

DESARROLLO DE APLICACIONES WEB

Unidad 9

Gestión de excepciones

1r DAW

IES La Mola de Novelda

Departament d'informàtica

ÍNDIX

1.- Introducció.....	3
2.- Excepcions.....	4
2.1.- Ejemplos.....	5
2.2.- Errores en tiempo de compilación.....	6
2.3.- Errores en tiempo de ejecución.....	6
3.- Bloque: try - catch - finally.....	7
3.1.- Bloque <i>catch</i>	8
3.2.- Bloque <i>finally</i>	12
4.- Control de varios tipos de excepciones.....	12
4.1.- Cláusulas <i>catch</i> múltiples.....	14
4.2.- Objeto <i>Exception</i>	16
5.- Lanzar excepciones (<i>throw</i>).....	17
5.1.- ¿Por qué lanzar excepciones?.....	17
5.2.- Cómo lanzar un excepción.....	17
6.- Declaración de un método con <i>throws</i>	20
7.- Creación de excepciones propias o personalizadas.....	21
8.- Recomendaciones.....	29

Unidad 9: GESTIÓN DE EXCEPCIONES

1.- INTRODUCCIÓN

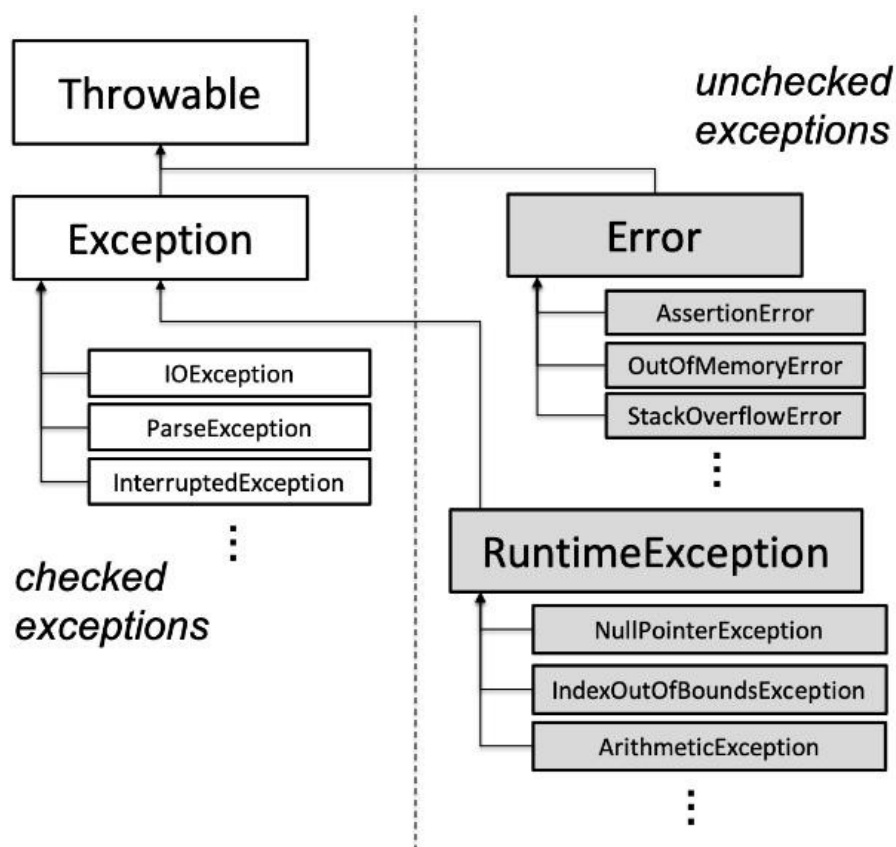
En la ejecución de un programa Java se pueden producir dos tipos de problemas: Errores y Excepciones.

- **Errores:** Se trata de problemas que no son producidos por una ejecución normal del programa, y ajenos a su control. Por ejemplo, *OutOfMemoryError*, *VirtualMachineError* (problemas en la JVM, como *out of memory*) o en el ordenador, son problemas que normalmente una aplicación no puede recuperarse, fallando irremisiblemente.

Como son problemas ajenos al control del programa, no es necesario (ni fácil) intentar controlarlos o manejarlos

- **Excepciones:** Se provocan por el propio código, al ejecutarse con resultados inesperados o mal programados. Es posible manejarlas y hacer que el programa se recupere y siga funcionando.

Ambos problemas se traducen a clases en Java. Todos ellos heredan de **Throwable**:



Al gráfico podemos comprobar que existen dos tipos de excepciones:

- **Checked exceptions**: Las excepciones comprobadas o verificadas son aquellas condiciones excepcionales que el compilador comprueba en el momento de la compilación. Una excepción verificada obliga a utilizar ***try-catch*** o ***throws***. Todas las excepciones excepto *Error*, *RuntimeException* y sus subclases son excepciones verificadas. Por ejemplo *IOException* y todas sus subclases, *SQLException* y todas sus subclases, etc.
- **Unchecked exceptions**: Las excepciones no comprobadas son aquellas condiciones especiales que el compilador no comprueba en el momento de la compilación. Las excepciones no comprobadas se comprueban en tiempo de ejecución. Una excepción no comprobada no nos obliga a utilizar ***try-catch*** o ***throws***. *RuntimeException* y todas sus subclases son excepciones no comprobadas. Esta excepción la puede evitar el programador. Por ejemplo: *NullPointerException*, *ArithmeticException*, etc.

2.- EXCEPCIONES

Una **excepción** es un error semántico que se produce en tiempo de ejecución. Aunque un código sea correcto sintácticamente (es código Java válido y puede compilarse), es posible que durante su ejecución se produzcan errores inesperados, como por ejemplo:

- Dividir por cero.
- Intentar acceder a una posición de un array fuera de sus límites.
- Al llamar al *nextInt()* de un *Scanner* el usuario no introduce un valor entero.
- Intentar acceder a un fichero que no existe o que está en un disco duro corrupto.
- Etc.

Cuando esto ocurre, la máquina virtual Java no detiene el programa inmediatamente, sino que crea un objeto de la clase ***Exception*** (las excepciones en Java son objetos) y se notifica el hecho al sistema de ejecución. Este objeto será de algunas de las clases que hereda de ***Exception***, tanto de las que son *checked* como de las *unchecked*. Se dice que se ha lanzado una excepción (*Throwing Exception*). La clase ***Exception*** deriva de la clase base ***Throwable***.

Además, un método se dice que es capaz de tratar una excepción (*Catch Exception*) si ha previsto el error que se ha producido y las operaciones a realizar para "recuperar" el programa de ese estado de error. No es suficiente capturar la excepción, si el error no se trata tan solo

conseguiremos que el programa no se pare, pero el error puede provocar que los datos o la ejecución no sean correctos.

En el momento en que es lanzada una excepción, la máquina virtual Java recorre la pila de llamadas de métodos en busca de alguno que sea capaz de tratar la clase de excepción lanzada. Para ello, comienza examinando el método donde se ha producido la excepción; si este método no es capaz de tratarla, examina el método desde el que se realizó la llamada al método donde se produjo la excepción y así sucesivamente hasta llegar al último de ellos. En caso de que ninguno de los métodos de la pila sea capaz de tratar la excepción, la máquina virtual Java muestra un mensaje de error y el programa termina. Por lo tanto, si el control de programa no encuentra mecanismos de control de excepciones, el programa va subiendo de método en método hasta llegar al *main* inicial, y simplemente termina, emitiendo un informe del error (*StackTrace*).

Los programas escritos en Java también pueden lanzar excepciones explícitamente mediante la instrucción **throw**, lo que facilita la devolución de un "código de error" al método que invocó el método que causó el error.

En resumen: en Java, cuando se produce un error de código, realmente se produce una Excepción, que conlleva la creación de un objeto **Exception**, y esta excepción es lanzada al programa, donde éste debe intentar procesar la excepción. ¿Cómo? pues mediante los mecanismos de captura de errores que veremos a continuación

2.1.-EJEMPLOS

Veamos algunos ejemplos de excepciones:

FORZAR UNA EXCEPCIÓN AL INTENTAR DIVIDIR UN NÚMERO ENTRE 0

```
int x = 3, y = 0, div;  
div = x/y;  
System.out.println("Resultado: "+div);
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at excepciones.EjemplosExcepciones.main(EjemplosExcepciones.java:6)
```

La máquina virtual Java ha detectado una condición de error, la división por 0, y ha creado un objeto de la clase **java.lang.ArithmeticException**. Como el método donde se ha producido la excepción no es capaz de tratarla, la máquina virtual Java finaliza el programa en la línea 6 y muestra un mensaje de error con la información sobre la excepción que se ha producido.

FORZAR UNA EXCEPCIÓN DE CONVERSIÓN

```
String cadena = "56s";  
int num;  
num = Integer.parseInt(cadena);  
System.out.println("Numero: "+num);
```

Debido a que la cadena no tiene el formato adecuado ("56s" no representa un número válido), el método `Integer.parseInt(...)` no puede convertirla a un valor de tipo `int` y lanza la excepción **NumberFormatException**. La máquina virtual Java finaliza el programa en la línea 10 y muestra por pantalla la información sobre la excepción

```
Exception in thread "main" java.lang.NumberFormatException: Create breakpoint : For input string: "56s"
    at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)
    at java.base/java.lang.Integer.parseInt(Integer.java:668)
    at java.base/java.lang.Integer.parseInt(Integer.java:786)
    at excepciones.EjemplosExcepciones.main(EjemplosExcepciones.java:19)
```

que se ha producido.

FORZAR UNA EXCEPCIÓN DE LÍMITES DEL VECTOR

```
int v[] = {1,2,3};
```

```
int elem;
elem = v[5];
System.out.println("Elemento: "+elem);
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Create breakpoint : Index 5 out of bounds for length 3
    at excepciones.EjemplosExcepciones.main(EjemplosExcepciones.java:14)
```

Al intentar acceder a una posición que sobrepasa el tamaño del vector se produce una excepción de tipo ***ArrayIndexOutOfBoundsException***. La máquina virtual de java finaliza el programa en la línea 3 y muestra el mensaje de error sobre la excepción que se ha producido.

Las excepciones son unos tipos de errores que se cometen en la programación y que tendremos que tratar para que la aplicación pueda continuar con su ejecución. Por lo tanto, programando en Java se pueden producir errores en la compilación y en la ejecución.

2.2.-ERRORES EN TIEMPO DE COMPILACIÓN

Es el más común. Generalmente ocurre cuando hay alguna sentencia que no está bien escrita, por ejemplo si falta un punto y coma al final de una línea, cuando hay comillas o paréntesis que se abren pero no se cierran, al intentar asignar una cadena de caracteres a una variable que es de tipo entero, etc.

Si un programa da errores en tiempo de compilación, no se crea el archivo de ***bytecode*** con la extensión ***.class*** y, por lo tanto, hasta que no se arreglen los fallos, no compilará y, en consecuencia, tampoco se podrá ejecutar.

- Ejemplo: El siguiente código da un error de compilación.

```
System.out.println(";Hola mundo!");
```

Por lo tanto, los errores en tiempo de compilación se suelen producir cuando el código no está bien escrito. Por ejemplo, cuando falta el punto y coma al final de la sentencia. Hasta que no se arreglen, no se genera el archivo con la extensión ***.class***.

2.3.-ERRORES EN TIEMPO DE EJECUCIÓN

Existen otros errores que se producen en tiempo de ejecución a los que llamaremos excepciones. El programa compila y se puede ejecutar pero, por algún motivo, se produce un fallo y el programa se “rompe”. Un buen programador debe prever esta situación y debe saber encauzar el programa para que quede todo bajo control.

- Ejemplo: Programa que calcula la media de dos números

```
Scanner s = new Scanner(System.in);
System.out.println("Este programa calcula la media de dos números");
System.out.print("Introduzca el primer número: ");
double numero1 = Double.parseDouble(s.nextLine());
System.out.print("Introduzca el segundo número: ");
double numero2 = Double.parseDouble(s.nextLine());
System.out.println("La media es " + (numero1 + numero2) / 2);
```

El programa compila y se ejecuta correctamente. El problema está cuando introduzco una palabra en lugar de un número.

Este programa calcula la media de dos números

Introduzca el primer número: hola

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "hola"
    at java.base/jdk.internal.math.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2054)
    at java.base/jdk.internal.math.FloatingDecimal.parseDouble(FloatingDecimal.java:110)
    at java.base/java.lang.Double.parseDouble(Double.java:651)
    at excepciones.MediaDeDosNumeros.start(MediaDeDosNumeros.java:14)
    at excepciones.MediaDeDosNumeros.main(MediaDeDosNumeros.java:8)
```

El programa ha terminado de forma repentina. ¿Y si, en lugar de ser un simple ejemplo que calcula una media, se tratase de la aplicación que controla un cajero automático o, peor aún, el programa de pilotaje de un avión?

Los errores en tiempo ejecución son también llamados **excepciones**. El programa compila y, en principio, se ejecuta bien... hasta que se dan determinadas circunstancias que provocan un fallo. Es necesario preveerlos para que el programa no termine de forma inesperada.

3.- BLOQUE: TRY - CATCH - FINALLY

El bloque **try - catch - finally** (manejadores de excepciones) sirve para encauzar el flujo del programa de tal forma que, si se produce una excepción, no se termine de forma drástica y se pueda reconducir la ejecución de una manera controlada. Los tres manejadores de excepciones trabajan conjuntamente y su significado es el siguiente:

- Bloque **try** (intentar): Código que podría lanzar una excepción y que queremos controlar. Si en cualquiera de estas instrucciones se produce un error, el control del programa salta directamente al bloque **catch** correspondiente al error que se ha producido..
- Bloque **catch** (capturar): Código que manejará la excepción si es lanzada. Cuando se acaba la ejecución del bloque **catch**, el programa se salta el resto de bloques **catch** si los hubiera, y

continúa tras todos ellos, salvo que exista *finally*. Conseguimos que el error no pare el programa, que continúa tras la instrucción *try-catch*.

- Bloque ***finally*** (finalmente): Código que se ejecuta tanto si hay excepción como si no. Esta parte es opcional.

Un **manejador de excepciones** es un bloque de código encargado de tratar las excepciones para intentar recuperarse del fallo y evitar que la excepción sea lanzada descontroladamente hasta el *main* y termine el programa.

El formato de este bloque es el siguiente:

```
try {  
    Instrucciones que se pretenden ejecutar. Si se produce una excepción se abandona dicho bloque (no  
    se ejecutarán las demás instrucciones del try) y se saltará al bloque catch. Lógicamente, si en el try no  
    se produce ninguna excepción el bloque catch se ignora.  
} catch (TipoExcepcion nombre Variable){  
    Instrucciones que se van a ejecutar cuando se produce una excepción.  
    Capturará las excepciones del tipo TipoExcepción. Aquí deberemos escribir las instrucciones que sean  
    necesarias para manejar el error. Pueden especificarse varios bloques catch para distintos tipos de  
    excepciones.  
} finally {  
    Instrucciones que se van a ejecutar tanto si se producen excepciones como si no se producen. Es  
    opcional.  
}
```

3.1.-BLOQUE CATCH

Se pueden especificar varios **catch** (al menos uno) para controlar diferentes excepciones. En cada *catch* se dispone de un objeto de la excepción producida (este objeto está identificado en el paréntesis siguiente a la palabra *catch*, tras el tipo de clase). Este objeto ofrece métodos con información de la excepción. Cada tipo de excepción tiene sus propios métodos, aunque hay un par de métodos que tienen todos los tipos de error:

- ***getMessage()*** → Mensaje asociado al error. Es un mensaje descriptivo.
- ***printStackTrace()*** → Pila de llamadas del error. Es la descripción de la pila de llamadas de métodos y clases que ha dado lugar al error, con alguna información añadida. Es la misma salida de error que se ve normalmente al dar error en un programa sin control de excepciones (si no se controlan excepciones, JVM ejecuta este método justo antes de acabar el programa)

Cuando hay muchos *catch*, una vez producido el error, el control del programa no va a cualquier *catch*, sino que se mira cada *catch* ordenadamente de arriba hacia abajo hasta encontrar

un *catch* que satisfaga el error provocado. Es decir, no se va directamente la *catch* más idóneo, sino que entra en el primero que encuentra que le valga el error. Por lo tanto, solo se accede a un bloque *catch*, aunque haya muchos que satisfagan la excepción generada. Esto provoca un problema con la clasificación de las excepciones, pues no son todas hijas directas de *RuntimeException* o de *Exception*, sino que tienen un árbol de herencia mas clasificado entre ellas. Por ejemplo, ***EOFException*** es hija de ***IOException***. Por eso, no se debe añadir una cláusula *catch* de ***IOException*** antes de una de ***EOFException***, ya que ésta jamás se ejecutará.

Puede introducirse un *catch* genérico, para cualquier situación no controlada por los otros *catch*. Realmente esto quiere decir que se podría poner un único *catch* con esta solución, aunque no se podría especificar comportamientos diferentes para cada excepción.

```
catch( Exception ex){  
    ...  
}
```

El *catch (Exception e)* es obligatorio que se ponga en el último bloque *catch*. Debería tener siempre al menos un *printStackTrace()*, para mostrar la pila de llamadas y así poder identificar más fácilmente el error producido y por qué se ha producido.

Desde la versión Java 7 se pueden recoger múltiples excepciones en el mismo bloque *catch*, usando el operador separador "|".

```
catch( IOException | SQLException ex){ // El objeto ex es el mismo para las dos excepciones  
    ex.printStackTrace();  
}
```

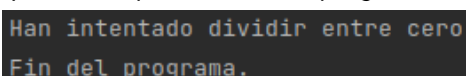
En estos casos, si se desea saber dentro del *catch* cuál de las dos excepciones se produjo, basta con usar el objeto que se recibe en el *catch*:

```
catch( IOException | SQLException ex){ // El objeto ex es el mismo para las dos excepciones  
    System.out.println(ex.getClass().getName());  
}
```

- **Ejemplo:** Vamos a producir una excepción y tratarla haciendo uso de los manejadores *try-catch*:

```
int x = 3, y = 0, div;  
try{  
    div = x/y;  
    System.out.println("La ejecución no llegará aquí");  
}catch (ArithmeticException ex){  
    System.out.println("Han intentado dividir entre cero");  
}  
System.out.println("Fin del programa.");
```

Al intenta dividir por cero se lanza automáticamente una *ArithmeticException*. Como esto sucede dentro del bloque *try*, la ejecución del programa pasa al primer bloque *catch* porque coincide con el tipo de excepción producida (*ArithmeticException*). Se ejecutará el código del bloque *catch* y luego el programa continuará con normalidad.



```
Han intentado dividir entre cero  
Fin del programa.
```

- Ejemplo: Siguiendo con el programa que calcula la media de dos números, vamos a introducir un bloque *try - catch - finally* para que el programa termine de forma controlada si se produce una excepción.

```
Scanner s = new Scanner(System.in);
System.out.println("Este programa calcula la media de dos números");
try {
    System.out.print("Introduzca el primer número: ");
    double numero1 = Double.parseDouble(s.nextLine());
    System.out.print("Introduzca el segundo número: ");
    double numero2 = Double.parseDouble(s.nextLine());
    System.out.println("La media es " + (numero1 + numero2) / 2);
} catch (Exception e) {
    System.out.print("No se puede calcular la media. ");
    System.out.println("Los datos introducidos no son correctos.");
} finally {
    System.out.println("Gracias por utilizar este programa ¡hasta la próxima!");
}
```

Ahora, si el usuario introduce un dato incorrecto, el programa termina de forma ordenada.

Este programa calcula la media de dos números

Introduzca el primer número: 23er

No se puede calcular la media. Los datos introducidos no son correctos.

Gracias por utilizar este programa ¡hasta la próxima!

Se puede mostrar tanto el tipo de excepción como el error exacto que se produce. Para ello, se aplican los métodos ***getClass()*** y ***getMessage()*** respectivamente al objeto ***e***. El tipo de excepción viene dado por el nombre de una clase que es subclase de ***Exception***. Bastará con añadir las siguientes líneas al ejemplo anterior.

```
System.out.println("Excepción: " + e.getClass());
System.out.println("Error: " + e.getMessage());
```

- Ejemplo: Siguiendo con el ejemplo anterior, si el usuario introduce un dato incorrecto, el programa mostrará la información adicional sobre la excepción que se produce.

```
Scanner s = new Scanner(System.in);
System.out.println("Este programa calcula la media de dos números");
try {
    System.out.print("Introduzca el primer número: ");
    double numero1 = Double.parseDouble(s.nextLine());
    System.out.print("Introduzca el segundo número: ");
    double numero2 = Double.parseDouble(s.nextLine());
    System.out.println("La media es " + (numero1 + numero2) / 2);
}
```

```
}catch (Exception e){
    System.out.println("Excepción: " + e.getClass());
    System.out.println("Error: " + e.getMessage());
    System.out.print("No se puede calcular la media. ");
    System.out.println("Los datos introducidos no son correctos.");
}finally {
    System.out.println("Gracias por utilizar este programa ¡hasta la próxima!");
}
```

El resultado será:

Este programa calcula la media de dos números

Introduzca el primer número: wer

Excepción: class java.lang.NumberFormatException

Error: For input string: "wer"

No se puede calcular la media. Los datos introducidos no son correctos.

Gracias por utilizar este programa ¡hasta la próxima!

Si queremos refinar un poco más el programa, si se produce una excepción al introducir un dato, el programa volverá a pedirlo una y otra vez hasta que el dato sea correcto.

```
Scanner s = new Scanner(System.in);
System.out.println("Este programa calcula la media de dos números");
boolean datoValido = false;
double numero1 = 0;
do {
    try {
        System.out.print("Introduzca el primer número: ");
        numero1 = Double.parseDouble(s.nextLine());
        datoValido = true;
    } catch (Exception e) {
        System.out.print("El dato introducido no es correcto, debe ser un número.");
        System.out.println(" Por favor, inténtelo de nuevo.");
    }
} while (!datoValido);

double numero2 = 0;
do {
    try {
        datoValido = false;
        System.out.print("Introduzca el segundo número: ");
        numero2 = Double.parseDouble(s.nextLine());
        datoValido = true;
    } catch (Exception e) {
        System.out.print("El dato introducido no es correcto, debe ser un número.");
        System.out.println(" Por favor, inténtelo de nuevo.");
    }
}
```

```
    }  
} while (!datoValido);
```

```
System.out.println("La media es " + (numero1 + numero2) / 2);
```

3.2.-BLOQUE *FINALLY*

Ha de ir detrás del último *catch*. Se ejecuta siempre, haya habido error o no, tras cualquier *catch* o tras el final del cuerpo normal del programa, e incluso cuando hay un error no controlado en los *catch*, antes de propagar la excepción. Aparentemente es igual que meter instrucciones final del código, sin usar el *finally*, pero no es así, pues podemos meter un *return* en un *catch*. En este caso, se ejecuta el *catch* hasta el *return*, no lo hacen las instrucciones finales tras los *catch*, pero si el *finally*.

- Ejemplo:

```
public static void main(String[] args) {  
    Scanner entrada = new Scanner(System.in);  
    int numero;  
    System.out.println("escribe un numero entero...");  
    try {  
        numero = entrada.nextInt();  
        int cuadrado = numero * numero;  
        System.out.println("El cuadrado de " + numero + " es" + cuadrado);  
    } catch (InputMismatchException ex) {  
        System.out.println("Error, no se introdujo un numero");  
    } catch (NumberFormatException ex) {  
        System.out.println("Error, el numero tiene formato erroneo");  
        return;  
    } catch (Exception ex) {  
        System.out.println("Error general, no especifico");  
        return;  
    } finally {  
        System.out.println("PASA SIEMPRE por aqui");  
    }  
    System.out.println("Pasa por aqui si no hay error o el error es InputMismatchException");  
}
```

4.- CONTROL DE VARIOS TIPOS DE EXCEPCIONES

La clase ***Exception*** hace referencia a una excepción genérica. Existen muchas subclases de ella como ***DataFormatException*** , ***IOException***, ***IndexOutOfBoundsException***, etc. Mediante la utilización de varios ***catch*** con diferentes subclases de ***Exception*** se pueden discriminar distintas excepciones.

Por eso, es importante entender que el bloque ***catch*** solo capturará excepciones del tipo indicado. Si se produce una excepción distinta no la capturará. Sin embargo capturará excepciones heredadas del tipo indicado. Por ejemplo, ***catch (ArithmeticException e)*** capturará cualquier tipo de excepción que herede de ***ArithmeticException***. El caso más general es ***catch (Exception e)***

que capturará todo tipo de excepciones porque en Java todas las excepciones heredan de **Exception**.

Sin embargo, es mejor utilizar excepciones lo más cercanas al tipo de error previsto, ya que lo que se pretende es recuperar el programa de alguna condición de error y si "se meten todas las excepciones en el mismo saco", seguramente habrá que averiguar después qué condición de error se produjo para poder dar una respuesta adecuada.

- Ejemplo: Realiza un programa (**PintarLineasAsteriscosVersion1.java**) que pida el número total de asteriscos y el número de líneas que se quieren pintar. Por ejemplo, si el usuario dice que quiere pintar 10 asteriscos y 3 líneas, el programa pintará dos líneas de 4 asteriscos más una línea de 2 asteriscos.

Cuando el usuario introduce los datos correctamente, es decir, dos números enteros, no hay ningún problema.

Este programa pinta varias líneas de asteriscos

Introduzca el número total de asteriscos: 13

Introduzca el número de líneas que quiere pintar: 4

Línea 1: ****

Línea 2: ****

Línea 3: ****

Línea 4: *

Ahora bien, si el usuario introduce un número con decimales en lugar de un número entero, se produce la excepción **NumberFormatException** como se muestra a continuación.

Este programa pinta varias líneas de asteriscos

Introduzca el número total de asteriscos: 20.75

Exception in thread "main" java.lang.NumberFormatException: For input string: "20.75"

at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)

at java.base/java.lang.Integer.parseInt(Integer.java:668)

at java.base/java.lang.Integer.parseInt(Integer.java:786)

at excepciones.PintarLineasAsteriscosVersion1.start(PintarLineasAsteriscosVersion1.java:14)

at excepciones.PintarLineasAsteriscosVersion1.main(PintarLineasAsteriscosVersion1.java:8)

La línea que ha producido el fallo será donde se realiza la lectura del número de asteriscos. Por ejemplo podría ser la siguiente instrucción:

```
int asteriscos = Integer.parseInt(s.nextLine());
```

Otro error que se puede producir es si el usuario introduce un 0 cuando el programa le pregunta cuántas líneas quiere pintar. En este caso salta la excepción **ArithmeticException** ya que el programa intenta realizar una división con el divisor igual a 0.

Este programa pinta varias líneas de asteriscos

Introduzca el número total de asteriscos: 13

Introduzca el número de líneas que quiere pintar: 0

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

at excepciones.PintarLineasAsteriscosVersion1.start(PintarLineasAsteriscosVersion1.java:18)

at excepciones.PintarLineasAsteriscosVersion1.main(PintarLineasAsteriscosVersion1.java:8)

La línea que ha provocado el error es aquella que calcula la cantidad de asteriscos que va a tener cada línea. Como la lectura de la línea es 0, entonces se produce una división por cero.

Por ejemplo, podría ser el siguiente código:

```
if((asteriscos % lineas) == 0 )
    longitud = asteriscos / lineas;
else
    longitud = (int)Math.ceil((double)asteriscos / lineas);
```

- Mejora el programa (**PintarLineasAsteriscosVersion2.java**) que has realizado utilizando las excepciones que controlen los errores de ejecución que se producen en la primera versión del programa (***NumberFormatException*** y ***ArithmeticException***). Comprueba que si el usuario introduce un número con decimales o una palabra en lugar de un número entero, o un número de líneas 0, no hay ningún problema, el programa no se rompe y termina de forma ordenada. Un ejemplo de esta versión mejorada puede ser:

Este programa pinta varias líneas de asteriscos

Introduzca el número total de asteriscos: w2

Los datos introducidos no son correctos.

Se deben introducir números enteros.

Este programa pinta varias líneas de asteriscos

Introduzca el número total de asteriscos: 23

Introduzca el número de líneas que quiere pintar: 0

El número de líneas no puede ser 0.

4.1.-CLÁUSULAS *CATCH* MÚLTIPLES

Se pueden especificar varias cláusulas *catch*, tantas como queramos, para que cada una capture un tipo diferente de excepción. Su formato sería:

```
try {
    // instrucciones que pueden producir distintos tipos de Excepciones
}
catch (TipoExcepción1 e1) {
    // instrucciones para manejar un TipoExcepción1
}
catch (TipoExcepción2 e2) {
    // instrucciones para manejar un TipoExcepción2
}
...
```

```

}
catch (TipoExcepciónN eN) {
    // instrucciones para manejar un TipoExcepciónN
}
finally { // opcional
    // instrucciones que se ejecutarán tanto si hay excepción como si no
}

```

Cuando se lanza una excepción dentro del **try**, se comprueba cada sentencia **catch** en orden y se ejecuta la primera cuyo tipo coincida con la excepción lanzada. Los demás bloques **catch** serán ignorados. Luego se ejecutará el bloque **finally** (si se ha definido) y el programa continuará su ejecución después del bloque **try-catch-finally**.

Si el tipo excepción producida no coincide con ninguno de los **catch**, entonces la excepción será lanzada al método que nos llamó.

- Ejemplo:

```

Scanner entrada = new Scanner(System.in);
int numero;
System.out.println("escribe un numero entero...");
try{
    numero = entrada.nextInt();
    int cuadrado = numero*numero;
    System.out.println("El cuadrado de "+numero+ " es"+ cuadrado);
}catch(InputMismatchException ex){
    System.out.println("Error, no se introdujo un numero");
}catch(NumberFormatException ex){
    System.out.println("Error, el numero tiene formato erroneo");
}finally{
    System.out.println("Salida siempre por aqui");
}

```

- Ejemplo: Vamos a ejecutar distintas instrucciones dentro del bloque **try** y manejaremos las excepciones de división por cero y de si sobrepasamos el tamaño del vector.

```

int x, y, div, pos;
int v[] = {1,2,3};
Scanner in = new Scanner(System.in);
try{
    System.out.print("Numerador: ");
    x = in.nextInt();
    System.out.print("Denominador: ");
    y = in.nextInt();
    div = x/y;
    System.out.println("División: "+div);
    System.out.print("Posición vector a consultar: ");
    pos = in.nextInt();
}

```

Pueden suceder tres cosas diferentes:

1. El **try** se ejecuta sin excepciones, se ignoran los **catch** y se imprime "Fin del programa".

```

Numerador: 4
Denominador: 2
División: 2
Posición vector a consultar: 1
Elemento: 1
Fin del programa.

```

2. Se produce la excepción de división por cero, el flujo de ejecución salta al 1er **catch**, se imprime el mensaje "División por cero..." y luego "Fin del programa".

```

Numerador: 8
Denominador: 0
División por cero: java.lang.ArithmeticException: / by zero
Fin del programa.

```

```

        System.out.println("Elemento: "+v[pos]);
    }catch (ArithmeticException ex){
        System.out.println("División por cero: "+ex);
    }catch (ArrayIndexOutOfBoundsException ex){
        System.out.println("Sobrepasado tamaño vector: "+ex);
    }finally {
        System.out.println("Fin del programa.");
    }
}

```

3. Se produce la excepción de sobrepasar el vector , el flujo de ejecución salta al 2º *catch*, se imprime el mensaje "Sobrepasado el tamaño del vector..." y "Fin del programa".

```

Numerador: 8
Denominador: 4
División: 4
Posición vector a consultar: 5
Sobrepasado tamaño vector: java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3
Fin del programa.

```

4.2.-OBJETO EXCEPTION

Toda excepción genera un objeto de la clase **Exception** (o uno más específico que hereda de *Exception*). Dicho objeto contendrá detalles sobre el error producido. Puede ser interesante mostrar esta información para que la vea el usuario (que sepa qué ha sucedido) o el desarrollador (para depurar y corregir el código si es pertinente). En la cláusula *catch* tenemos acceso al objeto en caso de que queramos utilizarlo:

Los dos métodos de **Exception** más útiles son (los hemos visto anteriormente):

- **getMessage()** → Devuelve un String con un texto simple sobre el error.
- **printStackTrace()** → Es el que más información proporciona. Indica qué tipo de Excepción se ha producido, el mensaje simple, y también toda la pila de llamadas. Esto es lo que hace Java por defecto cuando una excepción no se maneja y acaba parando el programa.

```

...
catch (Exception e){

    // Mostramos el mensaje de la excepción
    System.err.println("Error: " + e.getMessage());

    // Mostramos toda la información, mensaje y pila de llamadas
    e.printStackTrace();
}
...

```

Los objetos de tipo **Exception** tienen sobrecargado el método **toString()** por lo que también es posible imprimirlos directamente mediante **println()**.

```

...
catch (Exception e){

```



```
// Mostramos el mensaje de la excepción
System.out.println(e);

}

...
```

5.- LANZAR EXCEPCIONES (*THROW*)

5.1.-¿POR QUÉ LANZAR EXCEPCIONES?

Un programador puede programar su código de forma que se lancen excepciones cuando se intente hacer algo incorrecto o inesperado (en ocasiones es recomendable). Por ejemplo, cuando los argumentos que se le pasan a un método no son correctos o no cumplen ciertos criterios.

Esto es habitual en la Programación Orientada a Objetos (POO): Recordad que una clase debe ser la responsable de la lógica de sus objetos: asegurar que los datos sean válidos, además de controlar qué está permitido y no está permitido hacer. **Por ejemplo si se instancia una clase con valores incorrectos** como un objeto Persona con un DNI no válido, una edad negativa, una cuenta bancaria con saldo negativo, etc. En esos casos **es conveniente que el constructor lance una excepción**.

También puede ser apropiado lanzar excepciones en los *setters* si el valor no es válido, y en cualquier otro método en el que se intente hacer algo no permitido o que viole la integridad del objeto como por ejemplo retirar dinero de una cuenta sin saldo suficiente.

Las excepciones pueden manejarse y controlarse sin que el programa se pare. Es decir, lanzar una excepción no implica necesariamente que el programa terminará, simplemente es una forma de avisar de un error. **Quien llame al método es responsable de manejar la excepción para que el programa no se pare.**

5.2.-CÓMO LANZAR UN EXCEPCIÓN

Para lanzar la excepción se utiliza la palabra reservada **throw** seguido de un objeto de tipo **Exception** (o alguna de sus subclases como **ArithmeticException**, **NumberFormatException**, **ArrayIndexOutOfBoundsException**, etc.). Como las excepciones son objetos, deben instanciarse con **new**. Por lo tanto, podemos lanzar una excepción genérica así:

```
throw new Exception();
```

Esto es equivalente a primero instanciar el objeto *Exception* y luego lanzarlo:

```
Exception e = new Exception();  
throw e;
```

El constructor de **Exception** permite (opcionalmente) un argumento **String** para dar detalles sobre el problema. Si la excepción no se maneja y el programa se para, el mensaje de error se mostrará por la consola (esto es muy útil para depurar programas).

```
throw new Exception("La edad no puede ser negativa");
```

En lugar de lanzar excepciones propias de Java (**ArithmeticException**, **NumberFormatException**, **ArrayIndexOutOfBoundsException**, etc.) normalmente es preferible lanzar excepciones genéricas **Exception** o mejor aún, utilizar nuestras propias excepciones.

La orden **throw** permite lanzar de forma explícita una excepción.

- Ejemplo: La siguiente sentencia **throw new ArithmeticException()** crea de forma artificial una excepción igual que si existiera una línea como **System.out.println(1 / 0);**.

```
System.out.println("Inicio");  
throw new ArithmeticException();
```

```
Inicio  
Exception in thread "main" java.lang.ArithmeticException: Create breakpoint  
    at excepciones.LanzamientoExcepcionThrow.start(LanzamientoExcepcionThrow.java:10)  
    at excepciones.LanzamientoExcepcionThrow.main(LanzamientoExcepcionThrow.java:6)
```

De la misma manera, el programa que se muestra a continuación genera también una excepción del tipo **ArithmeticException()**.

```
System.out.println("Inicio");  
System.out.println(1 / 0);
```

```
Inicio  
Exception in thread "main" java.lang.ArithmeticException: Create breakpoint : / by zero  
    at excepciones.LanzamientoExcepcionThrow.start(LanzamientoExcepcionThrow.java:10)  
    at excepciones.LanzamientoExcepcionThrow.main(LanzamientoExcepcionThrow.java:6)
```

Hay que tener en cuenta que **throw** solo puede lanzar excepciones que pertenezcan a la clase **Throwable**. Como **throw** permite lanzar de forma explícita una excepción, nos servirá para lanzar excepciones propias como veremos más adelante. También es útil cuando se recoge la excepción en un método y luego, esa misma excepción se vuelve a lanzar para que la recoja, a su vez, otro método y luego otro y así sucesivamente hasta llegar al main.

- Ejemplo: Vamos a ver cómo se recoge y se trata una excepción dentro de una función y, además es la propia función la que “pasa la bola” al main. Partimos del siguiente ejemplo sin

control de excepciones (**ExcepcionesManzana.java**).

```
void start(){
    Scanner s = new Scanner(System.in);
    System.out.print("Número de manzanas: ");
    int m = Integer.parseInt(s.nextLine());
    System.out.print("Número de personas: ");
    int p = Integer.parseInt(s.nextLine());
    System.out.print("A cada persona le corresponden " + reparteManzanas(m, p) + " manzanas.");
}
public static int reparteManzanas(int manzanas, int personas) {
    return manzanas / personas;
}
```

Con los datos introducidos correctamente, el programa realiza su cometido:

Número de manzanas: 10

Número de personas: 2

A cada persona le corresponden 5 manzanas.

Pero cuando el usuario introduce un 0 como número de personas, el programa provoca una excepción:

Número de manzanas: 10

Número de personas: 0

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at excepciones.ExcepcionesManzanas.reparteManzanas(ExcepcionesManzanas.java:19)
    at excepciones.ExcepcionesManzanas.start(ExcepcionesManzanas.java:16)
    at excepciones.ExcepcionesManzanas.main(ExcepcionesManzanas.java:8)
```

Vamos a mejorar el ejemplo para llevar un control de las excepciones tanto en la función como en el **main**:

```
void start(){
    Scanner s = new Scanner(System.in);
    System.out.print("Número de manzanas: ");
    int m = Integer.parseInt(s.nextLine());
    System.out.print("Número de personas: ");
    int p = Integer.parseInt(s.nextLine());
    try {
        System.out.print("A cada persona le corresponden " + reparteManzanas(m, p) + " manzanas.");
    } catch (ArithmeticException ae){
        System.out.println("Los datos introducidos no son correctos.");
    }
}
public static int reparteManzanas(int manzanas, int personas) {
    try {
```

```
        return manzanas / personas;
    }catch (ArithmeticException ae){
        System.out.println("El número de personas vale 0.");
        throw ae;
    }
}
```

Si ahora introducimos un 0 como número de personas, el programa provoca una excepción que es recogida y lanzada desde la función **reparteManzanas** a la función **main**:

Número de manzanas: 10

Número de personas: 0

El número de personas vale 0.

Los datos introducidos no son correctos.

6.- DECLARACIÓN DE UN MÉTODO CON THROWS

Hemos visto en el apartado anterior cómo lanzar una excepción desde una función/método con **throw**. Es muy recomendable indicar de forma explícita en la cabecera que existe esa posibilidad, es decir, que el método en cuestión puede provocar una excepción. Se trata de una declaración de intenciones, algo parecido al `@Override`. Si el IDE que estemos utilizando detecta que un método tiene **throws** en su cabecera y, sin embargo, no hemos gestionado bien la posibilidad de provocar una excepción, nos avisará con el correspondiente mensaje de error.

- Ejemplo: Vamos a modificar la cabecera de la función `reparteManzanas()` para incluir **throws**. Lo grabaremos como **ExcepcionesManzanaConThrows.java**

```
public static int reparteManzanas(int manzanas, int personas) throws ArithmeticException {
    try {
        return manzanas / personas;
    }catch (ArithmeticException ae){
        System.out.println("El número de personas vale 0.");
        throw ae;
    }
}
```

También se podría definir de la siguiente manera:

```
public static int reparteManzanas(int manzanas, int personas) throws ArithmeticException {
    return manzanas / personas; //Si personas vale cero lanzará esa excepción
}
```

- Ejemplo: Veamos el método **setEdad(int edad)** de la clase Persona. Como hemos decidido que la edad de una persona no puede ser negativa, lanzaremos una excepción si $edad < 0$.

```
// Setter del atributo edad. Debe ser >= 0
public void setEdad(int edad) throws Exception {
    if (edad < 0)
        throw new Exception("Edad negativa no permitida");
    else
        this.edad = edad;
}
```

Al lanzar una excepción se parará la ejecución de dicho método (no se ejecutará el resto del código del método) y se lanzará la excepción al método que lo llamó. Si por ejemplo desde la función *main* llamamos a *setEdad()*, como puede suceder que *setEdad()* lance una excepción, entonces en la práctica es posible que el *main* lance una excepción (no directamente con un *throw*, sino por la excepción que nos lanza *setEdad()*). Por lo tanto, también tenemos que especificar en el *main* que se puede lanzar una excepción:

```
public static void main(String[] args) throws Exception {
    Persona p = new Persona("44193900L", "Pepito", 27);
    ...
    ...
    p.setEdad(valor); // Puede lanzar una excepción
}
```

- Ejemplo: También podemos lanzar diferentes tipos de excepciones.

```
public static void main(String[] args) {
    try {
        cuadrado();
    } catch (InputMismatchException ex) {
        System.out.println("Error, no se introdujo un numero");
    } catch (NumberFormatException ex) {
        System.out.println("Error, el numero tiene formato erroneo");
    } catch (Exception ex) {
        System.out.println("Error general, no especifico");
    }
}

public static void cuadrado() throws InputMismatchException, NumberFormatException {
    Scanner entrada = new Scanner(System.in);
    int numero;
    System.out.println("escribe un numero entero...");
    numero = entrada.nextInt();
    int cuadrado = numero * numero;
    System.out.println("El cuadrado de " + numero + " es " + cuadrado);
}
```

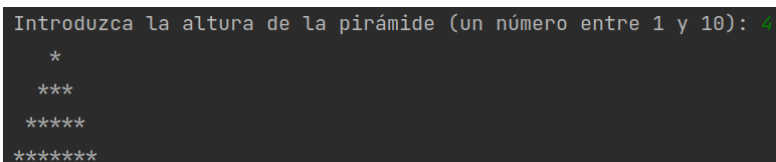
7.- CREACIÓN DE EXCEPCIONES PROPIAS O PERSONALIZADAS

Java nos permite crear excepciones propias, hechas a medida. Para ello, no hay más que utilizar una de las características más importantes de la programación orientada a objetos: la herencia. Crear una nueva excepción será tan sencillo como implementar una subclase de

Exception.

- Ejemplo: Tenemos un programa que pinta por pantalla una pirámide, tras haber pedido la altura a un usuario. En principio, el contenido del **main** sería el que se muestra a continuación.

```
void start(){
    Scanner s = new Scanner(System.in);
    System.out.print("Introduzca la altura de la pirámide (un
número entre 1 y 10): ");
    int h = Integer.parseInt(s.nextLine());
    pintaPiramide(h);
}
```



```
Introduzca la altura de la pirámide (un número entre 1 y 10): 7
*
***
*****
*****
*****
*****
*****
```

```
public static void pintaPiramide(int h) {
    int planta = 1;
    int longitudDeLinea = 1;
    int espacios = h - 1;
    while (planta <= h) {
        // inserta espacios
        for (int i = 1; i <= espacios; i++) {
            System.out.print(" ");
        }
        // pinta la línea
        for (int i = 1; i <= longitudDeLinea; i++) {
            System.out.print("*");
        }

        System.out.println();
        planta++;
        espacios--;
        longitudDeLinea += 2;
    }
}
```

Vamos a crear una nueva excepción llamada **ExcepcionAlturaFueraDeRango** de tal forma que si alguien intenta pintar una pirámide con una altura menor que 1 o mayor que 10, salte el error y se pueda tratar con un bloque **try - catch**. Será algo parecido al **IndexOutOfBoundsException** que se produce cuando intentamos acceder a una posición que no existe dentro de un *array*.

Crearemos una clase que se llame precisamente **ExcepcionAlturaFueraDeRango**.

```
public class ExcepcionAlturaFueraDeRango extends Exception {
    public ExcepcionAlturaFueraDeRango() {
        System.out.println("ExcepcionAlturaFueraDeRango: La altura está fuera del rango permitido.");
    }
}
```

Ahora tenemos que utilizarla desde nuestra función *pintaPiramide()* y desde el programa principal. Si queremos que la función *pintaPiramide()* pueda arrojar la excepción **ExcepcionAlturaFueraDeRango**, lo tendremos que indicar en la cabecera.

```
public static void pintaPiramide(int h) throws ExcepcionAlturaFueraDeRango{
    ...
}
```

Dentro de la función *pintaPiramide()*, se comprueba si la altura pasada como parámetro está fuera del rango permitido. En tal caso, se lanza la excepción con **throw**. A continuación tratamos la posible excepción desde el programa principal

```
void start(){
    ...
    try {
        pintaPiramide(h);
    } catch (ExcepcionAlturaFueraDeRango eafR) {
        System.out.println("No se ha podido pintar la pirámide.");
    }
}
```

```
Introduzca la altura de la pirámide (un número entre 1 y 10): 23
ExcepcionAlturaFueraDeRango: La altura está fuera del rango permitido.
No se ha podido pintar la pirámide.
```

```
public static void pintaPiramide(int h) throws ExcepcionAlturaFueraDeRango{
    if ((h < 1) || (h > 10)) {
        throw new ExcepcionAlturaFueraDeRango();
    }
    ...
}
```

Se pueden crear excepciones propias de usuario, que nos sirvan para enviar información y establecer comunicación de eventos entre diversas partes de un proyecto. Por lo que podemos crear errores que no son del propio Java, sino que pertenecen a reglas de negocio de nuestra propia aplicación.

- Ejemplo: Tenemos la siguiente clase Coche y su prueba .

```
public class Coche {
    private String marca;
    private String modelo;
    private int velocidad;

    public Coche(String marca, String modelo) {
        this.marca = marca;
        this.modelo = modelo;
        this.velocidad = 0;
    }

    public void acelerar(int cuanto) {
        this.velocidad = this.velocidad + cuanto;
    }
}
```

```
public class PruebaCoches {

    public static void main(String[] args) {

        Coche c = new Coche("OPEL", "INSIGNIA");
        c.acelerar(180);
        ...
    }
}
```

Imaginemos que en la clase Coche tenemos una restricción de que el coche no puede tener una velocidad superior a 120 km/h. Para que la clase prueba se entere de que hay un problema de velocidad, el método acelerar deberá de controlar si se produce el problema e informar a la clase que ha llamado al método.

La solución consiste en construir un tipo de error propio y provocar el error de modo que el *main* de la clase *PruebaCoches* pueda controlarlo con un *try-catch*.

Crearemos una clase propia que herede de alguna clase excepción que ya exista en Java : **Exception** o **RuntimeException**. La diferencia es que si hereda de *Exception*, nuestro error será del tipo *checked* (comprobado por el compilador). Si heredamos de *RuntimeException* será *unchecked* (no comprobado por el compilador). Elegiremos siempre *Exception*, y así nos aseguramos de que no nos dejamos nada sin hacer, pues el compilador nos avisará. Creamos la excepción **ExcesoVelocidadException** que nos permita controlar el problema:

```
public class ExcesoVelocidadException extends Exception {  
    . . .  
}
```

El constructor de mi clase **ExcesoVelocidadException** ha de llamar al constructor de la clase *Exception*. *Exception* tiene varios constructores, pero vamos a usar por ahora el mas simple, el que recibe un String por parámetro (mirar API de Java), que será el mensaje de la excepción a crear.

```
public class ExcesoVelocidadException extends Exception {  
    public ExcesoVelocidadException () {  
        super (" ¡Cuidado, error! Velocidad máxima alcanzada! ");  
    }  
}
```

El paso siguiente es descubrir dónde se puede producir mi problema y añadir el código que compruebe si se ha producido. En nuestro ejemplo, el problema se puede producir en el método *acelerar()*, cuando la velocidad supere los 120, así que ahí es donde añadiremos el código de control.

```
public void acelerar(int cuanto) {  
    this.velocidad = this.velocidad + cuanto;  
    if(this.velocidad>120){  
        // AQUI OCURRE MI PROBLEMA.....  
    }  
}
```

Para dar solución al problema, lo primero que haremos es crear un objeto de mi tipo de error personalizado.

```
public void acelerar(int cuanto) {  
    this.velocidad = this.velocidad + cuanto;  
    if(this.velocidad>120){  
        ExcesoVelocidadException mierror = new ExcesoVelocidadException();  
    }  
}
```

A continuación provocamos el error, es decir, vamos a hacer que nuestro error ocurra (igual que ocurre con otros errores propios de Java). Para ello usamos la instrucción *throw* junto con el objeto de nuestro error.


```
public void acelerar(int cuanto) {
    this.velocidad = this.velocidad + cuanto;
    if(this.velocidad>120){
        ExcesoVelocidadException mierror = new ExcesoVelocidadException();
        throw mierror;
    }
}
```

Ahora nos toca pensar que hacemos con el error que se acaba de producir. Como cualquier otro error, podemos tratarlo con un **try-catch** o propagarla. Tratarlo con un **try-catch** no tiene ningún sentido, pues para eso, no habríamos ni declarado ni provocado la excepción, hubiéramos controlado la circunstancia directamente. Además, nuestra intención es que el error se controle en el *main* que llama a `acelerar()`, así que optamos por la segunda opción: vamos a propagar nuestro error, con la instrucción **throw**:

```
public void acelerar(int cuanto) throws ExcesoVelocidadException {
    this.velocidad = this.velocidad + cuanto;
    if(this.velocidad>120){
        ExcesoVelocidadException mierror = new ExcesoVelocidadException();
        throw mierror;
    }
}
```

Finalmente, lo que resta es hacer el control del error en el método que llamó a `acelerar()`, que además era nuestra idea inicial. Para ello pondremos la llamada `acelerar()` dentro de un **try-catch** que capture y procese el posible error **ExcesoVelocidadException**. El error procesado en el *catch* tiene un método llamado **getMessage()** que devuelve el mensaje del error... que es aquel que se introdujo en el constructor.

```
public class Coche {
    private String marca;
    private String modelo;
    private int velocidad;

    public Coche(String marca, String modelo) {
        this.marca = marca;
        this.modelo = modelo;
        this.velocidad = 0;
    }

    public void acelerar(int cuanto) throws ExcesoVelocidadException {
        this.velocidad = this.velocidad + cuanto;
        if(this.velocidad>120){
            ExcesoVelocidadException mierror = new ExcesoVelocidadException();
            throw mierror;
        }
    }
}
```

```
public class AAZonaDePruebas {

    public static void main(String[] args) {

        Coche c = new Coche("OPEL", "INSIGNIA");
        try {
            c.acelerar(180);
        } catch (ExcesoVelocidadException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

- Ejemplo: También se puede crear un objeto de alguna excepción de las ya existentes en Java, por ejemplo, se puede crear un objeto directamente de **Exception**, y lanzarlo hacia algún método llamante, como hacíamos con las excepciones personalizadas:

```
public static int pedirDenominador() throws Exception{
    Scanner entrada = new Scanner(System.in);
    System.out.println("Dime el valor del denominador");
    int x = entrada.nextInt();
    if (x == 0) {
        Exception nfe = new Exception("El denominador no puede ser 0");
        throw nfe;
    }
    return x;
}

public static void main(String args[]) {
    try {
        int denom = pedirDenominador();
    } catch (Exception ex) {
        System.out.println(ex.getMessage());
    }
}
```

Igual que con **Exception**, se podría haber hecho un objeto de cualquier otra excepción (**NumberFormatException**, por ejemplo). Todas las excepciones permiten al construirse pasarles un *String* que será el mensaje que se use posteriormente en métodos como **getMessage()**.

- Ejemplo: Uso de un bucle de validación para simplificar la petición de datos con la clase *Scanner*. Controla con excepciones el caso de que el usuario no introduzca correctamente un número cuando se le pida.

```
public static int leerInt(String mensaje) {
    Scanner sc = new Scanner(System.in);
    boolean seguir = true;
    int numero = 0;
    while (seguir) {
        try {
            System.out.println(mensaje);
            numero = sc.nextInt();
            seguir = false;
        } catch (NumberFormatException ex) {
            System.out.println("Debe indicar un valor numerico entero");
        }
    }
    return numero;
}
```

- Ejemplo: Crea un método que analiza si un *String* contiene solo valores numéricos. Intenta convertir el *String* en un número, y usa el control de errores para evaluar si se produce un error o no en la conversión, es decir, si realmente es un número o no.

```
public static boolean esNumero(String strNum) {
    if (strNum == null) {
        return false;
    }
    try {
        double d = Double.parseDouble(strNum);
    } catch (NumberFormatException ex) {
        return false;
    }
    return true;
}
```

- Ejemplo: Método que recibe un *array*, donde se pide al usuario dos posiciones del mismo, y devuelve la división de los números que están en dichas posiciones del *array*. Se controlan en el método estas circunstancias: debe controlar que al pedir el valor por teclado, sea realmente un número; debe controlar que las posiciones del *array* existen y tienen valor; debe controlar que no se puede dividir por 0. Si se produce alguna de estas 3 circunstancias, el método devuelve 0.

```

public static double dividirArray(int[] array) {
    Scanner entrada = new Scanner(System.in);
    try{
        System.out.println("Dime dos posiciones del array:");
        int pos1 = entrada.nextInt();
        int pos2 = entrada.nextInt();
        double res = array[pos1]/array[pos2];
        return res;

    }catch(InputMismatchException ex){
        System.out.println("Se solicita un numero, no puede haber nada que no sean digitos");
    }catch(ArrayIndexOutOfBoundsException ex){
        System.out.println("No existe esa posicion en el array");
    }catch(ArithmeticException ex){
        System.out.println("No se pude dividir por cero");
    }
    return 0;
}

```

```

public static void main(String[] args) {
    // PROBANDO DIVIDIR ARRAY
    // =====
    int[] arr = {0,6,2,120,1,4,8,10};
    double res = dividirArray(arr);
    System.out.println(res);
}

```

• Ejemplo :

- Este ejercicio crea varios metodos sumar, restar, multiplicar y dividir, que reciben dos enteros por parametros, y devuelven la operacion correspondiente con esos dos numeros.
- Todos los métodos deben devolver una excepcion personalizada si alguno de los parámetros es negativo
- El método dividir debe además devolver una Exception (de la clase Java Exception) con mensaje personalizado, cuando el divisor sea un 0

```

// no hay por que lanzar las dos, con la segunda vale, pero ponemos ambas para ver que se usan ambas
public static int sumar(int a, int b) throws NumeroNoValidoException, Exception {
    if (a < 0 || b < 0) {
        NumeroNoValidoException n = new NumeroNoValidoException("no valen numeros negativos");
        throw n;
        //throw new NumeroNoValidoException("no valen numeros negativos");
    }
    return a + b;
}
public static int restar(int a, int b) throws NumeroNoValidoException, Exception {
    if (a < 0 || b < 0) {
        throw new NumeroNoValidoException("no valen numeros negativos");
    }
    return a - b;
}
public static int multiplicar(int a, int b) throws NumeroNoValidoException, Exception {
    if (a < 0 || b < 0) {
        throw new NumeroNoValidoException("no valen numeros negativos");
    }
    return a * b;
}
public static double dividir(double a, double b) throws NumeroNoValidoException, Exception {
    if (a < 0 || b < 0) {
        throw new NumeroNoValidoException("no valen numeros negativos");
    }
    if (b == 0) {
        Exception e = new Exception("El dividos no puede ser 0");
        throw e;
    }
    return a / b;
}

```

```

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

```

```

public static void main(String[] args) {
    int x = 4;
    int y = 8;
    try {
        double suma = sumar(x,y);
        double resta = restar(x,y);
        double multi = multiplicar(x,y);
        double divid = dividir(x,y);
        System.out.println("la suma es " + suma);
        System.out.println("la resta es " + resta);
        System.out.println("la multi es " + multi);
        System.out.println("la divi es " + divid);
    } catch ( NumeroNoValidoException ex) {
        System.out.println("Error en algun parámetro");
        System.out.println(ex.getMessage());
    } catch (Exception ex) {
        System.out.println("Error en algun sitio");
        System.out.println(ex.getMessage());
    }
}

```

- Ejemplo: Excepción llamada *ExcepcionPropia* que utilizaremos cuando se le pase al método un valor superior a 10.

El main y el método que lanza la excepción:

```
public static void main(String[] args) {
    try
    {
        metodo(1);
        metodo(20);
    }
    catch (ExcepcionPropia e)
    {
        System.out.println("capturada :" + e);
    }
}

static void metodo(int n) throws ExcepcionPropia
{
    System.out.println("Llamado por metodo(" + n + ")");

    if (n > 10)
        throw new ExcepcionPropia(n);

    System.out.println("Finalización normal");
}
```

La definición de la nueva excepción.

```
public class ExcepcionPropia extends Exception
{
    private int num;

    ExcepcionPropia(int n)
    {
        this.num = n;
    }
    public String toString()
    {
        return "Excepcion Propia[" + this.num + "]";
    }
}
```

Siendo la salida:

```
run:
Llamado por metodo(1)
Finalización normal
Llamado por metodo(20)
capturada :Excepcion Propia[20]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Como se puede observar la excepción se lanza cuando el número es mayor que 10 y será tratada por la nueva clase creada que hereda de `Exception`. Además se sobreescribe el método ***toString*** que es el encargado de mostrar el mensaje asociado a la excepción.

- Ejemplo: Lanzamiento de varios tipos de excepciones propias.

```
public Persona(String dni, int edad) throws InvalidDniException, InvalidEdadException {
    if (!dni.matches("[0-9]{8}[A-Z]")) {
        throw new InvalidDniException("DNI no válido: " + dni);
    }
    if (edad < 0) {
        throw new InvalidEdadException("Edad no válida: " + edad);
    }
    this.dni = dni;
    this.edad = edad;
}
```

8.- RECOMENDACIONES

Hemos visto cómo tratar diferentes tipos de excepciones para prevenir que un programa se rompa y termine de forma abrupta. No obstante, en términos generales, no se debe dejar en manos del control de excepciones lo que se puede hacer mediante sencillas comprobaciones previas.

Por ejemplo, antes de realizar una división, el programador puede utilizar un if para comprobar si el divisor es 0 y, en tal caso, derivar el flujo de ejecución convenientemente sin tener que recurrir a las excepciones.

Sucede algo muy parecido con la excepción `IndexOutOfBoundsException`. Si un programador maneja bien los índices de un *array*, este error nunca debería llegar a producirse.