IES CHAN DO MONTE
C.S. de Desarrollo de Aplicaciones Multiplataforma

Anexo 1: Excepciones

Índice

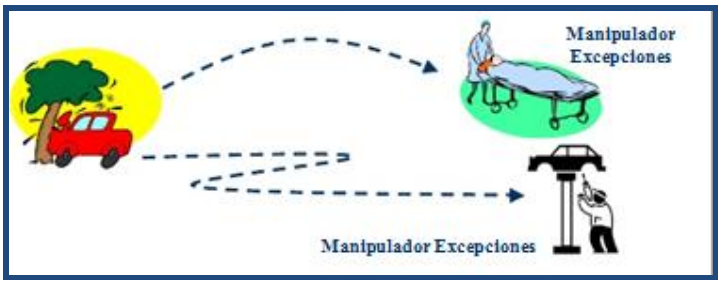
1. Excepciones

En Java una excepción es un error o una condición anormal que se ha producido durante la ejecución de un programa. Java tiene diferentes tipos de excepciones: excepciones de I/O, las excepciones en tiempo de ejecución y las de su propia creación.

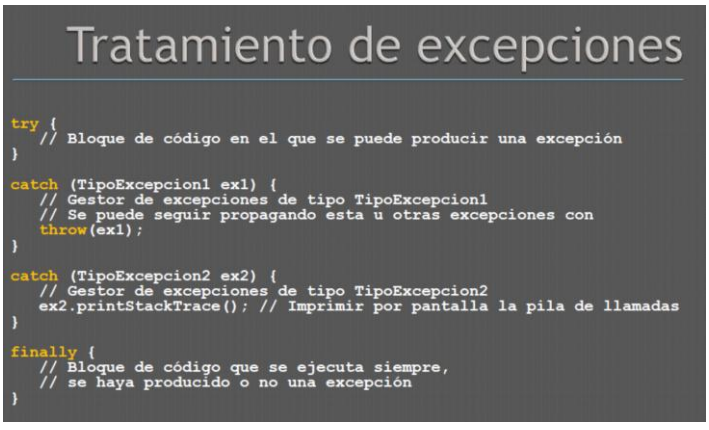
Durante la compilación solamente se detectan los errores de sintaxis, pero **el manejo de excepciones de Java permite el manipular los errores que ocurren en tiempo de ejecución**, entre estas podemos mencionar las excepciones aritméticas (ej.: división entre cero), excepciones de puntero (ej: acceso a punteros NULL), excepciones de indexación (ej.: acceso por encima o debajo de los límites de un vector), excepciones de entrada y salida (ej.: intentar abrir un fichero que no existe).

Algunas excepciones son fatales y causan el fin de la ejecución del programa. En este caso conviene terminar ordenadamente y enviar un mensaje explicando el tipo de error que se ha producido. En otras situaciones, por ejemplo cuando no se encuentra un archivo sobre el que se desea realizar una operación, el programa puede dar al usuario la oportunidad de corregir el error.

Cuando una excepción ocurre, decimos que fue **"lanzada**" y cuando manejamos dicho error, es decir hacemos algo con respecto del error, decimos que fue "**capturada**". La ejecución se transfiere al manipulador que pueda "lidiar" con la situación.



Un **buen programa debe manejar correctamente la mayoría de los errores que se puedan producir**, Java proporciona las siguientes herramientas para el manejo de excepciones: try, catch, throw, throws y finally.

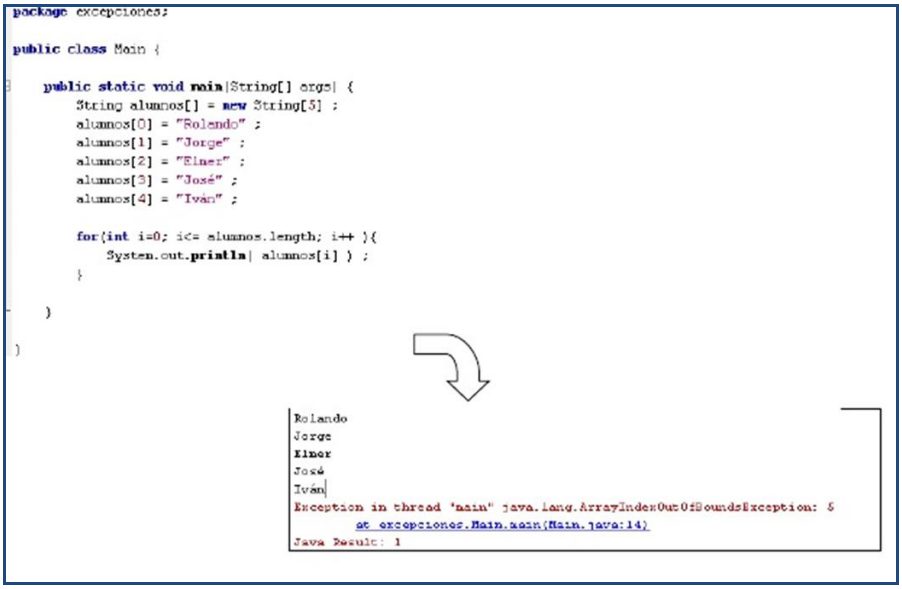


Cuando en **un programa se lanza una excepción y esta no es capturada**, la excepción supera los límites del programa, es capturada por la JVM, se detiene la ejecución y se muestra un mensaje. Ejemplo de mensaje, cuando se produce una división entre cero y no es capturada:

Exception in thread "main" java.lang.ArithmeticException: / by zero
at EjExcepcionFinally.UsoFinally.generaExcepcion(PruebaUsoFinally.java:12)

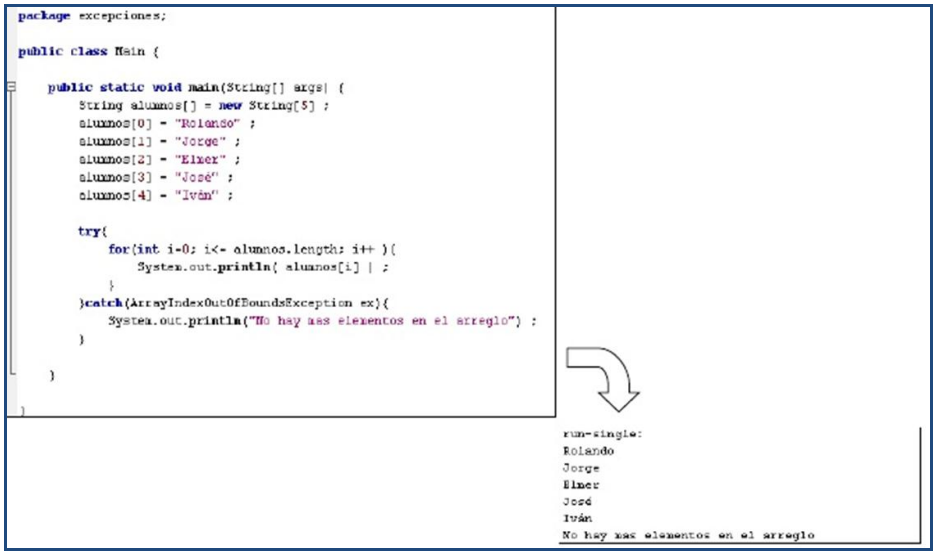
- **Ejemplo de un programa sin control de errores**:

Se produce un error en tiempo de ejecución por que se sobrepasa la longitud del array y se termina la ejecución del programa.



- **Ejemplo de un programa con control de errores**:

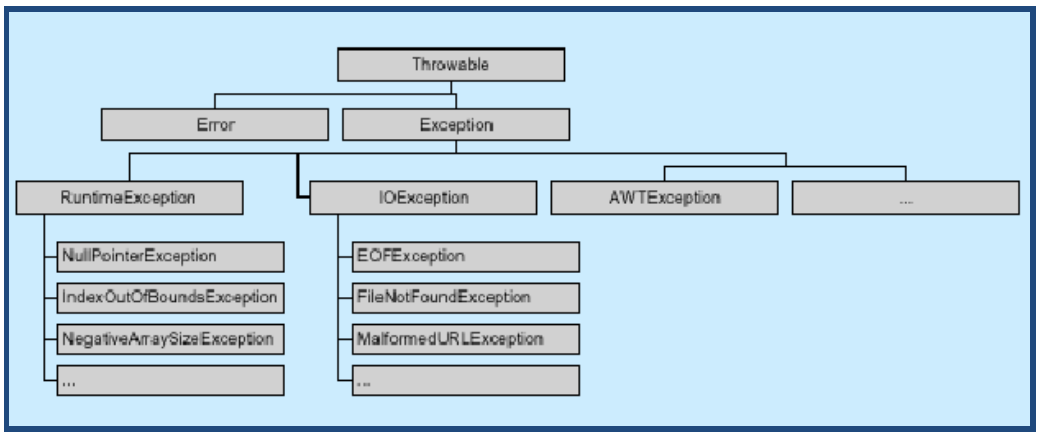
Ahora se trata el error por lo que se produce una terminación anormal del programa



En Java, **todas las excepciones** estan consideradas en el de árbol de excepciones que se **deriva de la clase Throwable**. Existen dos subclases directas de Throwable: **Error y Exception**. En la figura se observa la parte de la jerarquía de clases derivada de Throwable:

- **La clase Error:** está relacionada con errores de la máquina virtual de Java y no con el código, generalmente estos errores no dependen del programador por lo que no se espera que se capturen ni se traten. Ej: OutOfMemoryError

- **La clase Exception:** Las Excepciones derivadas de Exception sí que deben ser tratadas, y en algunos casos es obligatorio hacerlo para que el programa se compile.



Las **clases derivadas** de **Throwable** pueden **pertenecer a distintos packages de Java**. Algunas perenecen a java.lang (Throwable, Exception, RuntimeException, …); otras a java.io (EOFException, FileNotFoundException, ...) o a otros packages.

Constructores de Throwable:

Public Throwable().- Constructor por defecto.

Public Throwable(String msg).-

Da la oportunidad de construir el objeto con información que se transmite en la cadena

Por **heredar de Throwable** todos los tipos de excepciones pueden **usar los métodos siguientes**:

String getMessage(): Devuelve la cadena con que la excepción fue creada. Si fue creada por el constructor por omisión devuelve null.

String toString(): Devuelve un String que describe la excepción.

void printStackTrace():

Método que hace un vaciado del contenido de la Pila de llamadas de los métodos al momento de ocurrir el error

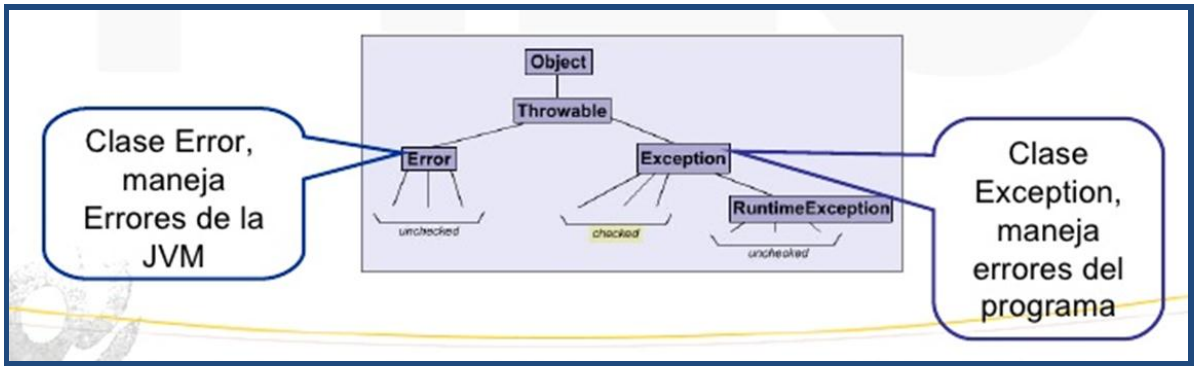
Ejemplo:

```
public class EjExcepcion1 {
        public static void main(String[] args) {
        int a=5, b=0;
        try
        {
           System.out.println("resultado:"+a/b);//se puede producir una excepción
        }
        catch(java.lang.ArithmeticException excepcion){
                System.out.println("Dividiendo por cero");
                System.out.println("mensaje devuelto por getMessage");
                System.out.println(excepcion.getMessage());
                System.out.println("mensaje devuelto por toString");
                System.out.println(excepcion.toString());
                 System.out.println("mensaje devuelto por printStackTrace()");
                excepcion.printStackTrace();
        }
    }
}
```

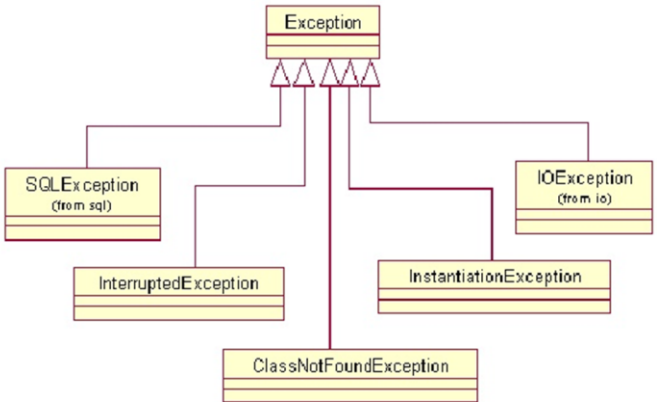
Salida:

```
run:
Dividiendo por cero
java.lang.ArithmeticException: / by zero
mensaje devuelto por getMessage
/ by zero
mensaje devuelto por toString
java.lang.ArithmeticException: / by zero
mensaje devuelto por printStackTrace()
        at ejexcepcion1.EjExcepcion1.main(EjExcepcion1.java:16)
BUILD SUCCESSFUL (total time: 0 seconds)
```

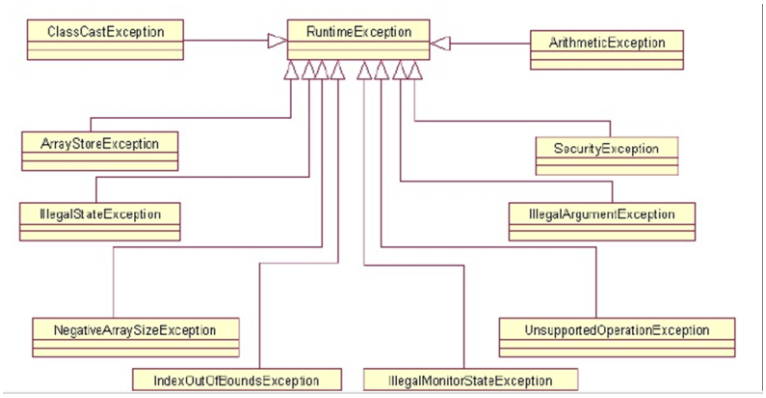
Además podemos considerar, los siguientes tipos de excepciones:



- **Excepciones comprobadas(Checked)**: son las excepciones que revisa el compilador y deben ser capturadas y tratadas para que el programa se compile.



- **Excepciones no comprobadas (Unchecked)**: son las excepciones que no revisa el compilador y se dan en tiempo de ejecución. Ejemplo: **RuntimeException**.



Como casi todo lo demás en java, **las excepciones son objetos** (que se crean cuando ocurre una situación anómala), se lanzan para que otra parte del código las capture y las trate.

2. Tratamiento de Excepciones

Cuando se produce un error en un método:

- Se **crea un objeto 'exception'** y se **le envía al método que lo ha provocado**. La excepción contiene información sobre la excepción, el tipo de excepción y el estado del programa al presentarse el problema.
- El **sistema de ejecución es el responsable de buscar algún bloque de código que maneje la excepción**.
- Cuando **un método ha provocado una excepción** tiene dos posibilidades:
  - **Gestionarla él mismo**: Capturarla y tratarla: **try y catch**
  - **Pasarla al método invocador**: Pasar el objeto de tipo excepción al método que lo invocase: throws

El núcleo del manejo de excepciones son los bloques try y catch.

A continuación se muestra la forma general del manejo de bloques de excepción try/catch:

El código que puede generar la excepción debe encerrarse dentro de un bloque **try:**

```
try
{
// Código que puede generar la excepción
}
```

A continuación, **la excepción se captura con un bloque catch**

```
catch ((Tipo_de_Excepción  Objeto_Excepcion){
{
// Código para tratar el error
}
```

Cuando un segmento de código lanza una excepción, esta es atrapada por su correspondiente manejador catch.

En un mismo bloque de instrucciones se puede generar más de una excepción, por lo que puede haber *más de un bloque catch asociado a un solo try*. Esto se puede hacer:

- Gracias a que **unas excepciones heredan de otras**.
- **Sólo se puede hacer con excepciones dentro de la misma jerarquía**.
- Sirve para tratar de manera común varios tipos de excepciones distintos

```
try{
/ código que pudiera ocasionar una excepción
}
catch (Tipo_de_Excepción1  Objeto_Excepcion){
código para manejar la excepción
}
catch (Tipo_de_Excepción2  Objeto_Excepcion){
código para manejar la excepción
}
```

El siguiente programa genera varias excepciones en el momento de su ejecución: una cuando se va a dividir por cero y otra cuando en el vector num y den nos pasamos de rango.

El bloque try de este método tiene tres posibilidades de salida diferentes:

- La sentencia num[i]/den[i] falla cuando den[i] es cero y lanza una **ArithmeticException.**
- La sentencia num[i] o den[i] falla y lanza una **ArrayIndexOutOfBoundsException**.
- **Todo tiene éxito y** la sentencia try sale normalmente visualizando el resultado.

class ExcepcionesMultiples{

```
    static void divide(){
       int num[]={4,8,16,32,64,128,256};
       int den[]={2,0,4,4,0,8};
       for (int i=0;i<num.length+1;i++)
       {
         try{
                  System.out.println(num[i]+ "/" + den[i]+ "=" + num[i]/den[i]);
         }
         catch(java.lang.ArithmeticException excepcion){
                  System.out.println("Dividiendo por cero");
         }
         catch(java.lang.ArrayIndexOutOfBoundsException excepcion){
                  System.out.println("Error al acceder al vector");
        }
       }
     }
}
public class PruebaExcepcionesMultiples{
    public static void main (String args[]){
        ExcepcionesMultiples.divide();
      }
}
```

SALIDA:

```
run:
4/2=2
Dividiendo por cero
16/4=4
32/4=8
Dividiendo por cero
128/8=16
Error al acceder al vector
Error al acceder al vector
BUILD SUCCESSFUL (total time: 0 seconds)
```

Si ocurre una excepción en un bloque try:

- **Este bloque termina inmediatamente**.
- El control de programa se transfiere a la primera cláusula catch que vaya después del bloque try.
- Al igual que con cualquier bloque de código, al terminar un bloque try, **las variables locales** declaradas dentro de este bloque, **quedan fuera de alcance**.
- A continuación, el **programa busca la primera cláusula catch que pueda procesar el tipo de excepción que ocurrió**. Cuando la encuentra, se ejecuta el código del bloque catch concordante.
- Cuando una cláusula catch termina de procesarse, las variables locales que se declaran dentro del bloque catch quedan fuera del alcance.
- **Cualquier bloque catch restante correspondiente a ese bloque try se ignora** y la **ejecución continúa en la primera línea de código después de la secuencia try/catch**.

Si no ocurren excepciones en un bloque try, el programa ignora los manejadores para este bloque. La ejecución del programa continúa con la siguiente instrucción que haya después de la secuencia try/catch

CUIDADO: Si un **bloque de código lanza varias excepciones** y se usan varios catch:

- La excepción **se captura en el primer catch que se ajusta a la excepción**.
- Los catch **deben capturar las excepciones más concretas** en primer lugar, y las **más**

generales al final.

- Si no lo hacemos así, hay bloques catch en los que no se entrará nunca.

```
try                                    try
{                                      {
    //código que genera                    //código que genera
    //excepciones                          //excepciones
}                                      }
catch(IOException ioe)                 catch(Exception e)
{                                      {
    //catch accesible                      //catch accesible
}                                      }
catch (Exception e)                    catch (IOException ioe)
{                                      {
    //catch accesible                      //catch NO accesible
}                                      }
```

Por ejemplo, si se incluyen dos bloques catch, uno que capture Exception y otro que capture NullPointerException, este último deberá colocarse el primero porque de lo contrario nunca podría llegar a ejecutarse.

```
public class TryCatchFinally {
    public static void main(String[] args) {
        try {
            System.out.println("Paso 1");
            int a = 10 / 0;      // Lanza una ArithmeticException
            System.out.println("Paso 2");
        } catch(ArithmeticException ae) {
            System.out.println("Paso 3");
        } catch (Exception e) {
            System.out.println("Paso 4");
        }

        try {
            System.out.println("Paso 5");
            int a = 10 / 1;      // NO lanza una excepción
            System.out.println("Paso 6");
            Object b = null;
            b.toString();        // Lanza una NullPointerException
            System.out.println("Paso 7");
        } catch(ArithmeticException ae) {
            System.out.println("Paso 8");
        } catch (Exception e) {
            System.out.println("Paso 9");
        }
    }
}
```

```
Símbolo del sistema

C:\Practicas\Programacion>java TryCatchFinally
Paso 1
Paso 3
Paso 5
Paso 6
Paso 9

C:\Practicas\Programacion>_
```

A veces, cuando se produce una excepción, la aplicación queda es un estado inestable. Al tratamiento de una excepción se le puede añadir al final un bloque **finally** que se **ejecuta siempre**, se produzcan o no excepciones, incluso si dentro de los bloques try/catch hay una sentencia continue, break o return. Se puede usar para cerrar ficheros, liberar recursos, etc.

Como ejemplo, se podría pensar en un bloque try dentro del cual se abre un fichero para lectura y escritura de datos y se desea cerrar el fichero abierto. El fichero abierto se debe cerrar tanto si produce una excepción como si no se produce, ya que dejar un fichero abierto puede provocar problemas posteriores. Para conseguir esto se deberá incluir las sentencias correspondientes a cerrar el fichero dentro del bloque finally.

```
try{
código que produce la (s) excepción (es)...
}
catch( TipoDeExcepcion objeto){
// Código para manejar la excepción
}
finally{
// código de finally
}
```

Ejemplo:

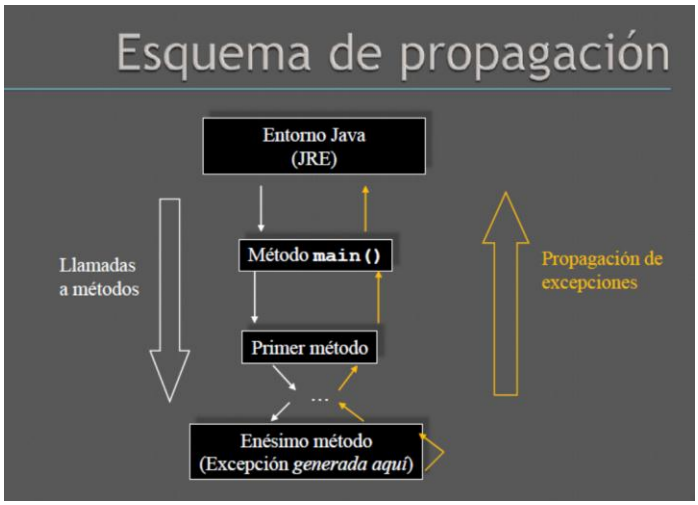
```
package EjExcepcionFinally;
class UsoFinally{
    public static void generaExcepcion(int i){
        int t;
        int num[] = {2,4,6};
        System.out.println("valor de i="+i);
        try{
            switch(i){
                        case 0: t=10/i; //division por cero
                        break;
                        case 1:num[4]=4; //genera un error. Fuera de rango
                      }
            }
        catch(ArithmeticException exc){
           System.out.println("No puede dividir entre cero");
         }
        catch(ArrayIndexOutOfBoundsException exc) {
           System.out.println(" No hay elementos que coincidan. Fuera de rango");
        }
        finally {
           System.out.println("Este código se ejecuta siempre");
        }
    } //fin de metodo
}//clase
public class PruebaUsoFinally {
    public static void main(String args[]){
        for (int i=0;i<4; i++)
        {   UsoFinally.generaExcepcion(i);
            System.out.println();
        }
    }
}
```

SALIDA:

```
run:
valor de i=0
No puede dividir entre cero
Este código se ejecuta siempre

valor de i=1
 No hay elementos que coincidan. Fuera de rango
Este código se ejecuta siempre

valor de i=2
Este código se ejecuta siempre

valor de i=3
Este código se ejecuta siempre
```

3. Propagación de excepciones

- Al **producirse un error en un método m**:
  - Se **genera un objeto** que representa el error (Excepción).
  - La JVM **busca un gestor adecuado dentro del propio método**.
    - **Si el gestor existe**, cederá el control a dicho gestor.
    - **Si el gestor no existe**, buscará el gestor en el método que haya invocado al método m, y así sucesivamente, hasta encontrar un gestor capaz de tratar la excepción producida.
- Las **excepciones se propagan de método en método** en sentido contrario a las llamadas:
  - Hasta que algún catch las captura y entonces la ejecución continúa por ahí
  - Si la excepción alcanza el método main() sin ser capturada, el programa y todo el JRE se detiene, mostrando el estado de la pila de llamadas por pantalla



Ejemplo:

```
public class Excepciones {
    public static void main(String[] args) {
        try {
            metodo1();
        } catch(NullPointerException npe) {
            System.out.println("Se ha producido una" +
                "NullPointerException, capturada en el main");
        }
    }

    public static void metodo1() {
        try {
            metodo2();
        } catch(NullPointerException npe) {
            System.out.println("Se ha producido una" +
                "NullPointerException, capturada en el metodo 1");
        }
    }

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

    public static void metodo3() {
        Object a = null;
        a.toString();   // Esto generará una NullPointerException
    }
}
```

```
Símbolo del sistema
C:\Practicas\Programacion>java Excepciones
Se ha producido unaNullPointerException, capturada
 en el metodo 1

C:\Practicas\Programacion>_
```

- Los **métodos obligatoriamente deben declarar las excepciones comprobadas** que lanzan en su cabecera con la palabra reservada **throws**.

4. Bloques try/catch anidados

Se **pueden anidar varias sentencias try/catch**:

- La búsqueda del gestor de la excepción se hace de los **bloques más internos a los más externos**.
- Cuando se ha encontrado un gestor de la **excepción se ejecuta el código correspondiente, no se sigue propagando la excepción**.

Ejemplo1:

```
public class TryCatchAnidados1 {
    public static void main(String[] args) {
        try {
            System.out.println("Paso 1");
            try {
                System.out.println("Paso 2");
                Object o = null;
                o.toString();   // NullPointerException
                System.out.println("Paso 3");
            } catch(NullPointerException npe) {
                System.out.println("Paso 4");
            }
            System.out.println("Paso 5");
        } catch(Exception e) {
            System.out.println("Paso 6");
        }
    }
}
```

```
Símbolo del sistema

C:\Practicas\Programacion>java TryCatchAnidados1
Paso 1
Paso 2
Paso 4
Paso 5

C:\Practicas\Programacion>
```

Ejemplo2:

```
public class TryCatchAnidados2 {
    public static void main(String[] args) {
        try {
            System.out.println("Paso 1");
            try {
                System.out.println("Paso 2");
                int[] a = new int[] {1,2};
                int b = a[5]; // ArrayIndexOutOfBoundsException
                System.out.println("Paso 3");
            } catch(NullPointerException npe) {
                System.out.println("Paso 4");
            }
            System.out.println("Paso 5");
        } catch(Exception e) {
            System.out.println("Paso 6");
        }
    }
}
```

```
Símbolo del sistema

C:\Practicas\Programacion>java TryCatchAnidados2
Paso 1
Paso 2
Paso 6

C:\Practicas\Programacion>
```

5. Sentencias throw y throws

- **La sentencia throw** se utiliza para **"lanzar" (crear) una excepción explícitamente desde el código**.

  Al **lanzar una excepción explícitamente** se **interrumpirá el flujo de ejecución** y se buscará un *código que la gestione* (sentencia catch), al igual que con las excepciones lanzadas implícitamente por la JVM.

  Para poder lanzar una excepción, es necesario que el objeto que lanzamos (la excepción) sea de la clase Throwable o de cualquier clase que herede de ésta (Error y Exception heredan de Throwable)

Ejemplo1:

```
public class SentenciaThrow1 {
    public static void main(String[] args) throws Exception {
        System.out.println("Paso 1");
        int a = 1;
        if(a == 1) {
            Exception e;
            e = new Exception();
            throw e;
        }
        System.out.println("Paso 2");
    }
}
```

```
Símbolo del sistema

C:\Practicas\Programacion>java SentenciaThrow1
Paso 1
Exception in thread "main" java.lang.Exception
        at SentenciaThrow1.main(SentenciaThrow1.java:8)

C:\Practicas\Programacion>
```

Ejemplo 2:

```
public class SentenciaThrow2 {
    public static void main(String[] args) throws Exception {
        try {
            int a = 1;
            System.out.println("Paso 1");
            NullPointerException npe;
            npe = new NullPointerException();
            if(a == 1) {
                throw npe;
            }
            System.out.println("Paso 2");
        } catch(NullPointerException npe) {
            System.out.println("Paso 3");
            npe.printStackTrace();
            System.out.println("Paso 4");
        }
    }
}
```

```
Símbolo del sistema

C:\Practicas\Programacion>java SentenciaThrow2
Paso 1
Paso 3
java.lang.NullPointerException
        at SentenciaThrow2.main(SentenciaThrow2.java:8)
Paso 4

C:\Practicas\Programacion>
```

6. Excepciones comprobadas(checked):

Cuando **un método no tiene un código para gestionar un determinado tipo de excepción**, pero en este método se puede generarse una excepción de ese tipo, el método debe declarar explícitamente que podría generar una excepción de este tipo.

- Para ello **se utiliza la sentencia throws**, que indica que un método puede lanzar un determinado tipo de excepción.
- Sólo es necesario **declarar explícitamente que un método puede lanzar una excepción** si ésta hereda de la clase Excepcion, y además no hereda de RuntimeException.
- Por tanto, la posible generación de excepciones que heredan de **Error o** heredan de **RuntimeException NO deben declararse explícitamente**.

Ejemplo: Si tratamos de compilar una clase que tuviera el siguiente método:

```
public boolean abreArchivo(){
    new FileInputStream("archivo.txt");
    return true;
}
```

El compilador nos daría un **error** debido a que ***no estamos capturando la excepción FileNotFoundException*** que puede ser lanzada.

Unreported exception java.io.FileNotFoundException;

Must be caugt or declared to be throws

Si nos fijamos en el contructor de la clase FileInputStream vemos que puede lanzar la excepción ***FileNotFoundException***.

Constructor Detail

FileInputStream

```
public FileInputStream(String name)
                throws FileNotFoundException
```

Para **corregir el problema** podemos hacer dos cosas:

- **Capturar la excepción**: Gestionar la excepción con una construcción del tipo try {…} catch {…}:

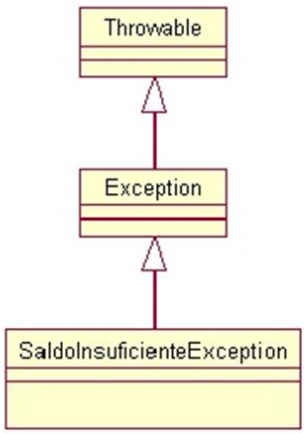
```
public boolean abreArchivo(){
try{
   new FileInputStream("archivo.txt");
   return true;
}
Catch( FileNotFoundException e){
       System.out.println("No se puede abrir el fichero");

}
}
```

- O indicar en la **declaración del método que la excepción puede ser lanzada**. Para ello se utiliza la **palabra throws en la cabecera del métdo**. En este **caso la excepcion se propaga** hacia un método anterior en la pila de llamadas.

```
public boolean abreArchivo()throws FileNotFoundException{
   new FileInputStream("archivo.txt");
   return true;
}
```

7. Lanzar nuestras propias excepciones

Si hacemos métodos propios, podemos lanzar excepciones en caso de que algo vaya mal. Estas excepciones pueden ser las de java o bien unas que nos creemos nosotros.




```
package excepciones;

public class SaldoInsuficienteException extends java.lang.Exception {

    public SaldoInsuficienteException() {
    }

    public SaldoInsuficienteException(String msg) {
        super(msg);
    }
}
```

El **programador puede crear sus propias excepciones** sólo con **heredar de la clase Exception o de una de sus clases derivadas**.

Lo lógico es heredar de la clase de la jerarquía de Java que mejor se adapte al tipo de excepción.

Las **clases Exception suelen tener dos constructores**:

Un constructor sin argumentos.

Un constructor que recibe un String como argumento. En este String se suele definir un mensaje que explica el tipo de excepción generada. Conviene que este constructor llame al constructor de la clase de la que deriva super(String).

Al ser clases como cualquier otra se podrían incluir variables y métodos nuevos. Por ejemplo:

```
class MiExcepcion extends Exception {
  public MiExcepcion() { // Constructor por defecto
     super();
  }
  public MiExcepción(String s) { // Constructor con mensaje
     super(s);
 }
}
```

Nuestro método que lanza esta excepción, o una de java, debe declararse con un **throws**:

```
public class MiClase
{
   public int miMetodoQueLanzaExcepcionSiHayFallo (...) throws MiExcepcion
   {
      ejecutoMiCodigo();
      // si hay fallo, se lanza la excepcion.
      if (hayFallo())
         throw new MiExcepcion(...);
      return resultado;
   }
}
```

Ej: Para lanzar la excepción, como vemos en ese código, basta con hacer **un new de MiExcepción y lanzarla con throw**.

```
package ExcepcionPropia;
public class ExcepcionIntervalo extends Exception {
    public ExcepcionIntervalo(String msg) {
        super(msg);
    }
}
```

```
package ExcepcionPropia;
import java.util.Date;
import java.util.Random;
public class ExcepcionAplicacion {
     public static void main(String[] args) {
       Random rnd = new Random();
       rnd.setSeed(new Date().getTime());
//para cada vez que se ejecuta cambia la semilla para que no se repitan los mismos
//números
        for (int j=1;j<6;j++)
        {
         try{
              int i=rnd.nextInt(150)+1;
              System.out.println( "numero generado:"+i);
              rango(i);
              System.out.println( "numero generado dentro rango");
         }
         catch (ExcepcionIntervalo e){
             System.out.println( e.getMessage() );       }
```

```
            }
        }
    static void rango(int num1) throws ExcepcionIntervalo{
            if ((num1<20)||(num1>100)){
                throw new ExcepcionIntervalo("Número fuera de rango");
            }
        }
}
```

SALIDA:

```
run:
numero generado:55
numero generado dentro rango
numero generado:81
numero generado dentro rango
numero generado:2
Número fuera de rango
numero generado:14
Número fuera de rango
numero generado:61
numero generado dentro rango
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ejemplo2 :

1. Lanzamiento de excepciones

```
class CuentaBancaria {
   ...
   boolean bloqueada;
   ...
   void retirar (long cantidad) throws SaldoInsuficiente,
                                       CuentaBloqueada {
      if (bloqueada)
         throw new CuentaBloqueada (numero);
      else if (cantidad > saldo)
         throw new SaldoInsuficiente (numero, saldo);
      else saldo -= cantidad;
   }
}
```

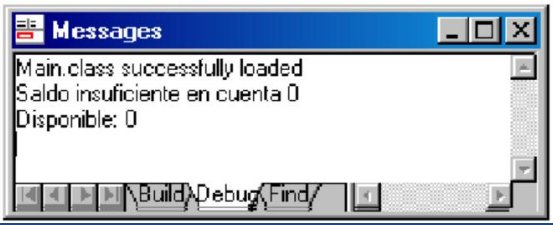
2. Definición de clases para excepciones

```
class SaldoInsuficiente extends Exception {
   long numero, saldo;
   SaldoInsuficiente (long num, long s) {
      numero = num; saldo = s;
   }
   public String toString () {
      return "Saldo insuficiente en cuenta " + numero
             + "\nDisponible: " + saldo;
   }
}
class CuentaBloqueada extends Exception {
   long numero;
   CuentaBloqueada (long num) { numero = num; }
   public String toString () {
      return "La cuenta " + numero + " esta bloqueada";
   }
}
```

3. Captura y procesamiento de excepciones

```
static public void main (String args[]) {
   try {
      new CuentaBancaria () .retirar (100000);
   }
   catch (SaldoInsuficiente excep) {
      System.out.println (excep);
   }
   catch (CuentaBloqueada excep) {
      System.out.println (excep);
   }
}
```

Se ejecuta el primer catch de tipo compatible



4. ¿Y si no se captura una excepcion?

```
static public void main (String args[])
            throws CuentaBloqueada, SaldoInsuficiente {
   CuentaBancaria cuenta = new CuentaBancaria ();
   cuenta.saldo = 1000;
   cuenta.retirar (2000);
}
```