NumberFormatException nfe) {
                System.out.println("\nError de formato numérico: "
+ nfe.toString());
            } catch (IOException ioe) {
```



```

        System.out.println("\nError de entrada/salida: " +
ioe.toString());
    } catch (Exception e) {
        System.out.println("Excepción: " + e.toString());
    }
}
}
}
}

```

6 MÉTODOS PARA EL CONTROL DE UNA EXCEPCIÓN

Todas las clases de excepción heredan una serie de métodos de `Throwable` que pueden ser utilizados en el interior de los `catch` para completar las acciones de tratamiento de la excepción.

Los métodos más importantes son:

- `String getMessage()`. Devuelve un mensaje de texto asociado a la excepción, dependiendo del tipo de objeto de excepción sobre el que se aplique.
- `void printStackTrace()`. Envía a la consola el volcado de pila asociado a la excepción. Su uso puede ser tremendamente útil durante la fase de desarrollo de la aplicación, ayudando a detectar errores de programación causantes de muchas excepciones.
- `void printStackTrace(PrintStream s)`. Esta versión de `printStackTrace()` permite enviar el volcado de pila a un objeto `PrintStream` cualquiera, por ejemplo, un fichero log.

7 CLASES DE EXCEPCIÓN PERSONALIZADAS

Cuando un método necesita lanzar una excepción como forma para notificar una situación anómala, puede suceder que las clases de excepción existentes no se adecuen a las características de la situación que quiere notificar.

Por ejemplo, en el caso anterior de la cuenta bancaria no tendría mucho sentido lanzar una excepción de tipo `NullPointerException` o `IOException` cuando se produce una situación de saldo insuficiente.

En estos casos resulta más práctico definir una clase de excepción personalizada, subclase de **Exception**, que se ajuste más a las características de la excepción que se va a tratar.

Para el ejemplo de la cuenta bancaria, podríamos definir la siguiente clase de excepción:

```

public class SaldoInsuficienteException extends Exception{

    public SaldoInsuficienteException (String mensaje){

        super(mensaje);

    }

}

```

La cadena de texto recibida por el constructor permite personalizar el mensaje de error obtenido al llamar al método `getMessage()`, para ello, es necesario suministrar dicha cadena al constructor de la clase `Exception`.

A continuación tenemos una nueva versión del programa anterior. En este caso, **se utiliza una clase de excepción personalizada, `SaldoInsuficienteException`**, para representar la situación de saldo negativo en la cuenta:

1º Definición de la excepción propia.

```

public class SaldoInsuficienteException extends Exception{
    public SaldoInsuficienteException (String mensaje){
        super(mensaje);
    }
}

```

2º Lanzamiento de la excepción cuando se produce una situación anómala.

```

public class Cuenta {
    private double saldo;
    public Cuenta(){
        saldo = 0;
    }
    public void ingresar (double c)
    {
        saldo += c;
    }

    //el método declara la excepción que puede provocar
    public void extraer (double c) throws SaldoInsuficienteException
    {
        if (saldo < c) {
            //creación y lanzamiento de la excepción
            throw new SaldoInsuficienteException("Error: Saldo en números
rojos");
        }else{
            saldo -= c;
        }
    }
    public double getSaldo(){
        return saldo;
    }
}

```

3° En la clase de más alto nivel se programan secciones de código que recogen la excepción creada.

```
public class Cajero{
    public static void main (String [] args){
        Cuenta c = new Cuenta();
        try{
            c.ingresar(100);
            c.extraer (20);
            c.extraer (120);
        }catch (SaldoInsuficienteException e){
            System.out.println (e.getMessage());
        }
        System.out.println("Fin del programa");
    }
}
```

El siguiente programa hace uso de una excepción definida por el usuario. El programa realiza la media de una serie de notas introducidas por el usuario, cuando éste intenta poner una nota inferior a 0 o superior a 10, salta la excepción NotaMal. También se han tenido en cuenta otras posibles excepciones predefinidas como errores de formato numérico, errores de entrada salida, etc.

Fichero Nota.java

```
import java.io.*;
public class Notas {
    public static void main(String args[]) {
        double media = 0, total = 0, notanum = 0;
        int contador = 0;
        String notatxt = "";
        BufferedReader entrada = new BufferedReader(new
        InputStreamReader(System.in));
        while (!notatxt.equals("Z")) {
            try {
                System.out.print("\nTeclee calificación (0-10), Z para
terminar: ");
                notatxt = entrada.readLine().toUpperCase();
                notanum = Double.parseDouble(notatxt);
                if (notanum < 0 || notanum > 10) {
                    throw new NotaMal();
                }
                total += notanum;
                contador++;
            } catch (NotaMal nm) {
                System.out.println("\n" + nm.getMessage());
            } catch (NumberFormatException nfe) {
                if (!notatxt.equals("Z")) {
                    System.out.println("\nError de formato numérico: " +
nfe.toString());
                }
            } catch (IOException ioe) {
                System.out.println("\nError de entrada/salida: " +
ioe.toString());
            } catch (Exception e) {
                // Captura cualquier otra excepción
                System.out.println("Excepción: " + e.toString());
            }
        }
    }
}
```

```

    }
}

if (contador != 0) {
    media = (double) total / contador;
    System.out.println("\nEl promedio del grupo es: " + media);
} else {
    System.out.println("\nNo se introdujeron calificaciones.");
}
}
}

class NotaMal extends Exception {

    public NotaMal() {
        super("Excepción definida por el usuario: NOTA INCORRECTA.");
    }
}

```

Podemos modificar la excepción NotaMal para que admita un mensaje que le pasamos como parámetro.

```

import java.io.*;
public class Notas2 {
    public static void main(String[] args) {
        double media = 0, total = 0, notanum = 0;
        int contador = 0;
        String notatxt = "";
        BufferedReader entrada = new BufferedReader(new
InputStreamReader(System.in));
        while (!notatxt.equals("Z")) {
            try {
                System.out.print("\nTeclee calificación (0-10), Z para
terminar: ");
                notatxt = entrada.readLine().toUpperCase();
                notanum = Double.parseDouble(notatxt);
                if (notanum < 0) {
                    throw new NotaMal2("No se admiten calificaciones menores
que 0.");
                } else if (notanum > 10) {
                    throw new NotaMal2("No se admiten calificaciones mayores
que 10.");
                }
                total += notanum;
                contador++;
            } catch (NotaMal2 nm) {
                System.out.println("\n" + nm.getMessage());
            } catch (NumberFormatException nfe) {
                if (!notatxt.equals("Z")) {
                    System.out.println("\nError de formato numérico: " +
nfe.toString());
                }
            } catch (IOException ioe) {
                System.out.println("\nError de entrada/salida: " +
ioe.toString());
            } catch (Exception e) {
                // Captura cualquier otra excepción
                System.out.println("Excepción: " + e.toString());
            }
        }
    }
}

```

```

    }

    if (contador != 0) {
        media = (double) total / contador;
        System.out.println("\nEl promedio del grupo es: " + media);
    } else {
        System.out.println("\nNo se introdujeron calificaciones.");
    }
}

}

class NotaMal2 extends Exception {

    public NotaMal2() {
        super("Excepción definida por el usuario: NOTA INCORRECTA.");
    }

    public NotaMal2(String msg) {
        super("Excepción definida por el usuario: " + msg);
    }
}

```

En el siguiente diagrama se muestra gráficamente cómo se propaga una excepción que se genera en un método, a través de la pila de llamadas durante la ejecución de un programa:

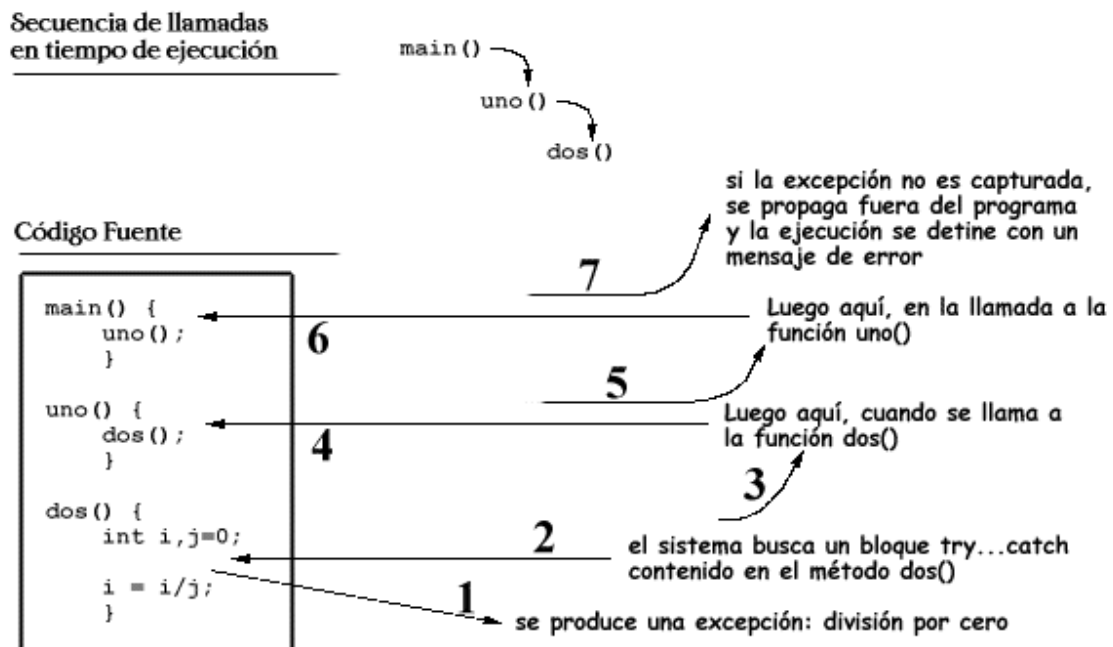


Ilustración 4 Cómo se propaga una excepción

Cuando una excepción no es tratada en el método en donde se produce, se busca en el bloque `try..catch` del método que hizo la llamada. Si la excepción se propaga de todas formas hasta lo alto de la pila de llamadas sin encontrar un controlador específico para la excepción, entonces la ejecución se detendrá dando un mensaje de error con la excepción no tratada.

En el siguiente programa se hace uso del método **`printStackTrace(System.out)`**. Invocando a este método sobre una excepción se volcará a pantalla todas las

llamadas hasta el momento en donde se generó la excepción.

```
// PropExcepciones.java
public class PropExcepciones {

    public static void main(String args[]) {
        try {
            metodo1();
        } catch (ExcepcionUsuario eu) {
            System.err.println("\nCapturada excepción en método1: " +
                eu.getMessage() +
                "\n\nTrazado de la pila:\n");
            eu.printStackTrace(System.out);
        }
    }

    public static void metodo1() throws ExcepcionUsuario {
        metodo2();
    }

    public static void metodo2() throws ExcepcionUsuario {
        metodo3();
    }

    public static void metodo3() throws ExcepcionUsuario {
        throw new ExcepcionUsuario();
    }
}

class ExcepcionUsuario extends Exception {

    public ExcepcionUsuario() {
        super("Excepción definida por el usuario.");
    }
}
```

La ejecución del programa presentará por pantalla:

```
>java PropExcepciones
Capturada excepción en método1: Excepción definida por el usuario.
```

Trazado de la pila:

```
ExcepcionUsuario: Excepción definida por el usuario.
    at PropExcepciones.metodo3(PropExcepciones.java:29)
    at PropExcepciones.metodo2(PropExcepciones.java:24)
    at PropExcepciones.metodo1(PropExcepciones.java:19)
    at PropExcepciones.main(PropExcepciones.java:7)
```