EXCEPCIONES

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

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.

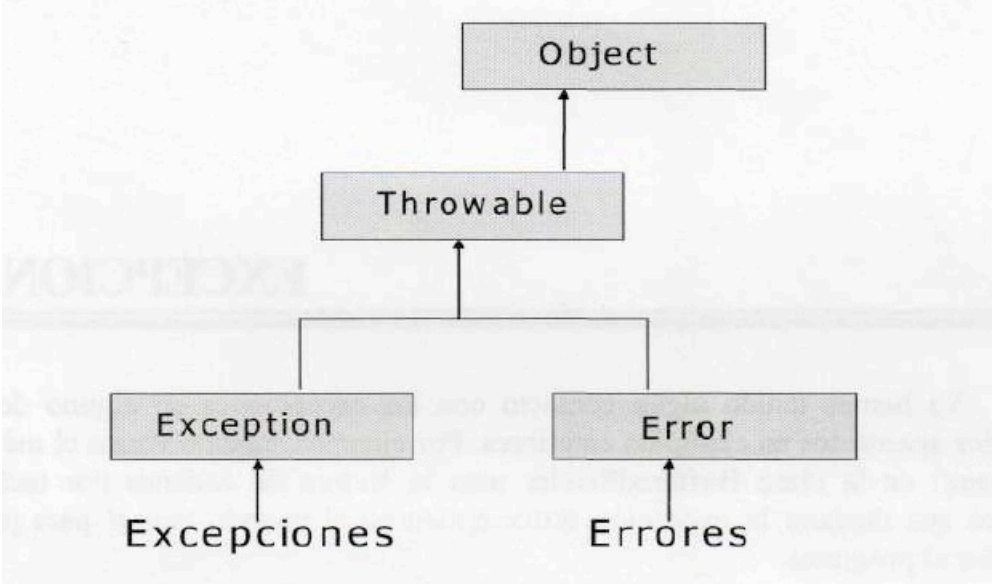
1 EXCEPCIONES Y ERRORES

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 **Throwable**

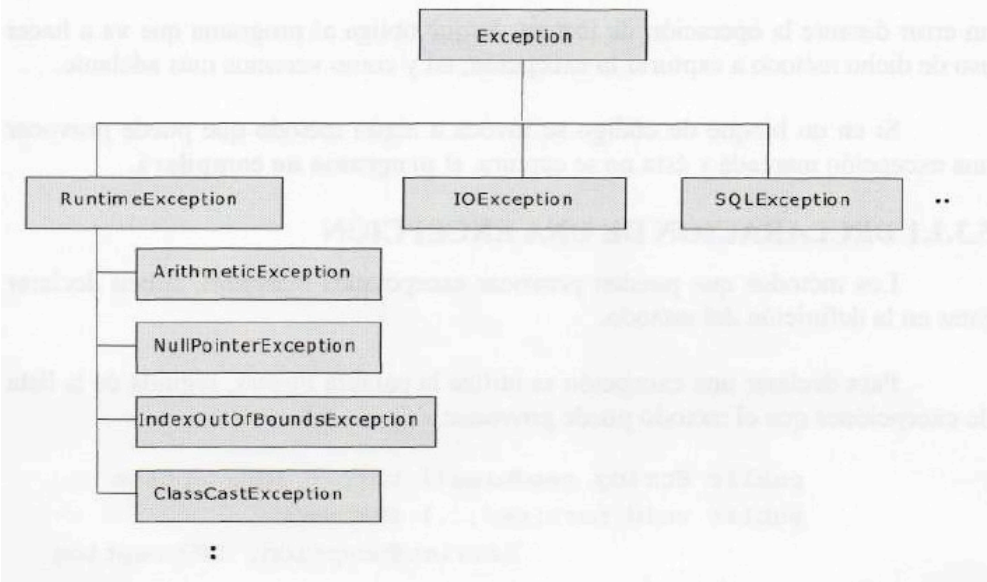


Superclases de excepciones y errores

2 CLASES DE EXCEPCIÓN

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

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

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

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

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

DECLARACIÓN DE UNA EXCEPCIÓN

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

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

```
public String readLine() throws lOException

public void service(...) throws ServletExceptíon, lOException
```

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

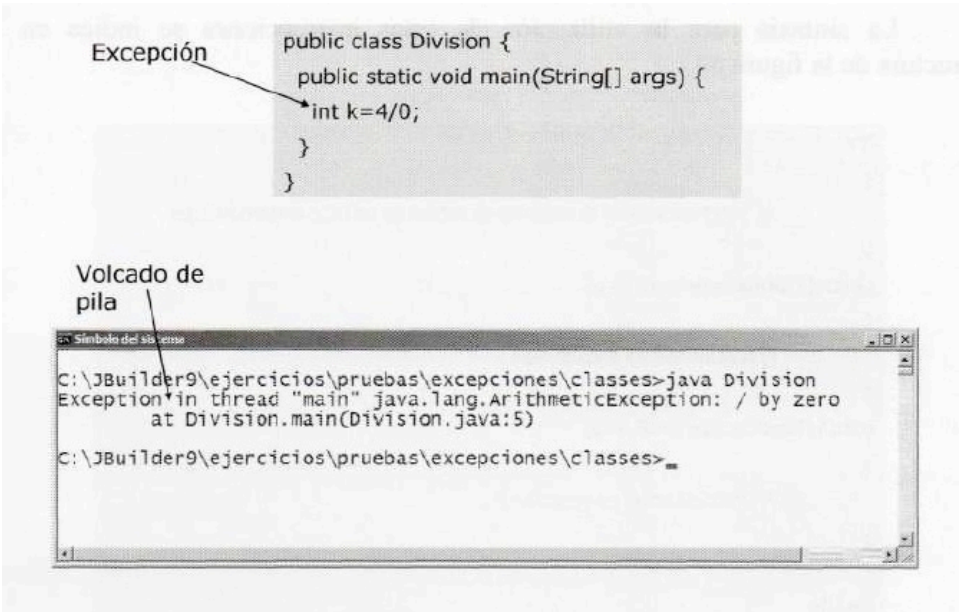
3.2 Excepciones no marcadas

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

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

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

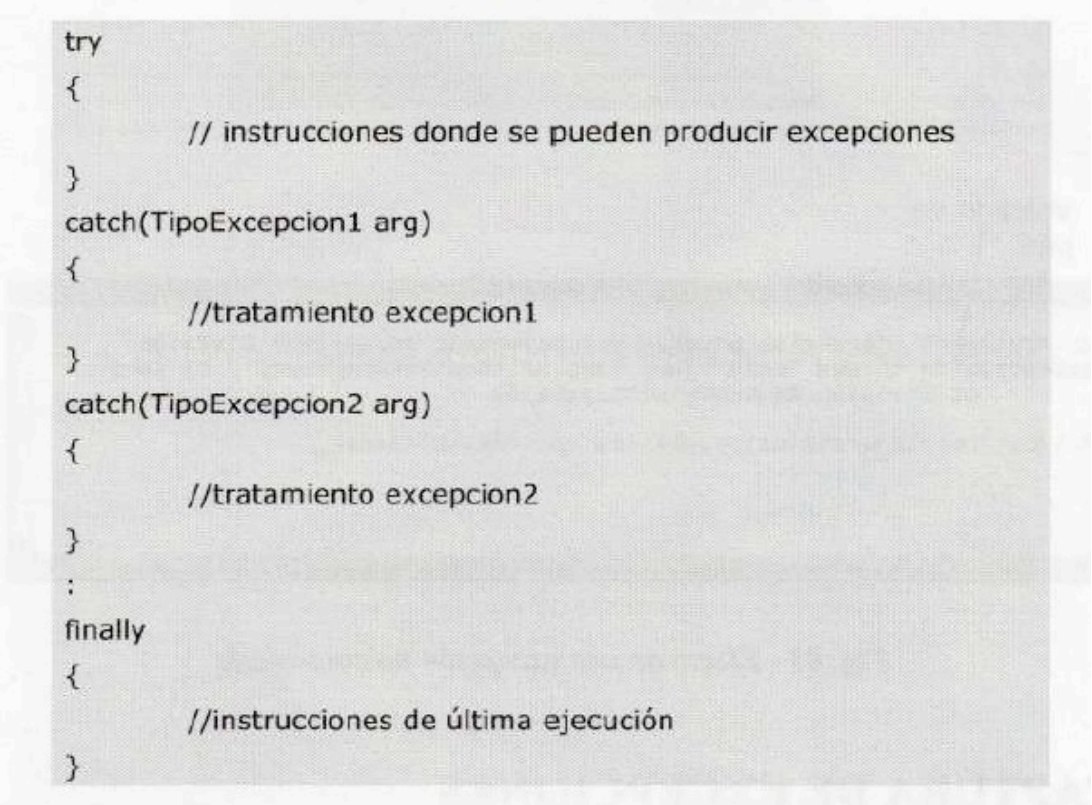
Como ya se apuntó anteriormente, en el momento en que se produce una excepción en un programa, se crea un objeto de la clase de excepción correspondiente y se "lanza" a la línea de código donde la excepción tuvo lugar.

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

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

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

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



```
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 de que se produzca un determinado tipo de excepción.

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

- Se pueden definir tantos bloques *catch* como se considere necesario. Cada bloque *catch* servirá para tratar un determinado tipo de excepción, **no pudiendo haber dos o más catch que tengan declarada la misma clase de excepción.**

- Un bloque *catch* sirve para capturar cualquier excepción que se corresponda con el tipo declarado o cualquiera de sus subclases. Por ejemplo, un *catch* como el siguiente:

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

se ejecutaría al producirse cualquier excepción de tipo NullPointerException, ArithmeticException, etc. Esto significa que una excepción podría ser tratada en diferentes *catch,* 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 de este, el control del programa se transferirá al bloque *finally* o, si no existe, a la instrucción siguiente al último bloque *catch,* independientemente de que hubiera o no más catch coincidentes.

  Esto queda reflejado en el siguiente ejemplo: (pasar a ordenador)

```
public class PruebaExcepciones{
   public static void main (String [] args) {
         try{
              int s=4/0;
              System . out .println {"E1 programa sigue");
         }

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

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

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

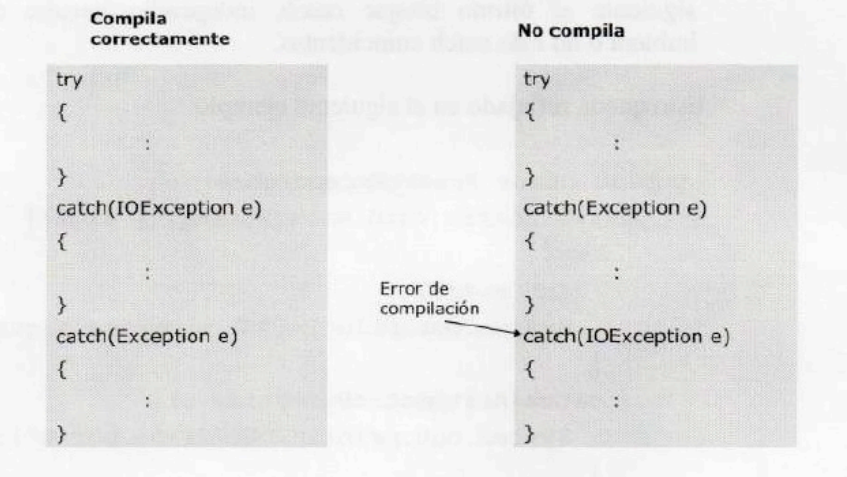
  Tras la ejecución del método *main(),* se mostrará en pantalla lo siguiente:

  División por 0

  Final del main

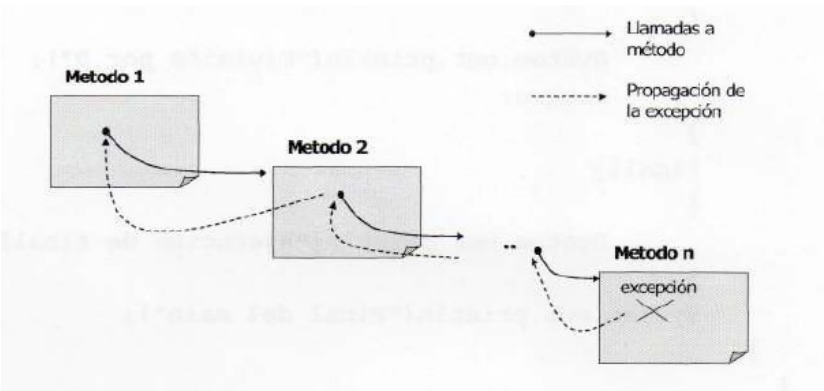
- Del listado anterior se deduce otro punto importante a tener en cuenta en el tratamiento de excepciones: tras la ejecución de un bloque *catch,* **el control del programa nunca se devuelve al lugar donde se ha producido la excepción.**

- 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.* (Modifica el ejemplo anteriormente pasado)

```
public class Excepciones{
  publíc static voíd main(Stríng [] args){

      try
      {
            int s=4/0;
            System.out.println("El programa sigue");
      }
      catch{ArithmeticExceptíon e)
      {
            System, out .println ( "División por 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 fmally

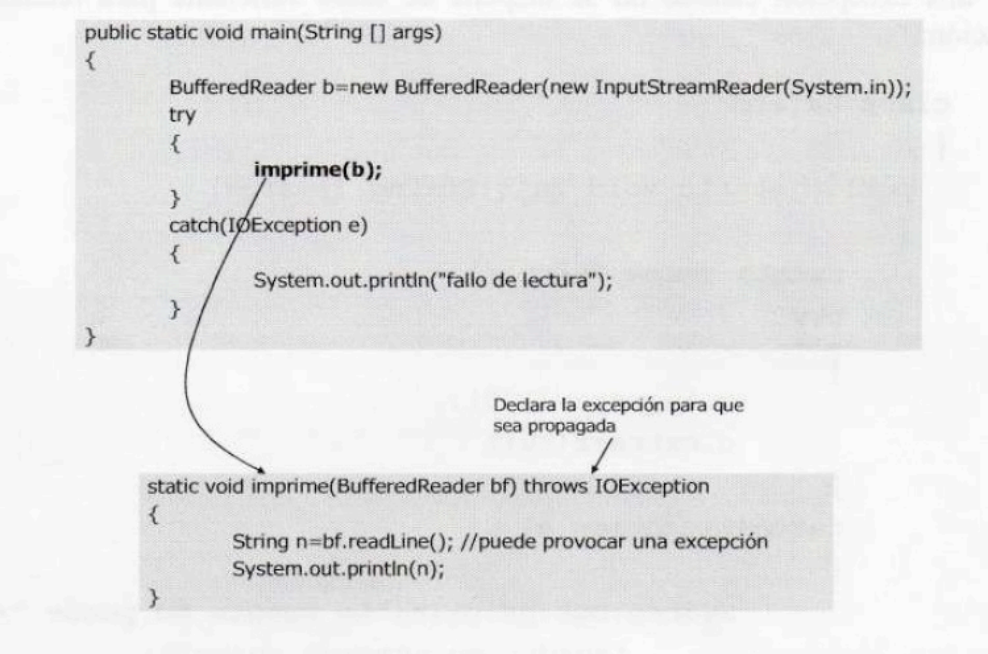
Esto demuestra que, **aún existiendo una instrucción para la salida del método (return), el bloque finally se ejecutará antes de que esto suceda.**

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.

Ejercicios de prueba:

1.- Prueba el programa siguiente:

```
public class Excepcion1{
    public static void main(String args[])
    {
        int numero[]=new int[5];
        numero[7]=0;
    }
}
```

Este programa genera una excepción al momento de su ejecución.
A continuación, agrega el código para el manejo de la excepción.

2.- Prueba el programa siguiente

```
class Excepcion2{
    void miMetodo(){
        int numero[]=new int[5];
        try{
                System.out.println("Accesando a una posicion fuera del vector");
                numero[7]=0;
```

```
            }
            catch (ArrayIndexOutOfBoundsException excep)
            {
                    System.out.println ("Ocurrio una excepcion");
            }
        }
}

public class PruebaExcepcion2{
        public static void main (String args[]){
                GeneraExcepcion2 objeto =new GeneraExcepcion2();
                objeto.miMetodo();
        }
}
```

¿En qué método se genera la excepción?
¿Qué método la captura?
¿Vuelve el control del programa al main, después de la captura?

3.- Prueba el programa siguiente

```
class Excepcion3{
        static void divide(){
                int num[]={4,8,16,32,64,128,256};
                int den[]={2,0,4,4,0,8,16};
                for (int i=0;i<num.length;i++){
                    try{
                            System.out.println(num[i]+ "/" + den[i]+"=" + num[i]/den[i]);}
                    catch(java.lang.ArithmeticException excepcion){
                            System.out.println("Dividiendo por cero");
                    }
                }
        }
}


public class PruebaExcepcion3{
        public static void main (String args[]){
                ExcepcionContinua.divide();
        }
}
```

¿Qué ocurre después de que el programa responde a un error?

4.- En un mismo segmento de código se pueden generar más de una excepción por diferentes motivos, por lo tanto dicho segmento puede tener un bloque catch para cada excepción. Completa los dos bloques match, el primero captura la excepción que se produce al dividir por cero y el segundo al acceder a una posición fuera del vector.

```
public class Excepcion4 {

   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 . . . . . . . {
                        . . . . .
                    }
        catch . . . . . . . {
                        . . . . .
                    }
      }
   }
}

public class PruebaExcepcion4 {
   public static void main (String args[]){
      Excepcion4.divide();
      System.out.println("FIN");
   }
}
```

5.- catch específicos y genéricos.

En el ejemplo siguiente se deberán completar los dos bloques catch, el primero atrapa específicamente la excepción que se produce cuando se accede fuera del límite del vector, el segundo atrapa cualquier excepción que se produzca en la superclase throwable.

```
public class Excepcion5 {

      public static void main(String args[]){
         int num[]= {4,8,16,32,64,128,256,512};
         int den[]= {2,0,4,4,0,8};
         for(int i=0;i<num.length;i++){
            try{
               System.out.println(num[i]+"/"+den[i]+"="+ num[i]/den[i]);
            }
                  catch . . . . . . . {
                        . . . . .
                            }
                    catch . . . . . . . {
                        . . . . .
                            }

         }
      } // fin de main
}
```

6.- Los bloques try se pueden anidar y así permitir diferentes niveles de errores, que serán manejados de diferentes formas. Algunos de estos errores son menores y pueden arreglarse fácilmente, pero otros son catastróficos y no se pueden corregir. Un método comúnmente empleado por los programadores es usar **un bloque externo try** para atrapar los **errores graves** y **bloques internos try para el control de los errores**

sencillos. Prueba el programa siguiente que usa bloques try anidados:

```
class Excepcion6 {
   public static void main(String args[]){
      int num[]= {4,8,16,32,64,128,256,512};
      int den[]= {2,0,4,4,0,8};

      try
      {
            for (int i=0;i<num.length;i++)
               {
                  try //try interno
                  {
                    System.out.println(num[i]+"/"+den[i]+"="+ num[i]/den[i]);
                  }
                  catch (ArithmeticException Excep)
                  {
                    System.out.println("División por cero "+ i);
                  }
               }
      } //try externo

      catch (Throwable Excep)
      {
         System.out.println("Ocurrio una excepcion fatal ");
      }

      System.out.println("El programa continu aqui");

   } // fin de main}
}
```

7.- Completa los dos bloques catch del ejemplo siguiente. El primero atrapa la excepción que se produce cuando se divide entre cero, el segundo atrapa la excepción que se produce cuando se accede a una posición fuera del vector.
Diseña el bloque finally que se ejecuta siempre tanto si se produce excepción como no. Este bloque contiene la orden: **System.out.println("Ejecutando código de limpieza");**

```
public class Excepcion7 {

   public static void generaExcepcion(int i){
      int t;
      int num[] = {2,4,6};
      System.out.println("Recibiendo "+ i);
      try
      {
         switch(i)
         {
               case 0: t=10/i; //division por cero
                  break;
               case 1:num[4]=4; //genera un error
                  break;
               case 2: return;
         }
      }
```

```
        catch . . . . . {
                    . . . . .
                        }
        catch . . . . . {
                    . . . . .
                        }

         finally
              {
                 . . . . . .
              }

} //fin de metodo
}//clase
```

SOLUCIÓN 1:

```
package excepcion1;

public class Excepcion1 {


   public static void main(String[] args) {
      int numero[]=new int[5];
      try{
         numero[7]=0;
         }
      catch(java.lang.ArrayIndexOutOfBoundsException Error){
         System.out.println("Se genero una excepción al acceder al vector");
      }
   }
}
```

SOLUCIÓN 2:

¿En qué método se genera la excepción?
¿Qué método la captura?
¿Vuelve el control del programa al main, después de la captura?

La excepción es generada y capturada en un método invocado por main() de nombre miMetodo().

Dado que la excepción fue capturada por el método, el control del programa no podrá volver a main.

SOLUCIÓN 3:

¿Qué ocurre después de que el programa responde a un error?

El programa responde a un error y luego continua con la ejecución, este es uno de los mayores beneficios del manejo de excepciones.

SOLUCIÓN 4:

```
package excepcion4;

public class Excepcion4 {

   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 "+i);
            }
            catch(java.lang.ArrayIndexOutOfBoundsException excepcion){
                System.out.println("Error al acceder el vector "+i);
            }
        }

    }
}
```

```
package excepcion4;

public class PruebaExcepcion4 {
    public static void main (String args[]){
        Excepcion4.divide();
        System.out.println("FIN");
    }
}
```

SOLUCIÓN 5:

Una claúsula catch que atrapa a la superclase throwable atrapará a cualquier excepción. Si se requiere atrapar excepciones con un catch general y además hacer un catch para una excepción específica, entonces se debe colocar la subclase y después la clase throwable.

```
package excepcion5;

public class Excepcion5 {

public static void main(String args[]){
    int num[]= {4,8,16,32,64,128,256,512};
    int den[]= {2,0,4,4,0,8};
    for(int i=0;i<num.length;i++){
        try{
            System.out.println(num[i]+"/"+den[i]+"="+ num[i]/den[i]);
        }
        catch (ArrayIndexOutOfBoundsException Excep)
        {
            System.out.println("Fuera de limite "+ i);
        }
        catch (Throwable Excep)
        {
            System.out.println("Ocurrio una excepcion generica "+ i);
        }
    }
} // fin de main
}
```

SOLUCIÓN 6:

try anidados

```
class Excepcion6 {
   public static void main(String args[]){
      int num[]= {4,8,16,32,64,128,256,512};
      int den[]= {2,0,4,4,0,8};

      try
      {
            for (int i=0;i<num.length;i++)
               {
                  try //try interno
                  {
                    System.out.println(num[i]+"/"+den[i]+"="+ num[i]/den[i]);
                  }
                  catch (ArithmeticException Excep)
                  {
                    System.out.println("División por cero "+ i);
                  }
               }
      } //try externo

      catch (Throwable Excep)
      {
         System.out.println("Ocurrio una excepcion fatal ");
      }

      System.out.println("El programa continu aqui");

   } // fin de main}
}
```

SOLUCIÓN 7:

7.- Completa los dos bloques catch del ejemplo siguiente. El primero atrapa la excepción que se produce cuando se divide entre cero, el segundo atrapa la excepción que se produce cunado se accede a una posición fuera del vector.
Diseña el bloque finally que se ejecuta siempre tanto si se produce excepción como no. Este bloque contiene la orden: **System.out.println("Ejecutando código de limpieza");**

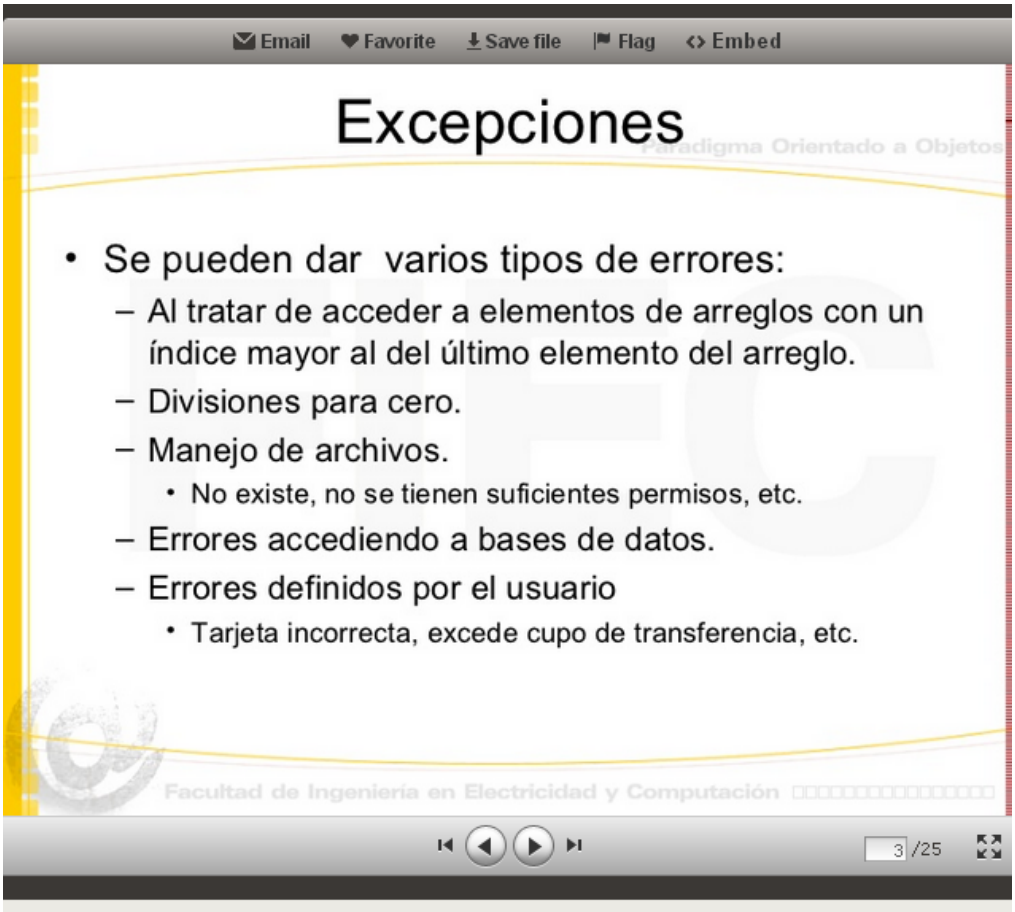
```
package excepcion7;

public class Excepcion7 {

   public static void generaExcepcion(int i){
      int t;
      int num[] = {2,4,6};
      System.out.println("Recibiendo "+ i);
      try
      {
```

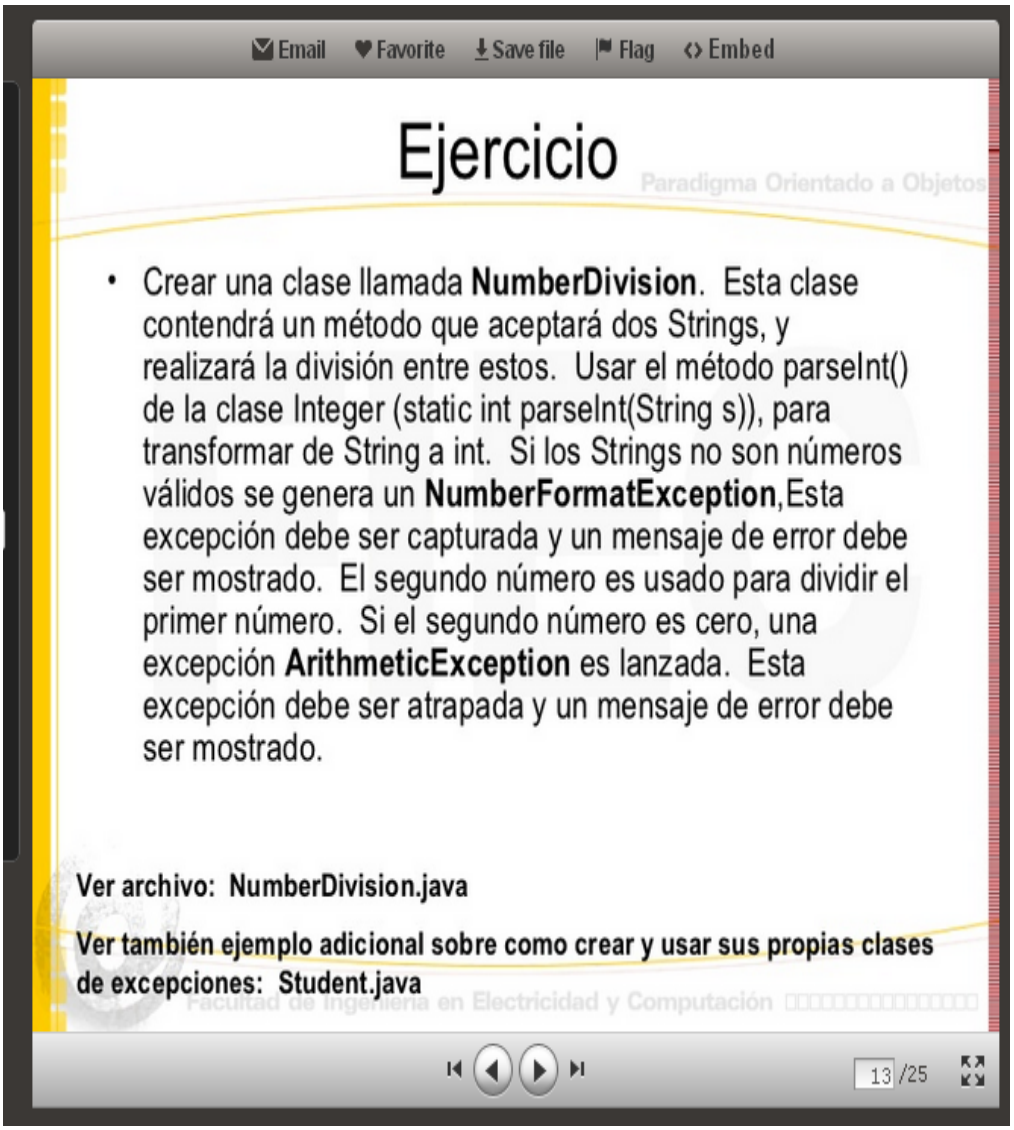
```
            switch(i)
            {
                  case 0: t=10/i; //division por cero
                     break;
                  case 1:num[4]=4; //genera un error
                     break;
                  case 2: return;
            }
         }
         catch(ArithmeticException exc)
         {
            System.out.println("No puede dividir entre cero");
            return; // regresa desde catch
         }
         catch(ArrayIndexOutOfBoundsException exc)
         {
            System.out.println(" No existe índice 4 en el vector");
         }
         finally
         {
            System.out.println("Ejecutando código de limpieza");
         }
      } //fin de metodo
}//clase
```

```
package excepcion7;

class PruebaExcepcion7{
      public static void main(String args[]){
                for (int i=0;i<4; i++)
                {
                     Excepcion7.generaExcepcion(i);
                     System.out.println();
                }
      }
}
```

Muchas veces, cuando se produce una excepción es necesario un mecanismo que limpie el estado del método antes de que el control pase a otra parte del programa. Por ejemplo, una excepción podría causar un error que termine el método actual, pero tal vez antes sea necesario cerrar un archivo o una conexión a red. En Java esto se puede hacer esto encerrando el código de limpieza dentro de un bloque **finally.**

```
class UseFinally{
        public static void generaExcepcion(int i){
            int t;
            int num[] = {2,4,6};
            System.out.println("Recibiendo "+ what);
            try
            {
                switch(i)
                {
                    case 0: t=10/i; //division por cero
                            break;
                    case 1:num[4]=4; //genera un error
                            break;
                    case 2: return;
                }
            }
            catch(ArithmeticException exc)
            {
                System.out.println("No puede dividir entre cero");
                return; // regresa desde catch
            }
            catch(ArrayIndexOutOfBoundsException exc)
            {
                System.out.println(" No hay elementos que coincidan");
            }
            finally
            {
                System.out.println("Ejecutando código de limpieza");
            }
        } //fin de metodo
}//clase


class PruebaUseFinally{
        public static void main(String args[]){
            for (int i=0;i<4; i++)
            {
                UseFinally.generaExcepcion(i);
                System.out.println();
            }
        }
}
```


Email
Favorite
Save file
Flag
Embed
Excepciones
Paradigma Orientado a Objetos
• Se pueden dar varios tipos de errores:
– Al tratar de acceder a elementos de arreglos con un índice mayor al del último elemento del arreglo.
– Divisiones para cero.
– Manejo de archivos.
• No existe, no se tienen suficientes permisos, etc.
– Errores accediendo a bases de datos.
– Errores definidos por el usuario
• Tarjeta incorrecta, excede cupo de transferencia, etc.
Facultad de Ingeniería en Electricidad y Computación
3 /25


Email
Favorite
Save file
Flag
Embed
Ejercicio
Paradigma Orientado a Objetos
• Crear una clase llamada **NumberDivision**. Esta clase contendrá un método que aceptará dos Strings, y realizará la división entre estos. Usar el método parseInt() de la clase Integer (static int parseInt(String s)), para transformar de String a int. Si los Strings no son números válidos se genera un **NumberFormatException**,Esta excepción debe ser capturada y un mensaje de error debe ser mostrado. El segundo número es usado para dividir el primer número. Si el segundo número es cero, una excepción **ArithmeticException** es lanzada. Esta excepción debe ser atrapada y un mensaje de error debe ser mostrado.
Ver archivo: NumberDivision.java
Ver también ejemplo adicional sobre como crear y usar sus propias clases de excepciones: Student.java
Facultad de Ingeniería en Electricidad y Computación
13 /25

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: (pasar a ordenador, añade la clase PruebaCuenta)

```
class Cajero
{
  public static void main (Striṅg [] 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 lOExceptíon 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 versión de *printStackTrace()* permite enviar el volcado de pila a un objeto *PrintStream* cualquiera, por ejemplo, un fichero log.

7 CLASES DE EXCEPCIÓN PERSONALIZADAS

Cuando un método necesita lanzar una excepción como forma 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 lOException 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: (pasar a ordenador)

1° Definición de la excepción propia.

```
class SaldoInsuficienteException extends  Exception{
  publíc SaldoInsuficieteExceptíon (String mensaje){
     super(mensaje);
   }
}
```

2° 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 ("nümeros
          rojos");
     else
          saldo- =c;
   }

  public double getSaldo(){
     return saldo;
   }
```

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

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

EJERCICIOS:

8.- Sea la clase siguiente:

```
class Piscina
{
  private int nivel;
  public final int MAX_NIVEL;

  public Piscina(int max)
  {
    if (max<0) max=0;
    MAX_NIVEL=max;
  }

  public int getNivel()
  { return nivel; }

  public void vaciar(int cantidad)
  { nivel=nivel-cantidad; }

  public void llenar(int cantidad)
  { nivel=nivel+cantidad; }
}
```

A.- Modifica los métodos vaciar(…) y llenar(…) de manera que lancen una excepción cuando al vaciar el nivel quede por debajo de cero y cuando al llenar el nivel quede por encima de `MAX_NIVEL`.

B.- Declara la clase **PiscinaCliente()** que contiene los métodos:

public static void operacionesPiscina(Piscina p)

En este método se capturan las excepciones con try-catch, El código que puede producir una excepción es un for de 1 a 3 que llena la piscina con un número generado aleatoriamente (p.llenar((int)(Math.random()*100));), mostrando luego su nivel, a continuación se vacía la piscina por el mismo procedimiento, mostrando luego también su nivel.

Si se produce una excepción se emite mensaje de texto asociado a la excepción más el nivel de la piscina.
El método main instancia una piscina **p** y ejecuta **operacionesPiscina(p).**

9.- Declara la clase **MailException** que extienda de la clase Exception. En esta clase declaramos el atributo **descripción** de tipo String y dos constructores:

public **MailException**(String mensaje); En este constructor se pasa el mensaje al método constructor de la clase Exception. Y la cadena "" al atributo.

public **MailException**(String mensaje, String descripcion); En este constructor se pasa el mensaje al método constructor de la clase Exception. Y la cadena descripcion al atributo.

Además declara los métodos:

public String **getDescripcion**();
public void **setDescripcion**(String descripcion)

Declara la clase **PruebaMail** que contiene:

El método **public static void validarEnteroPositivo(int valor)…..** que utiliza la excepción personalizada, ésta se lanza si valor es menor que cero.

El método main() en el que se leen desde teclado 5 valores y los valida.

10.- Piscina con excepción personalizada

A.- Declara la clase **PiscinaNivelExcepcion** que extienda de la clase Exception. En esta clase declaramos el atributo **nivel** de tipo int y un constructor:

public **PiscinaNivelExcepcion** (String descripción, int valor); En este constructor se pasa la cadena descripción al método constructor de la clase Exception y el entero al atributo **nivel**.

B.- Declara la clase **Piscina**:

```
class Piscina
{
      private int nivel;
      public final int MAX_NIVEL;

      public Piscina(int max)
      {
            if (max<0) max=0;
            MAX_NIVEL=max;
      }
      public int getNivel()
      {
            return nivel;
      }

      public void vaciar(int cantidad) throws
      PiscinaNivelException
      {
```

```
        /*Si (nivel-cantidad <0), se lanza la excepción.
        La descripción será "Vaciado excesivo" y nivel será igual a nivel-cantidad.
        En otro caso nivel será igual a nivel-cantidad. */
    }
    public void llenar(int cantidad) throws
    PiscinaNivelException
    {
        /*Si (nivel+cantidad > MAX_NIVEL), se lanza la excepción.
        La descripción será "Vaciado excesivo" y nivel será igual a nivel+cantidad.
        En otro caso nivel será igual a nivel+cantidad. */

    }

}
```

C.- Declara la clase PiscinaCliente() que contiene:

El método **public static void** operacionesPiscina(Piscina p) **throws** PiscinaNivelException que sólo contiene el código que puede producir la excepción, en caso de producirse se propaga.

El método main() que captura las excepciones con try-catch.. El bloque try ejecuta operacionesPiscina(Piscina p) y en el bloque catch se emite el mensaje de texto asociado a la excepción más el nivel de la piscina.

11.- Escribe un programa que lea dos números por teclado, numerador y denominador respectivamente. A continuación se debe comprobar que numerador es menor que 100 y denominador mayo de -5, si esto no es así se lanzará una excepción propia o personalizada de nombre ExcepciónIntervalo. Luego se calculará el cociente y se mostrará por pantalla. Debemos considerar que aparte de la excepción propia también se puede producir una división entre cero y también que al pedir los números por teclado introduzcamos caracteres no numéricos.