EXCEPCIONES

Durante este capítulo 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.

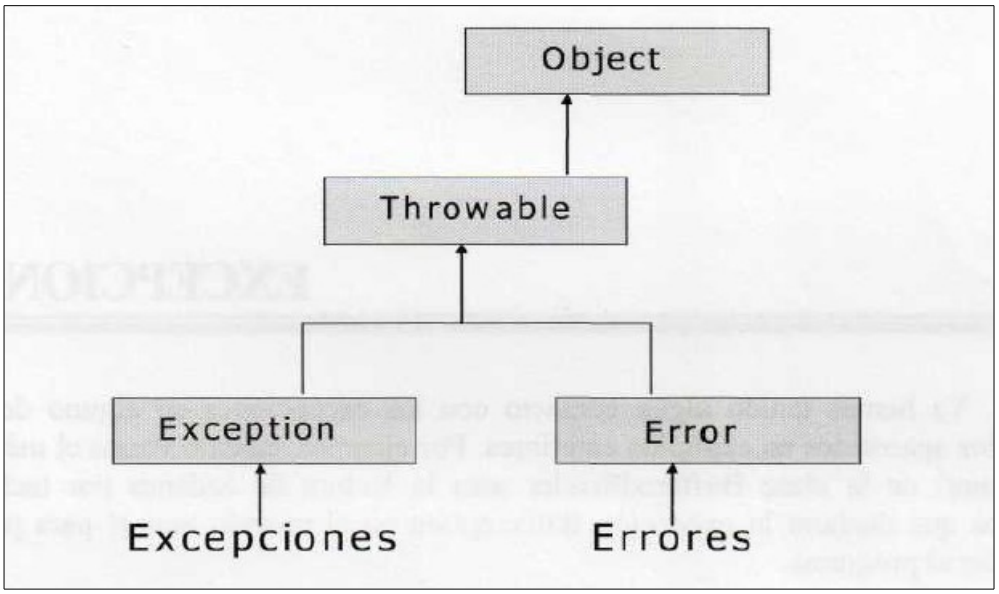
1. EXCEPCIONES Y ERRORES

Una excepción es una situación anómala que puede producirse durante la ejecución de un programa, como puede ser un 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, como por ejemplo un fallo de la máquina virtual. Por regla general, un programa no deberá intentar recuperarse de un error, dado que son situaciones que se escapan al control del programador.

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 la clase **`Throwable`**

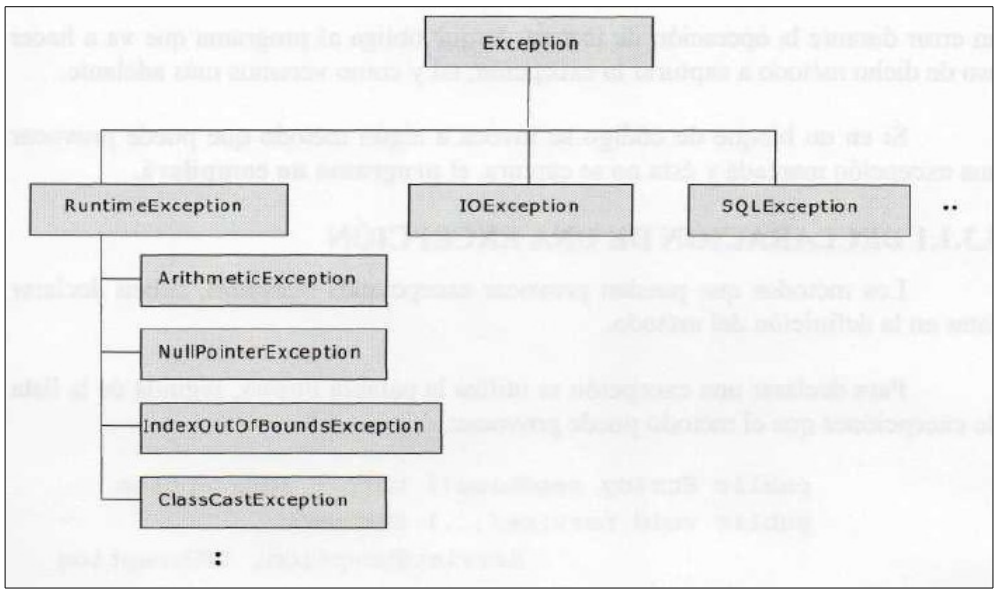


Superclases de excepciones y errores

2. CLASES DE EXCEPCIÓN

Al producirse una excepción en un programa, se crea un objeto de la clase `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 la misma.

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



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 (checked)

Se entiende por excepciones marcadas (también denominadas comprobadas) aquellas 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 los mismos.

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 indicarlo declarando las mismas 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
```

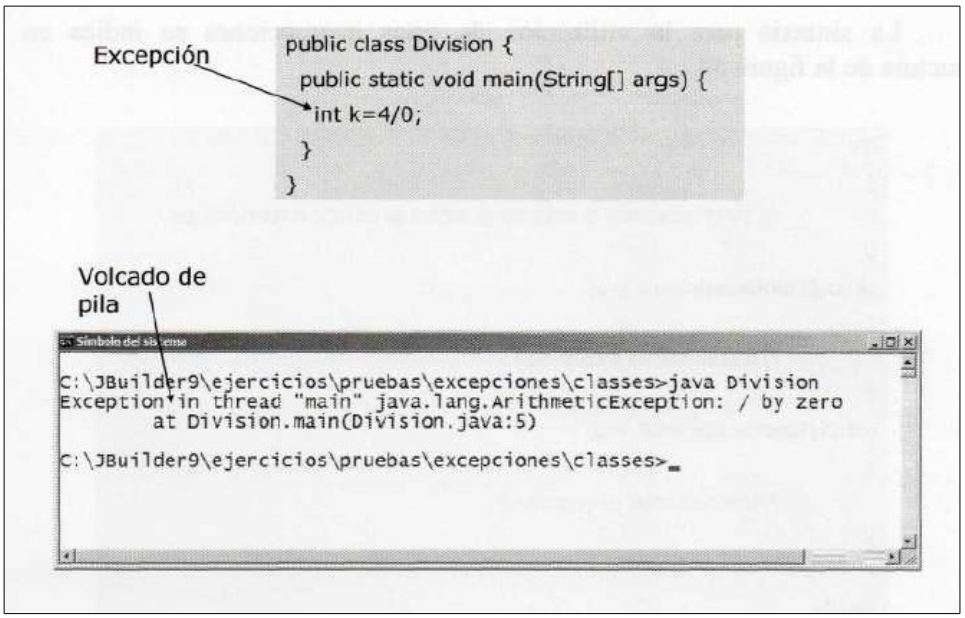
Así, siempre que vayamos a utilizar algún método que tenga declaradas excepciones, hemos de tener presente que estamos obligados a capturar dichas excepciones.

3.2. Excepciones no marcadas (unchecked)

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 (`ArrayIndexOutOfBoundsException`, `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 ésta, sino por evitar que se produzca.

Si durante la ejecución de un programa Java se produce una excepción y ésta no es capturada, la máquina virtual 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 del mismo.



Efecto de una excepción no controlada

4. CAPTURA DE EXCEPCIONES

Como ya se apuntó anteriormente, en el momento en que se produce una excepción en un programa, se crea un objeto de la clase de `Exception` correspondiente y se "lanza" en 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.

```
try
{
        // instrucciones donde se pueden producir excepciones
}
catch(TipoExcepcion1 arg)
{
        //tratamiento excepcion1
}
catch(TipoExcepcion2 arg)
{
        //tratamiento excepcion2
}
:
finally
{
        //instrucciones de última ejecución
}
```

Estructura de código para la captura de excepciones

TRY

El bloque ***try*** delimita aquella o aquellas instrucciones donde se pue producir una excepción. Cuando esto sucede, 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 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*, por ejemplo, una excepción `NullPointerException` podría ser tratada en un catch que capturase directamente dicha excepción y en uno que capturase `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 del mismo, 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.

Esto queda reflejado en el siguiente ejemplo:

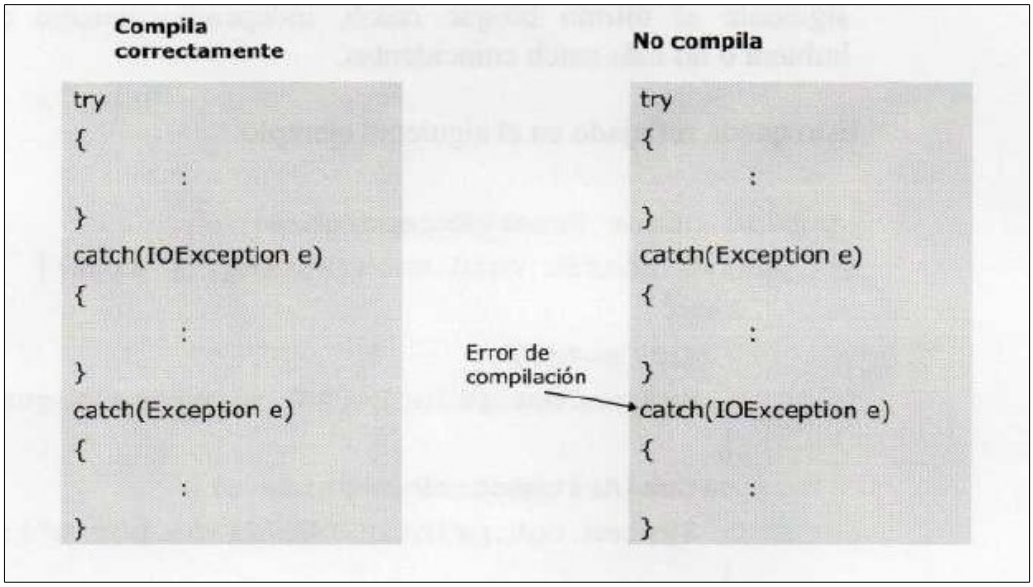
```
public class PruebaExcepciones{
   public static void main (String[] args) {
         try{
              int s = 4 / 0;
              System.out.println("El programa sigue");
         }

         catch (ArithmeticException e){
              System.out.println("División entre 0");
         }

         catch (Exception e){
              System.out.println("Excepción general");
         }

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

Tras la ejecución del método *main()*, se mostrará en pantalla lo siguiente:

```
División entre 0
Final del main
```

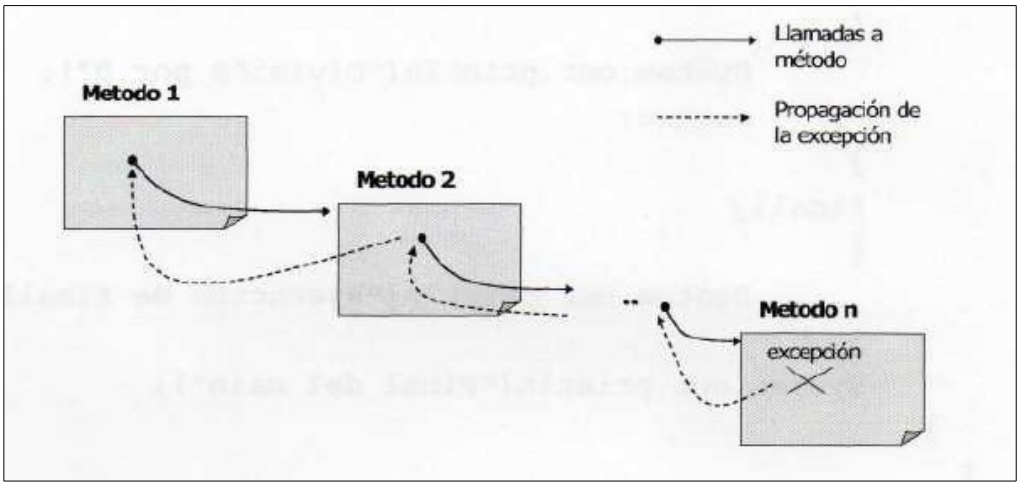
- Del listado anterior se deduce otro punto importante a tener en cuenta en el tratamiento de excepciones: tras la ejecución de un bloque *catch*, **el control del programa nunca se devuelve al lugar donde se ha producido la excepción.**

- 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.



Diferencia entre situar los bloques catch específicos antes y después de los genéricos

- 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.



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 *fmally* se ejecutará tanto si se produce una excepción como si no, garantizando así que un determinado conjunto de instrucciones siempre sean 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 ilustra el funcionamiento de *finally*.

```
public class Excepciones{
  public static void main(String[] args) {
      try {
            int s = 4 / 0;
            System.out.println("El programa sigue");
      } catch(ArithmeticException e) {
            System.out.println("División entre O");
            return;
      } finally {
            System.out.println("Ejecución de finally");
      }
      System.out.println("Final del main");
   }
}
```

Tras la ejecución del método *main()* se mostrará en pantalla lo siguiente:

```
División por 0
Ejecución de finally
```

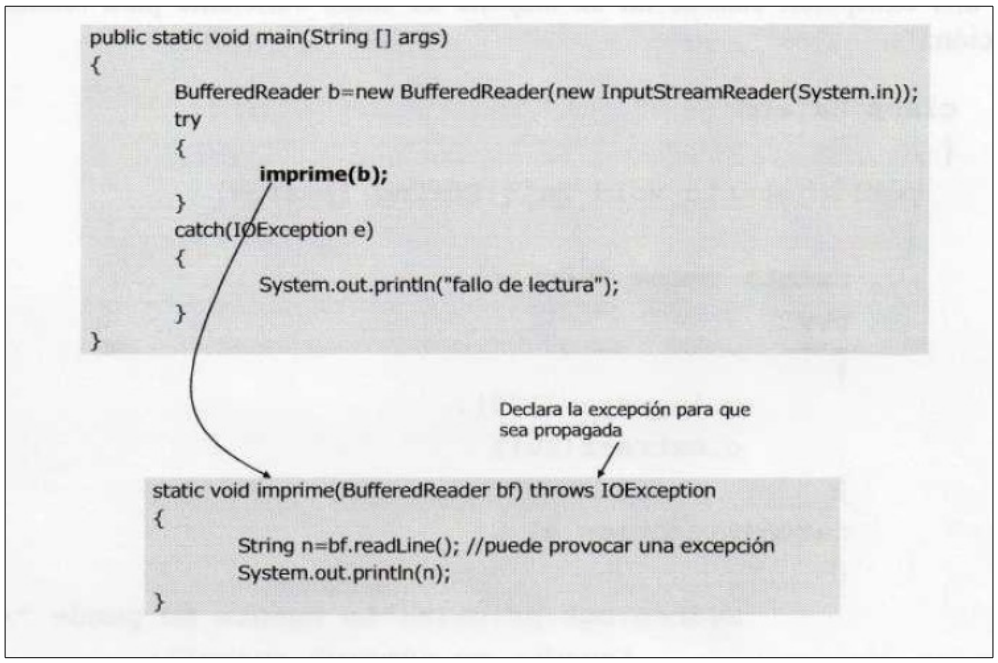
Esto demuestra que, **aún existiendo una instrucción para la salida del método (return), el bloque finally se ejecutará antes de que esto suceda.**

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:



Propagación de una excepción

En este ejemplo el método *main()* es el que captura la excepción producida en *imprime()*. 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.

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 ejemplo siguiente 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:

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

class Cuenta {
      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;
      }
}
```

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.

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 sobrecarga del método `printStackTrace()` permite enviar el volcado de pila a un objeto *PrintStream* cualquiera, por ejemplo, un fichero de log.

7. CLASES DE EXCEPCIÓN PERSONALIZADAS

Cuando un método necesita lanzar una excepción como forma de 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:

```
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.

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

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

```
class Cuenta {
     double saldo;
     public Cuenta() {
         saldo = 0;
     }

     public void ingresar(double c) {
         saldo += c;
     }

     public void extraer(double c) throws
                                SaldoInsuficienteException {
         if (saldo < c)
               throw new SaldoInsuficienteException("números
               rojos");
         else
               saldo -= c;
     }

     public double getSaldo(){
         return saldo;


     }
}
```

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

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