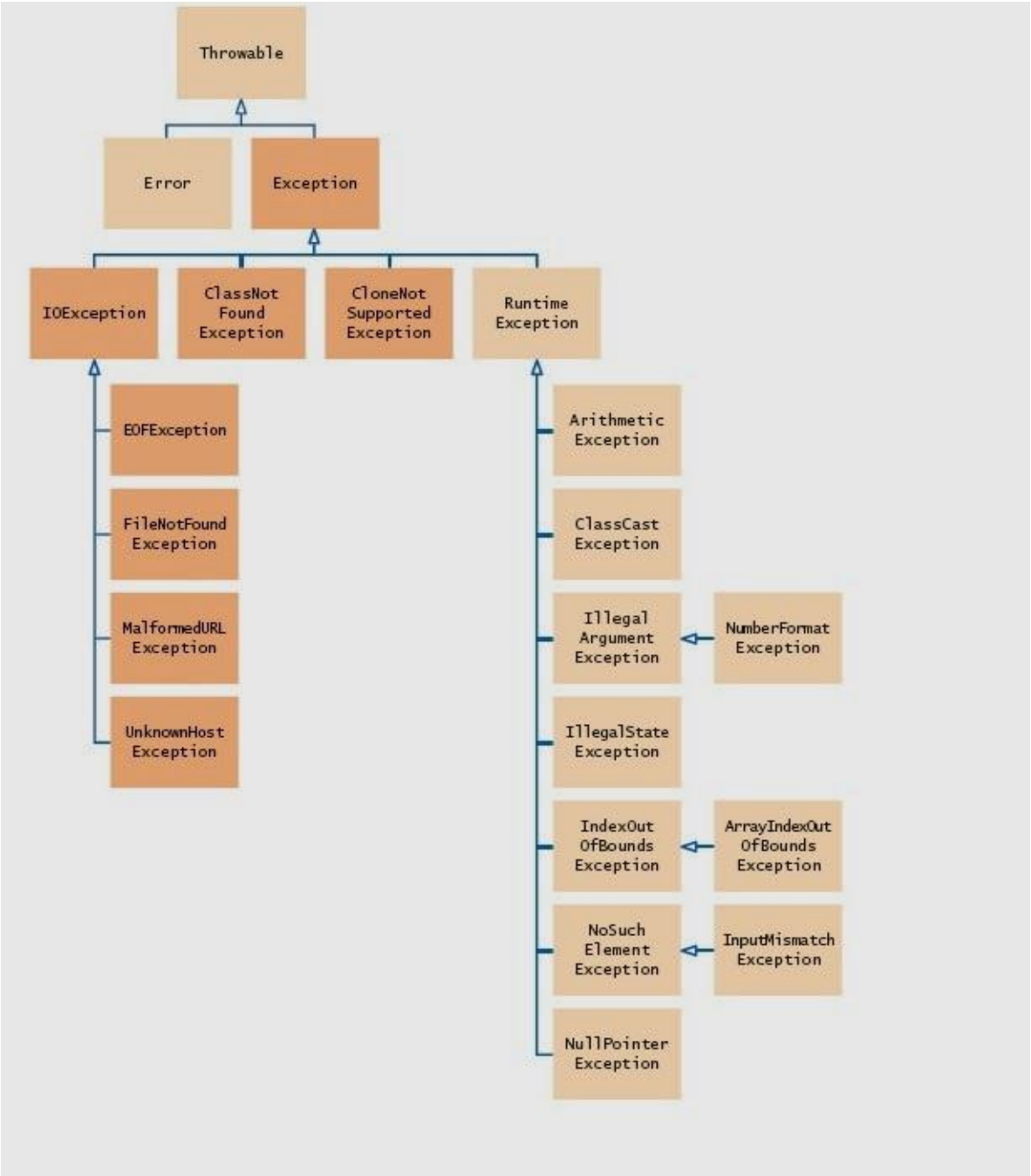
Excepciones



Definición.

Una excepción es un objeto que se genera automáticamente cuando se produce un acontecimiento circunstancial que impide el normal funcionamiento del programa:

- Dividir por cero
- No encontrar un determinado fichero
- Utilizar un puntero nulo en lugar de una referencia a un objeto

…

El objeto generado "excepción" contiene información sobre el acontecimiento ocurrido y transmite esta información al método desde el que se ha generado la excepción.

La ocurrencia de estas situaciones excepcionales provocará la terminación no controlada del programa o aplicación.

Las excepciones estándar

En Java las situaciones que pueden provocar un fallo en el programa se denominan excepciones.

Las excepciones pueden originarse de dos modos:

- El programa hace algo ilegal (caso normal)

El siguiente código de ejemplo origina una excepción de división por cero:

```
public class PruebaExcepcion {
    public static void main( String[] a ) {
        int i=0, j=0, k;
        k = i/j;    // Origina un error de division-by-zero
    }
 }
```

Si compilamos y ejecutamos esta aplicación Java, obtendremos la siguiente salida por pantalla:

```
> javac PruebaExcepcion.java
> java PruebaExcepcion
     java.lang.ArithmeticException: / by zero
```

```
        at PruebaExcepcion.main(melon.java:5)
```

Las excepciones predefinidas, como por ej. ArithmeticException, se conocen como excepciones runtime. Las excepciones en tiempo de ejecución ocurren cuando el programador no ha tenido cuidado al escribir su código.

Por ejemplo: cuando se sobrepasa la dimensión de un array, se lanza una excepción *ArrayIndexOutOfBounds*.

Cuando se hace uso de una referencia a un objeto que no ha sido creado se lanza la excepción *NullPointerException*.

Estas excepciones le indican al programador que tipos de fallos tiene el programa y que debe arreglarlo antes de proseguir.

Actualmente, como todas las excepciones son eventos runtime, sería mejor llamarlas excepciones irrecuperables. Esto contrasta con las excepciones que generamos explícitamente, que suelen ser mucho menos severas y en la mayoría de los casos podemos recuperarnos de ellas. Por ejemplo, si un fichero no puede abrirse, preguntamos al usuario que nos indique otro fichero; o si una estructura de datos se encuentra completa, podremos sobreescribir algún elemento que ya no se necesite.

- El programa explícitamente genera una excepción.

Para generar explícitamente una excepción se ejecutará la sentencia **throw**.

La sentencia throw tiene la siguiente forma:

throw ObjetoExcepction;

El objeto ObjetoException es un objeto de una clase Excepcion o que hereda (ver siguiente unidad) de la clase Exception. Para que un método en Java, pueda lanzar excepciones será necesario indicarlo expresamente.

void MetodoQueLanzaExcepcion() throws Exception1, Exception2,....

Se pueden definir excepciones propias, no hay por qué limitarse a las predefinidas. Para definir una excepción será necesario con extender la clase Exception (herencia) y proporcionar la funcionalidad extra que requiera el tratamiento de esa excepción.

```
public      void  metodoQueLanzaDivisionPorCero  ( )  throws  DivisioCero{
…
if (…)     throw  new DivisionCero( );
…
}
.
..metodoQueUtilizaMetodoQueLanza(…){
…
try{
obj.metodoQueLanzaDivisionCero();
}catch( DivisionCero    objExcepcion){
….
}

….
```

Habría que definir DivisionCero

```
public class DivisionCero extends Exception{

…

}
```

Existe toda una jerarquía de clases derivada de la clase base *Exception* fig1

Las excepciones en Java son siempre objetos de alguna clase derivada de la clase base *Exception*. Existen también los errores internos que son objetos de la clase *Error* que no estudiaremos. Ambas clases *Error* y *Exception* son clases derivadas de la clase base abstracta *Throwable*.

Ejemplos de excepciones.

Por formato

```
String str="  12 ";
int numero=Integer.parseInt(str);
```

Si se introducen caracteres no numéricos, o no se quitan los espacios en blanco al principio y al final del string, se lanza una excepción *NumberFormatException.*

El mensaje que aparece en la ventana nos indica:

El tipo de excepción: N*umberFormatException,*

La función que la ha lanzado: *Integer.parseInt()* que se llama dentro de *main*(por ejemplo)

Por objeto no inicializado

Habitualmente, en un mensaje a un objeto no inicializado

```
    public static void main(String[] args) {
                String str;
            str.length();
                //...
   }
```

El compilador se queja con el siguiente mensaje "variable str might not have been initilized". En otras ocasiones, se lanza una excepción del tipo *NulPointerException.*

```
class MiCanvas....{
                Grafico grafico;
    public void paint(...){
                grafico.dibuja();
                //...
    }
//...
}
```

Si al llamarse a la función *paint*, el objeto *grafico* no ha sido inicializado con el valor devuelto por **new** al crear un objeto de la clase *Grafico* o de alguna de sus clases derivadas, se lanza la excepción *NullPointerException*apareciendo en la consola el siguiente texto.

Exception occurred during event dispatching:
java.lang.NullPointerException
…

Entrada/salida

En otras situaciones el mensaje de error aparece en el momento en el que se compila el programa. Así, cuando intentamos leer un carácter del teclado, llamamos a la función

```
System.in.read();
```

Cuando compilamos el programa, nos aparece un mensaje de error que no nos deja proseguir.

```
unreported exception: java.io.IOException; must be caught or declared to
be thrown
```

Gestión de excepciones

Una excepción es una condición anormal que surge en una secuencia de código durante la ejecución de un programa. O sea, es un error en tiempo de ejecución.

Excepciones y errores

1. Fundamentos

- Cuando surge una condición excepcional se crea un objeto que representa la excepción, y se *envía* al método que ha provocado la excepción.

Este método puede gestionar la excepción él mismo o bien pasarla al método llamante. En cualquier caso, la excepción es *capturada* y procesada en algún punto.

- La gestión de excepciones usa las palabras reservadas **try**, **catch**, **throw**,**throws** y **finally**.

```
try {
  // bloque de código
}
catch (TipoExcepcion1 obEx){
  // gestor de excepciones para TipoExcepcion1
}
catch (TipoExcepcion2 obEx){
  // gestor de excepciones para TipoExcepcion2
}
// ...
finally {
  // bloque de código que se ejecutara antes de
  // que termine el bloque try
}
```

Captura de las excepciones

Empecemos por solucionar el error que se produce en el programa durante la compilación. Tal como indica el mensaje que genera el compilador, se ha de poner la sentencia *System.in.read();* en un bloque **try**…**catch**, del siguiente modo.

```
try {
     System.in.read();
}catch (IOException ex) {  }
```

Para solucionar el error que se produce en el programa durante su ejecución, se debe poner la llamada a *Integer.parseInt* en el siguiente bloque **try**…**catch**.

```
String str="  12 ";
int numero;
try{
numero=Integer.parseInt(str);
}catch(NumberFormatException ex){
System.out.println("Non número");
}
```

En el caso de que el string *str* contenga caracteres no numéricos como es éste el caso, el número 12 está acompñado de espacios en blanco, se produce una excepción del tipo *NumberFormatException* que es capturada y se imprime el mensaje “No es un número”.

En vez de un mensaje propio se puede imprimir el objeto *e* de la clase*NumberFormatException*

```
try{
                //...
}catch(NumberFormatException e){
System.out.println(e);
}
```

La clase base *Throwable* de todas las clases que describen las excepciones, redefine el método *toString()*, que devuelve el nombre de la clase que describe la excepción acompañado del mensaje asociado, que en este caso es el propio string *str*.

```
java.lang.NumberFormatException:    12
```

Podemos extraer dicho mensaje mediante la función miembro *getMessage*, del siguiente modo

```
try{
                //...
}catch(NumberFormatException e){
System.out.println(e.getMessage());
}
```

Manejando varias excepciones

Ejemplo: Un programa que divida dos números.

Supongamos que dos números que se introducen en dos controles de edición. Se obtiene el texto de cada uno de los controles de edición que se guarda en dos strings. En esta situación se pueden producir dos excepciones *NumberFormatException*, si se introducen caracteres no numéricos y *ArithmeticException* si se divide entre cero.

```
public class EjemploExcepcion {
    public static void main(String[] args) {
        String str1="12";
               String str2="0";
        String respuesta;
               int numerador, denominador, cociente;
        try{
            numerador=Integer.parseInt(str1);
            denominador=Integer.parseInt(str2);
            cociente=numerador/denominador;
            respuesta=String.valueOf(cociente);
        }catch(NumberFormatException ex){
            respuesta="Se han introducido caracteres no numéricos";
        }catch(ArithmeticException ex){
            respuesta="División entre cero";
        }
        System.out.println(respuesta);
    }
}
```

Las sentencias susceptibles de lanzar una excepción se sitúan en un bloque **try**…**catch**. Si el denominador es cero, se produce una excepción de la clase *ArithmeticException* en la expresión que halla el cociente, que es inmediatamente capturada en el bloque **catch** que maneja dicha excepción, ejecutándose las sentencias que hay en dicho bloque. En este caso se guarda en el string *respuesta* el texto “División entre cero”.

Hay veces en las que se desea estar seguro de que un bloque de código se ejecute se produzcan o no excepciones. Se puede hacer esto añadiendo un bloque **finally** después del último **catch**. Esto es importante cuando accedemos a archivos, para asegurar que se cerrará siempre un archivo se produzca o no un error en el proceso de lectura/escritura.

```
               try{
                              //Este código puede generar una excepción
               }catch(Exception ex){
                              //Este código se ejecuta cuando se produce
una excepción
               }finally{
```

```
                                //Este código se ejecuta se produzca o no
una excepción
                }
```

2 Tipos de excepción

Todos los tipos de excepción son subclase de **Throwable**. Esta clase tiene dos subclases:

1. **Exception**: Se usa para las excepciones que deberán capturar los programas de usuario.

Esta clase tiene como subclase a **RuntimeException**, que representa excepciones definidas automáticamente por los programas (división por 0, índice inválido de matriz, etc). Además tiene otras subclases como`ClassNotFoundException`, `InterruptedException`, etc.

1. **Error**: Excepciones que no se suelen capturar en condiciones normales.

Suelen ser fallos catastróficos no gestionados por nuestros programas. Ejemplo: desbordamiento de la pila.

8.3 Excepciones Predefinidas

Las excepciones predefinidas y su jerarquía de clases es la que se muestra en la figura:

3 Excepciones no capturadas

```
public class EjemploExcepcion {
  static void subroutine() {
    int d = 0;
    int a = 10 / d;
  }
  public static void main(String args[]) {
    System.out.println("Antesxc1.subroutine");
    EjemploExcepcion.subroutine();
    System.out.println("Despuesxc1.subroutine");
  }
}
```

- Cuando la máquina virtual Java detecta la división por 0 construye un objeto de excepción y lanza la excepción. Esto detiene la ejecución de EjemploExcepcion, ya que no hemos incluido un *gestor de la excepción*.
- Cualquier excepción no tratada por el programa será tratada por el ***gestor por defecto***.

En el caso de J2SE, el gestor por defecto muestra la excepción y el trazado de la pila en la salida estándar, y termina el programa.

Salida del programa

```
Antes de EjemploExcepcion.subroutine()
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at EjemploExcepcion.subroutine(EjemploExcepcion.java:4)
        at EjemploExcepcion.main(EjemploExcepcion.java:8)
```

try y catch

```
public class Exc2 {
  public static void main(String args[]) {
    int d, a;

    try { // controla un bloque de cÃ³digo.
      d = 0;
      a = 42 / d;
      System.out.println("Estoe imprimirá.");
    }
    catch (ArithmeticException e) {// captura el error de división
      System.out.println("Divisióncero.");
    }
    System.out.println("Despuésa sentencia catch.");
  }
}
```

El objetivo de una sentencia **catch** bien diseñada será resolver la condición de excepción y continuar.

Otro ejemplo:

```
// Gestiona una excepción y continua.
import java.util.Random;
public class ManejadorExcepcion {
  public static void main(String args[]) {
    int a=0, b=0, c=0;
    Random r = new Random();
    for(int i=0; i<32000; i++) {
      try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
      } catch (ArithmeticException e) {
        System.out.println("Divisioncero.");
        a = 0; // asigna a la variable el valor 0 y continua
      }
      System.out.println("a a);
    }
  }
```

```
}
```

Descripción de una excepción

La clase **Throwable** sobreescribe el método **toString()** de la clase **Object**, devolviendo una cadena con la descripción de la excepción.

```
catch (ArithmeticException e) {
  System.out.println("Excepcion e);
  a = 0; // hacer a=0 y continuar
}
```

Salida producida cuando se produce la excepción:

```
Excepcion: java.lang.ArithmeticException
```

5 Cláusula catch múltiple

En algunos casos un bloque de código puede activar más de un tipo de excepción. Usaremos varios bloques **catch**.

El siguiente programa produce una excepción si se ejecuta sin parámetros y otra distinta si se ejecuta con un parámetro.

Ejemplo

```
Public class MultiCatch {
  public static void main(String args[]) {
    try {
      int a = args.length;
      System.out.println("a+ a);
      int b = 42 / a;
      int c[] = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Division0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Indicea de limites: " + e);
    }
    System.out.println("Despuesbloque try/catch.");
  }
}
```

Al ordenar los bloques **catch**, las subclases de excepción deben ir antes que la superclase (en caso contrario no se ejecutarán nunca y dará error de compilación por código no alcanzable).

Ejemplo

```
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
    }
    catch(Exception e) {
      System.out.println("catch cualquier tipo de excepción.");
    }
    /* Este catch nunca se ejecutará */
    catch(ArithmeticException e) { // ERROR - no alcanzable
      System.out.println("Estoa se ejecutará.");
    }
  }
}
```

Sentencias try anidadas

- Una sentencia **try** puede estar incluida dentro del bloque de otra sentencia **try**.
- Cuando un **try** no tiene **catch** para una excepción se busca si lo hay en el**try** más externo, y así sucesivamente.

Ejemplo

```
public class TrysAnidados {
  public static void main(String args[]) {

    try {
          int a = args.length;
          /* Si no hay ningún argumento en la línea de órdenes
            se generará una excepción de división por cero. */
          int b = 42 / a;
          System.out.println("a= a);
                  try { // bloque try anidado
                      /* Si se utiliza un argumento en la línea de
órdenes
                           se generará una excepción de división por
cero. */
                      if(a==1) a = a/(a-a); // división por cero
                           /* Si se le pasan dos argumentos en la lÃnea
de órdenes,
                          se genera una excepción al sobrepasar los
límites
                          del tamaño de la matriz. */
                      if(a==2) {
                            int c[] = { 1 };
                            c[42] = 99; // genera una excepción de fuera
de límites
                        }
                  } catch(ArrayIndexOutOfBoundsException e) {
```

```
                            System.out.println("Indicea de limites: " +
e);
                    }
    } catch(ArithmeticException e) {
      System.out.println("División0: " + e);
    }
  }
}
```

Ejemplo:

/* Las sentencias try pueden estar implícitamente anidadas a través de llamadas a métodos.
*/

```
class TrysAnidadosImplicitamente {
  public static void metodoConTryAnidado(int a) {
    try { // bloque try anidado
      /* Si se utiliza un argumento  igual a 1 en la línea de órdenes, la
         siguiente sentencia efectúa división por cero */
      if(a==1) a = a/(a-a); // división por zero
      /* Si se le pasan un argumento igual a dos en la línea de órdenes,
         Se sobrepasan los límites de la matriz */
      if(a==2) {
        int c[] = { 1 };
        c[42] = 99; // genera una excepción de fuera de límites
      }
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Indicea de limites: " + e);
    }
  }
  public static void main(String args[]) {
    try {
      int a = args.length;
      /* Si no hay ningún argumento en la línea de órdenes, la
         siguiente sentencia generará una excepción de división
         por cero */
      int b = 42 / a;
      System.out.println("a+ a);
      metodoConTryAnidado(a);
    } catch(ArithmeticException e) {
      System.out.println("División0: " + e);
    }
  }
}
```

6. Lanzar excepciones explícitamente: throw

Usando la sentencia **throw** es posible hacer que el programa lance una excepción de manera explícita: `throw objetoThrowable;`

El `objetoThrowable` puede obtenerse mediante:

1. El parámetro de una sentencia **catch**.
2. Con el operador **new** .

Ejemplo

```
public class ThrowDemo {
  public static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("Capturaro de demoproc.");
      throw e; // relanza la excepción
    }
  }
  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      System.out.println("Nueva captura: " + e);
    }
  }
}
```

El flujo de ejecución se detiene tras la sentencia **throw** (cualquier sentencia posterior no se ejecuta).

Salida del anterior ejemplo:

```
Captura dentro de demoproc.
Nueva captura: java.lang.NullPointerException: demo
```

7. Sentencia throws

Sirve para enumerar una lista de los tipos de excepción que un método puede lanzar.

- Debe usarse para proteger los métodos que usan un determinado método, si tal método lanza la excepción pero no la maneja.
- Es necesario usarla con todas las excepciones excepto **Error** y**RuntimeException** y sus subclases.
- Si las excepciones que lanza el método y no maneja no se ponen en**throws** se producirá error de compilación.

Forma general de declaración de método con throws

```
tipo metodo(lista_de_parametros) throws lista_de_excepciones
```

```
{
  // cuerpo del metodo
}
```

Ejemplo

```
// Programa erróneo que no compila
public class ThrowsDemo {
  static void throwOne() {
    System.out.println("DentrohrowOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    throwOne();
  }
}
```

El método que use al del throws debe capturar todas las excepciones listada con el throws.

Ejemplo

```
// Programa correcto
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("DentrohrowOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Capturae);
    }
  }
}
```

8. Sentencia finally

- Se puede usar esta sentencia en un bloque **try-catch** para ejecutar un bloque de código despues de ejecutar las sentencias del **try** y del **catch**.
- Se ejecutará tanto si se lanza, como si no una excepción, y aunque no haya ningún **catch** que capture esa excepción.
- Podáa usarse por ejemplo para cerrar los archivos abiertos.

Ejemplo

```
public class FinallyDemo {
  public static void procA() {// Lanza una excepción fuera del método
    try {
      System.out.println("DentrorocA");
```

```
      throw new RuntimeException("demo");
    } finally {
      System.out.println("Sentencially de procA");
    }
  }
  public static void procB() {// Ejecuta la sentencia return
                       // dentro del try
    try {
      System.out.println("DentrorocB");
      return;
    } finally {
      System.out.println("Sentencially de procB");
    }
  }
  public static void procC() {// Ejecuta un bloque try normalmente
    try {
      System.out.println("DentrorocC");
    } finally {
      System.out.println("Sentencially de procC");
    }
  }
  public static void main(String args[]) {
    try {procA();
    } catch (Exception e) {
      System.out.println("Excepciónurada");
    }
    procB(); procC();
  }
}
```

Salida del anterior programa

```
Dentro de procA
Sentencia finally de procA
ExcepciÃ³n capturada
Dentro de procB
Sentencia finally de procB
Dentro de procC
Sentencia finally de procC
```

9. Subclases de excepciones propias

- Sirven para crear tipos propios de excepción que permitan tratar situaciones específicas en una aplicación.
- Para ello solo hay que definir una subclase de **Exception**.
- Estas subclases de excepciones no necesitan implementar nada.
- La clase **Exception** no define ningún método, solo hereda los de**Throwable**.

Ejemplo

```
class MyException extends Exception {
  private int detail;
  MyException(int a) {
    detail = a;
  }
  public String toString() {
    return "MyException[" + detail + "]";
  }
}
class ExceptionDemo {
  static void compute(int a) throws MyException {
    System.out.println("Ejecutaute(" + a + ")");
    if(a > 10)
      throw new MyException(a);
    System.out.println("Finalizaciónal");
  }
  public static void main(String args[]) {
    try {
      compute(1);
      compute(20);
    } catch (MyException e) {
      System.out.println("Capturae);
    }
  }
}
```

La salida de este programa será:

```
Ejecuta compute(1)
Finalización normal
Ejecuta compute(20)
Captura MyException[20]
```