

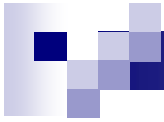
Tema 3.6

Excepciones

Análisis y Diseño de Software

2º Ingeniería Informática

Universidad Autónoma de Madrid



Indice

■ Introducción.

- Funcionamiento.
- Tipos.
- Creando nuevos tipos de excepciones.
- Tratamiento de excepciones.
- Lanzando excepciones.



Introducción: Excepciones

■ ¿Qué son?

- Evento que ocurre durante la ejecución normal, con el que se asocia un objeto *excepción* para notificar el evento
- Interrumpe el flujo normal del programa

■ ¿Cuándo ?

- Únicamente en situaciones excepcionales:
 - Errores al convertir tipos de datos
 - Limitaciones físicas
 - Fallos de dispositivos
 - Errores de programación
 - Etc.



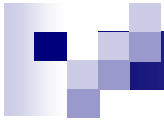
Ejemplo: Sin tratamiento de errores

```
public void readFile() {  
    abrir el fichero;  
    determinar el tamaño del fichero;  
    reservar esa cantidad de memoria;  
    leer el fichero en memoria;  
    cerrar el fichero;  
}
```



Con tratamiento de errores “estilo C”

```
public errorCodeType readFile {  
    inicializar errorCode = 0;  
    abrir el fichero;  
    if (ficheroAbierto) {  
        determinar el tamaño del fichero;  
        if (obtuvimosLaLongitud) {  
            reservar esa cantidad memoria;  
            if (obtuvimosBastanteMemoria) {  
                leer el fichero en memoria;  
                if (lecturaFallo) errorCode = -1;  
            }  
            else errorCode = -2;  
        }  
        else errorCode = -3;  
        cerrar el fichero;  
        if (elFicheroNoCerró && errorCode == 0)  
            errorCode = -4;  
        else errorCode = errorCode & -4;  
    }  
    else errorCode = -5;  
    return errorCode;  
}
```



Con Excepciones

```
public void readFile {  
    try {  
        abrir el fichero;  
        determinar el tamaño del fichero;  
        reservar esa cantidad de memoria;  
        leer el fichero en memoria;  
        cerrar el fichero;  
    }  
    catch (fileOpenFailed) { doSomething; }  
    catch (sizeDeterminationFailed) { doSomething; }  
    catch (memoryAllocationFailed) { doSomething; }  
    catch (readFailed) { doSomething; }  
    catch (fileCloseFailed) { doSomething; }  
}
```



Excepciones

- Error → Se genera un objeto
- Objeto de excepción, contiene:
 - Tipo de error
 - Mensaje de error
 - Estado del programa cuando ocurrió
- También se puede generar mediante código
→ Lanzamiento de excepciones con **throw**

```
if (inesperado()) throw new Exception("error");
```



Manejador: bloque `try/catch/finally`

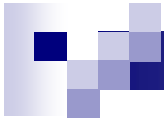
```
try {  
    ...    // bloque a proteger, asociándolo con manejadores  
}  
catch (TipoExcepcion1 e1) {  
    ...    // manejo de la excepción e1  
}  
catch (TipoExcepcion2 e2) {  
    ...    // manejo de la excepción e2  
}  
...  
catch (TipoExcepcionN eN) {  
    ...    // manejo de la excepcion e_N  
}  
finally {  
    ...    // opcional: si está se ejecuta  
}          // p.ej., para liberar recursos
```


Ejemplo

```
try {  
    ...  
    obj.f ();  
    ...  
}  
  
catch (EdadNegativa ex) {  
    ...  
}
```

```
void f () throws EdadNegativa {  
    ...  
    if (edad < 0)  
        throw new EdadNegativa(persona, edad);  
    ...  
}
```

The diagram illustrates a try-catch block in a Java program. The try block contains a call to `obj.f ();`. The catch block is for the `EdadNegativa` exception. The method `f ()` is shown to throw `EdadNegativa` when `edad < 0`. Arrows indicate the flow of control: one arrow points from the `obj.f ();` line in the try block to the `throw` statement in the `f ()` method, and another arrow points from the closing brace of the `f ()` method to the catch block.



Ejemplo: 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;  
    }  
}
```



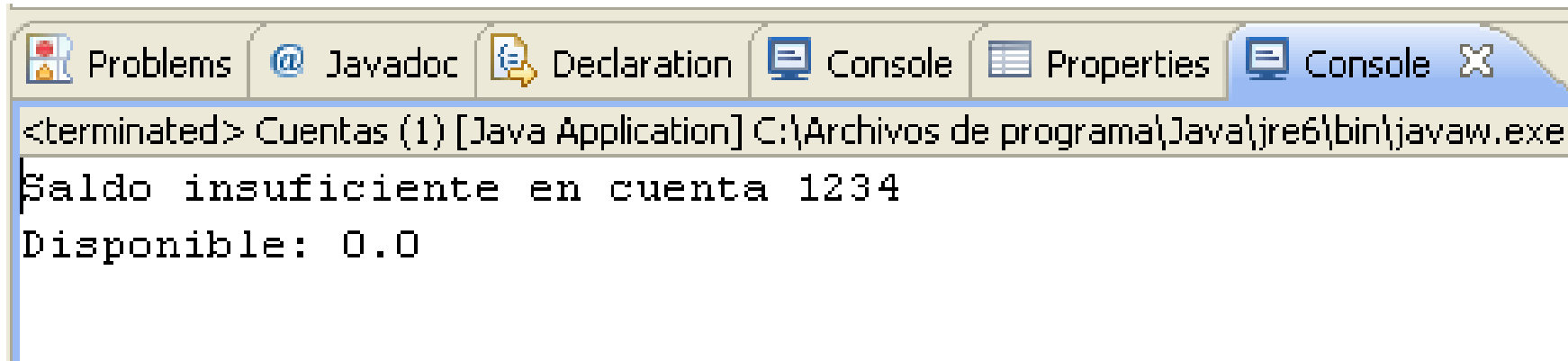
Definición de clases para Excepciones

```
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";  
    }  
}
```

Captura de Excepciones

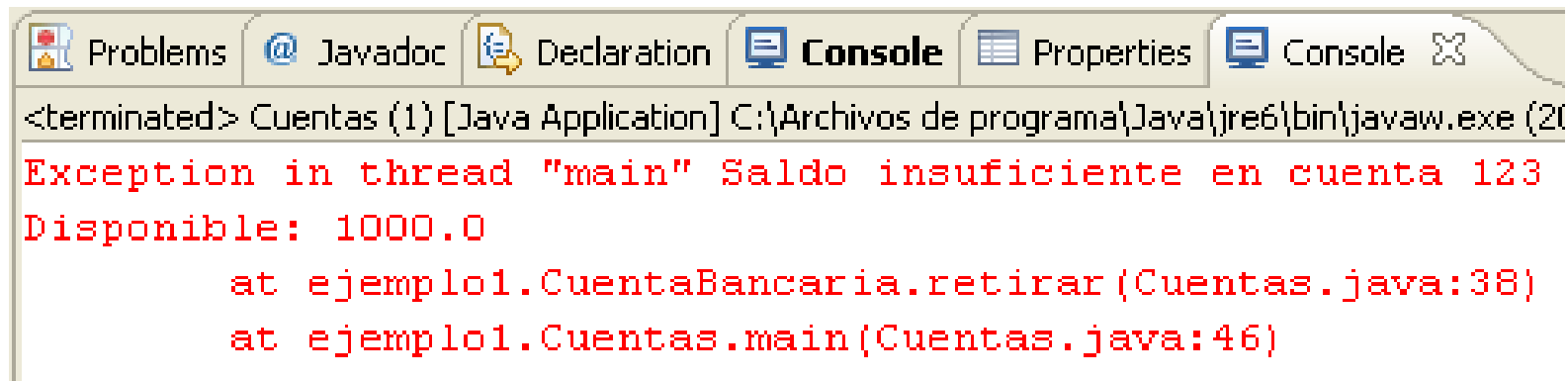
*Se ejecuta el primer
catch de tipo compatible*

```
public static public void main (String args[]) {  
    try {  
        new CuentaBancaria (1234,0).retirar (100000);  
    }  
    catch (SaldoInsuficiente excep) {  
        System.out.println (excep);  
    }  
    catch (CuentaBloqueada excep) {  
        System.out.println (excep);  
    }  
}
```



¿...y si no se capturan?

```
public static void main (String args[])  
    throws CuentaBloqueada, SaldoInsuficiente {  
    CuentaBancaria cuenta = new CuentaBancaria (123,1000);  
    cuenta.retirar (2000);  
}
```



The screenshot shows a Java IDE's console window. The title bar includes tabs for Problems, Javadoc, Declaration, Console (selected), Properties, and another Console. The console text is as follows:

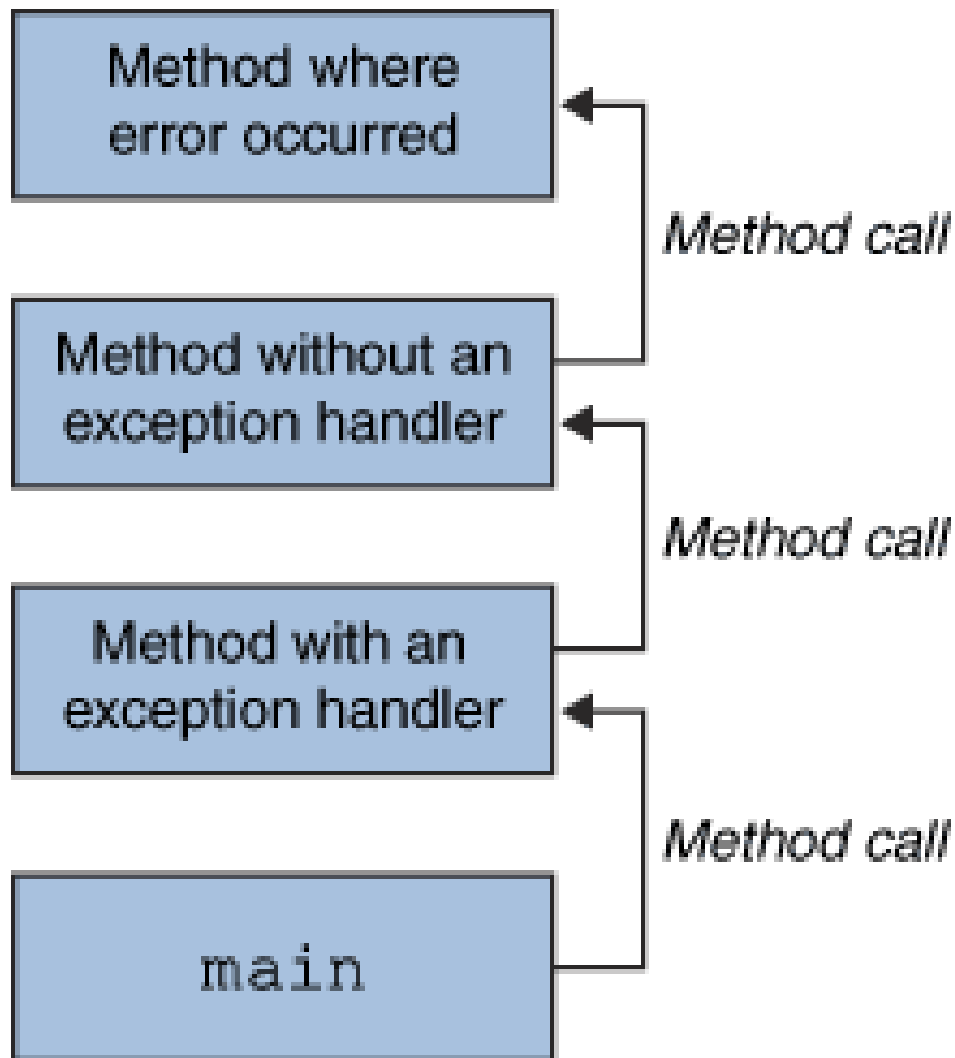
```
<terminated> Cuentas (1) [Java Application] C:\Archivos de programa\Java\jre6\bin\javaw.exe (21  
Exception in thread "main" Saldo insuficiente en cuenta 123  
Disponible: 1000.0  
    at ejemplo1.CuentaBancaria.retirar(Cuentas.java:38)  
    at ejemplo1.Cuentas.main(Cuentas.java:46)
```



Indice

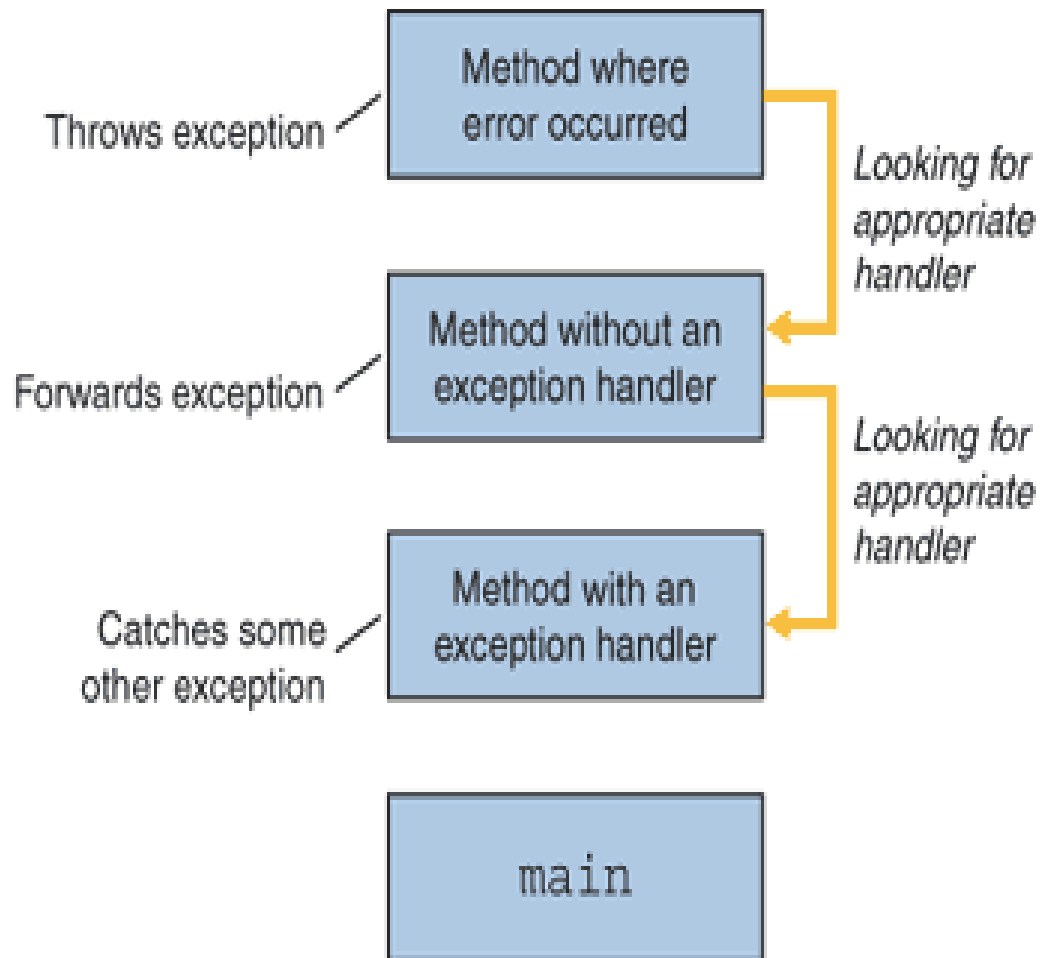
- Introducción.
- **Funcionamiento.**
- Tipos.
- Creando nuevos tipos de excepciones.
- Tratamiento de excepciones.
- Lanzando excepciones.

Pila de llamadas

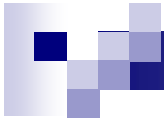


- Pila de llamadas en el hilo de ejecución actual
- El último método es el que produjo la excepción
- Indica la línea de código en cada método de la pila

Captura de la excepción

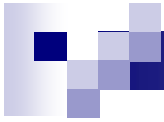


- Se pueden capturar (catch) distintos tipos de excepciones en distintos puntos (manejadores de excepciones)
- La búsqueda del manejador apropiado comienza en el punto más cercano al error
- Continúa bajando en la pila de llamadas



Requisito: catch o throws

- En Java es necesario capturar las excepciones que se puedan producir (catch)
- O alternativamente declarar en el método que puede emitirlas (throws)
- El código no compilará si no se cumple
- Excepción a la regla: ***Runtime Exceptions*** (ej: División por cero)



Indice

- Introducción.
- Funcionamiento.
- **Tipos.**
 - Creando nuevos tipos de excepciones.
 - Tratamiento de excepciones.
 - Lanzando excepciones.



Clasificación

Origen/Controladas	No controladas (Unchecked Exceptions)	Excepciones Controladas (Checked Exceptions)
Origen externo	Errores Externos	La causa original es externa (e.j: usuario, red, permisos, etc.)
Origen interno	Errores Internos (Bugs/ Runtime Exceptions)	No tienen sentido como excepción (condicionales)



Checked Exceptions

■ Excepciones *controladas*

- Es el tipo más habitual, y las únicas que requieren `catch` o `throws`
- Son condiciones excepcionales que un programa debería anticipar y controlar. Ej:
FileNotFoundException
- Son subclase de ***Exception*** (Salvo `RuntimeException`)



No controladas

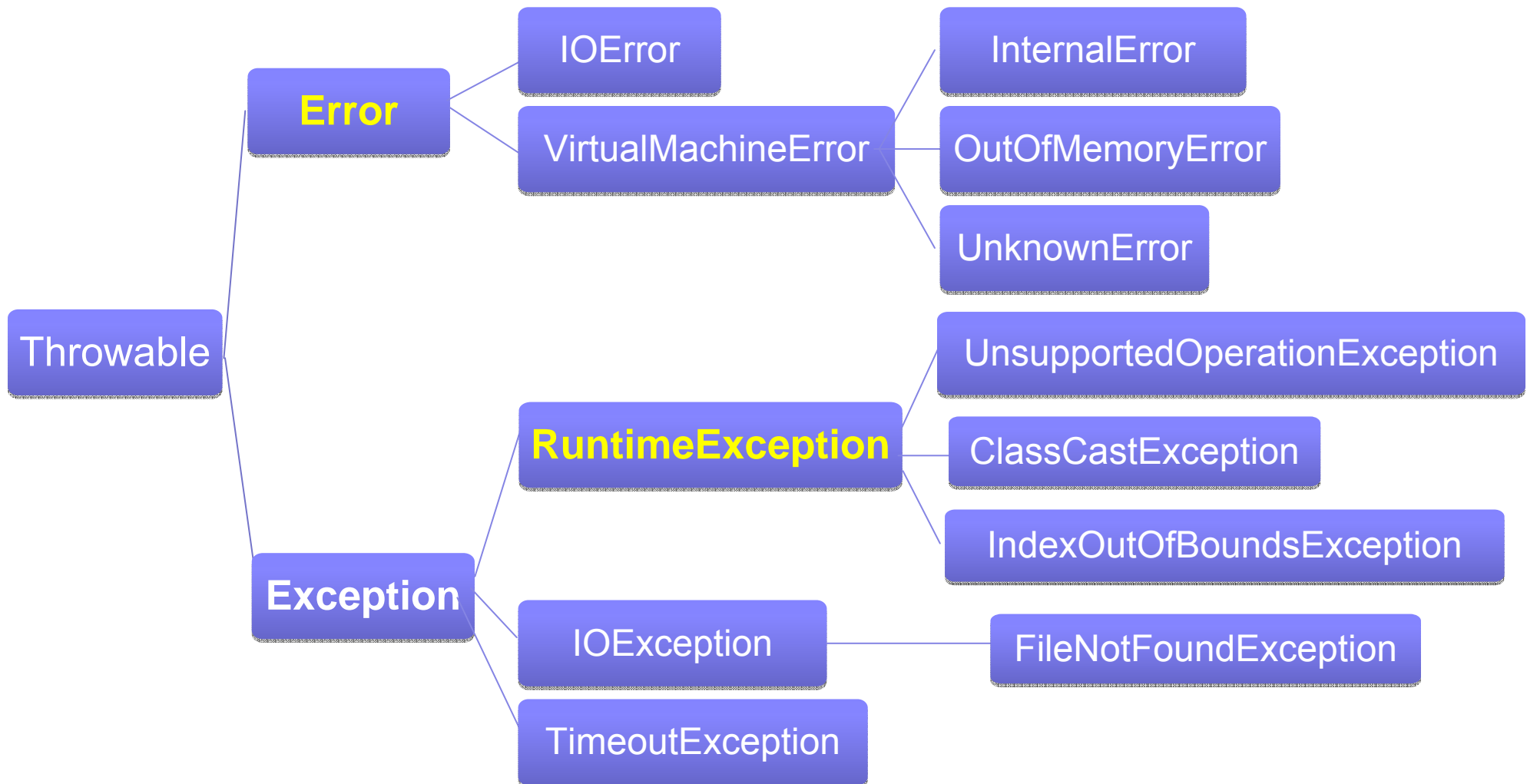
- No están sujetas al requisito catch o throws
- Condiciones excepcionales, normalmente no anticipables y de difícil recuperación
- Subtipos:
 - Errores externos:
 - Subclases de `Error`
 - Ej: *IOException* al fallar el disco duro en una lectura
 - Errores internos o Runtime Exceptions
 - Subclases de *RuntimeException*
 - Ej: *NullPointerException*



Jerarquía de excepciones y errores

- Todos heredan de *Throwable* (que es subclase directa de Object)
- La jerarquía ayuda a capturar varios tipos de situaciones
- Clase **Error** No se suelen capturar, son errores graves
StackOverflowError, OutOfMemoryError, UnknownError, fallo hardware,
errores internos de JVM, errores en carga e inicialización del programa, ...
- Clase **Exception**
separadas de los errores para permitir darles un tratamiento muy general
`try {...} catch (Exception e) { /* trata cualquier Exception */ }`
- Clase **RuntimeException** (subclase directa de Exception)
No son subclase de Error, al no ser de origen externo:
Fallos de programación, habitual capturarlas e informar para depuración
Caso especial de Exception: **no es obligatorio catch ni throws**
¿mejor subclase de Throwable directamente?

Jerarquía de excepciones y errores

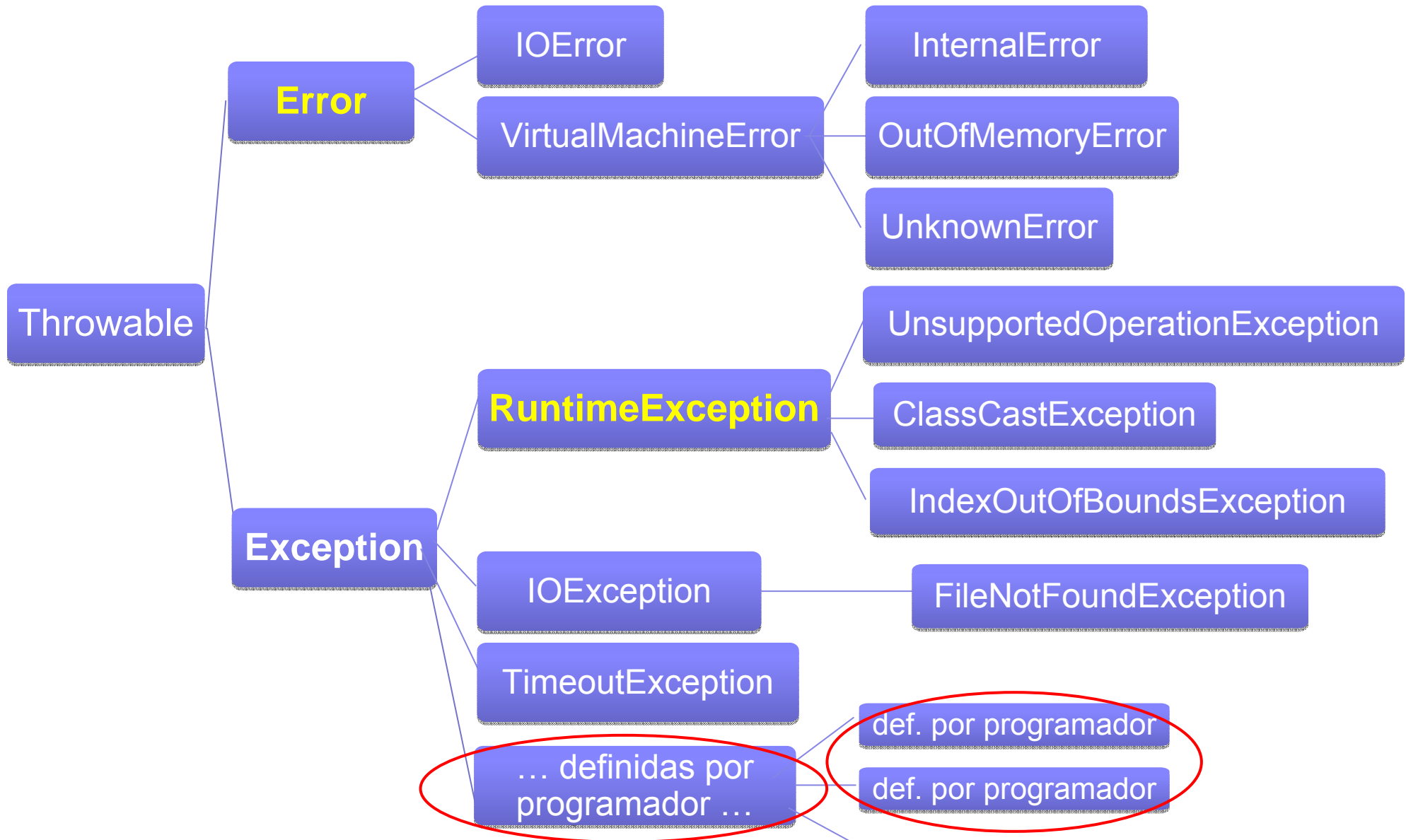




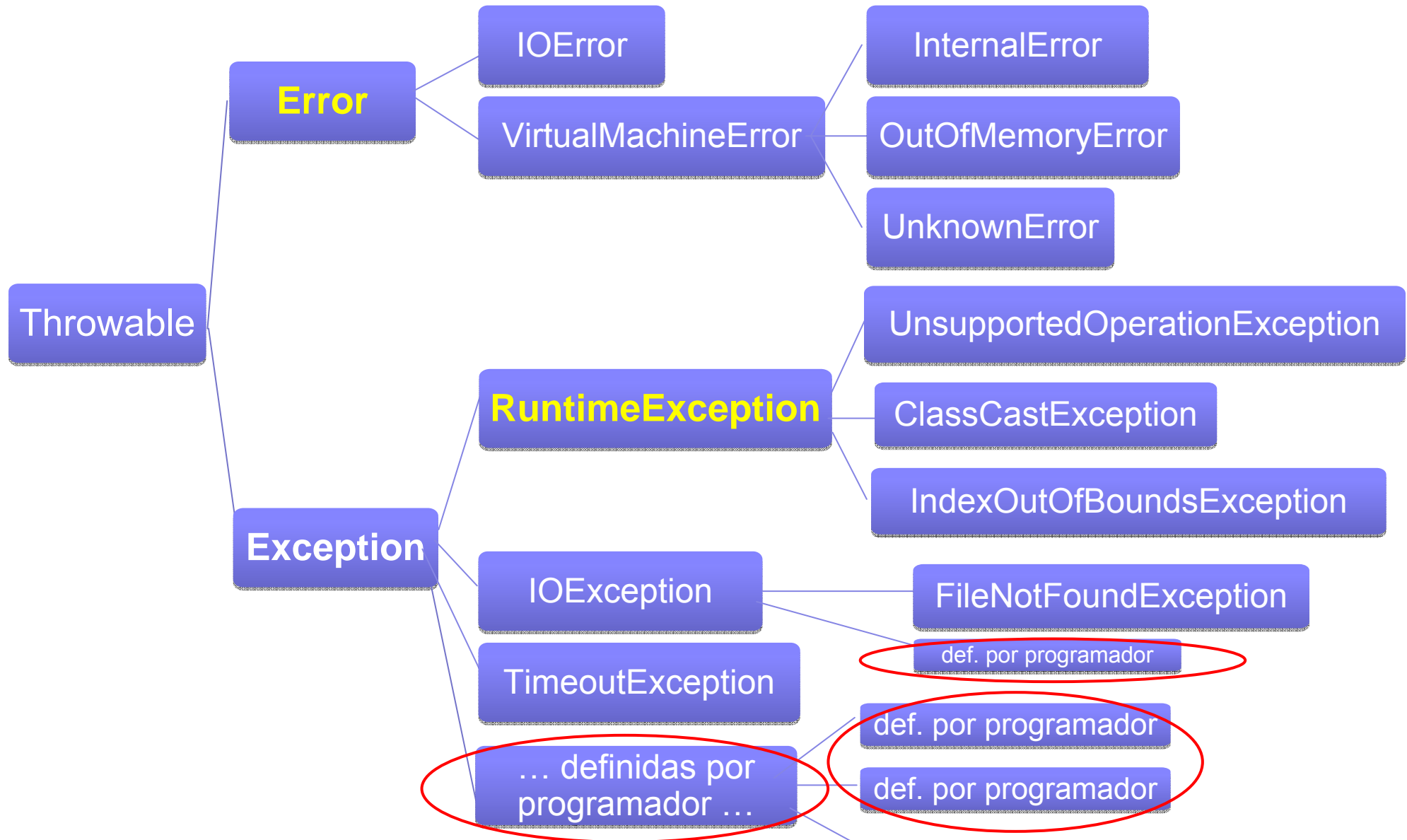
Indice

- Introducción.
- Funcionamiento.
- Tipos.
- **Creando nuevos tipos de excepciones.**
- Tratamiento de excepciones.
- Lanzando excepciones.

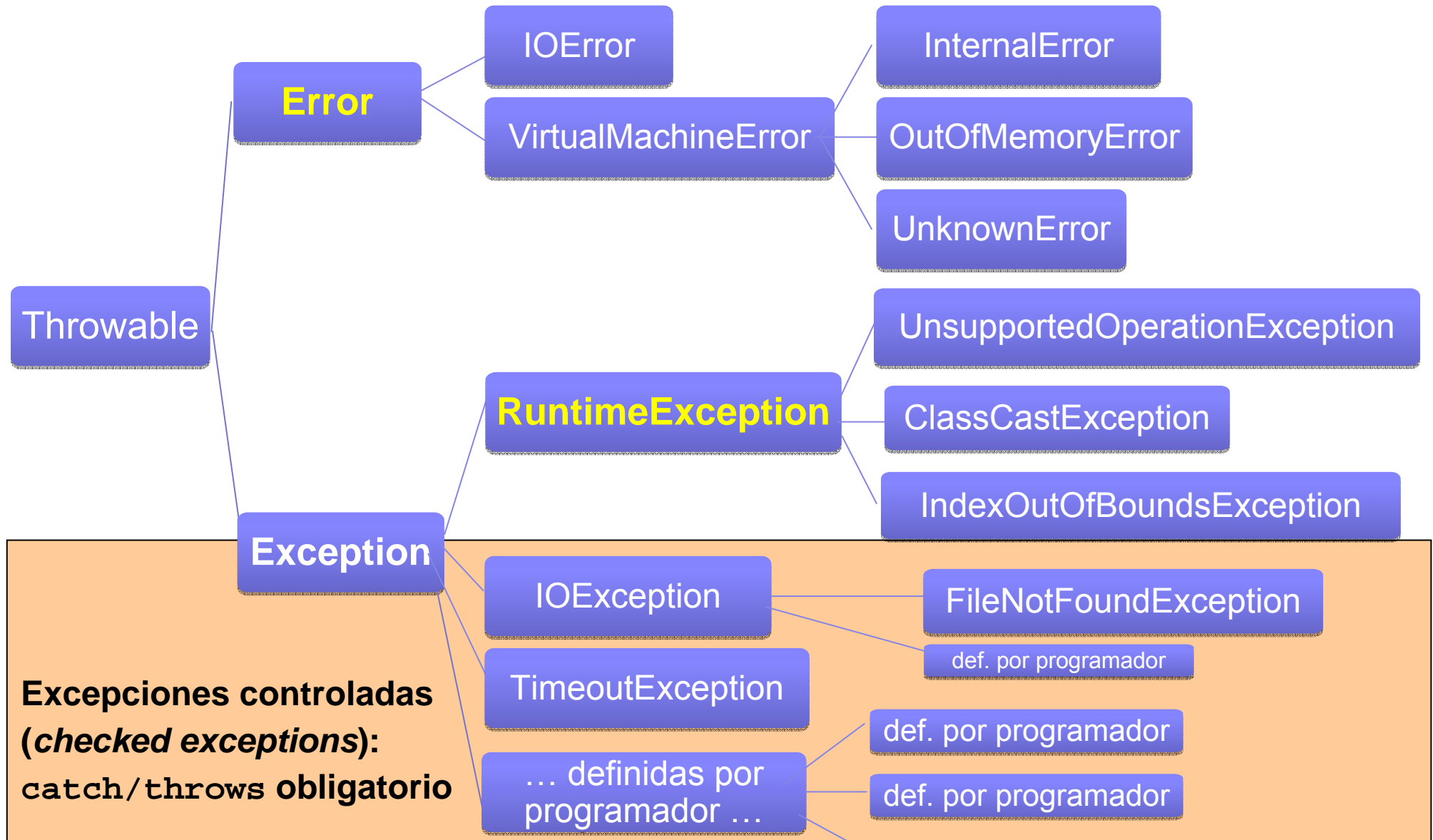
Ampliación de excepciones y errores



Ampliación de excepciones y errores



Ampliación de excepciones y errores





Creando nuevas excepciones

■ Regla básica:

- ¿el *cliente* puede, razonablemente, recuperarse de la excepción?
 - Sí → Excepción controlada (*checked exception*)
 - No → No controlada (Error o RuntimeException)

- No usar subclases de RuntimeException simplemente para evitar el requisito de indicar las excepciones que un método puede emitir (`throws`)



Indice

- Introducción.
- Funcionamiento.
- Tipos.
- Creando nuevos tipos de excepciones.
- **Tratamiento de excepciones.**
- Lanzando excepciones.



Ejemplo: tratamiento de excepciones

```
private List<Integer> lista;  
PrintWriter salida = null;  
...
```

```
System.out.println("Inicio Try");  
salida = new PrintWriter(new FileWriter("d/out.txt"));  
for (int i=0; i<=lista.size(); i++) {  
    salida.println("lista[" + i + "] = " + lista.get(i));  
}
```

- Tratamiento de excepciones
 - El código principal cambia poco o nada
 - Evitamos complicarlo con los detalles de comprobación de casos excepcionales
 - El tratamiento de errores y excepciones al final de cada bloque que pueda provocarlas

Ejemplo: tratamiento de excepciones

```
private List<Integer> lista;  
PrintWriter salida = null;  
...  
try {  
    System.out.println("Inicio Try");  
    salida = new PrintWriter(new FileWriter("d/out.txt"));  
    for (int i=0; i<=lista.size(); i++) {  
        salida.println("lista[" + i + "] = " + lista.get(i));  
    }  
} ...  
//tratar  
//excepciones
```

- Posibles excepciones
 - Específicas
 - Al crear "d/out.txt"
 - Generales
 - Al escribir en salida
 - Al acceder a la lista (elemento inexistente)
 - Error de programación



Ejemplo: tratamiento de excepciones

```
private List<Integer> lista;
PrintWriter salida = null;
...
try {
    System.out.println("Inicio Try");
    salida = new PrintWriter(new FileWriter("d/out.txt"));
    for (int i=0; i<=lista.size(); i++) {
        salida.println("lista[" + i + "] = " + lista.get(i));
    }
} catch (FileNotFoundException e) {
    System.err.println("FileNotFoundException: "+e.getMessage());
    //relanzamos una excepción que habremos declarado aparte
    throw new MiExcepcion(e);
}
```




Ejemplo: tratamiento de excepciones

```
private List<Integer> lista;
PrintWriter salida = null;
...
try {
    System.out.println("Inicio Try");
    salida = new PrintWriter(new FileWriter("d/out.txt"));
    for (int i=0; i<=lista.size(); i++) {
        salida.println("lista[" + i + "] = " + lista.get(i));
    }
} catch (FileNotFoundException e) {
    System.err.println("FileNotFoundException: "+e.getMessage());
    //relanzamos una excepción
    throw new MiExcepcion(e);
}
// el compilador todavía da error:
// unhandled exception type IOException
```



Ejemplo: tratamiento de excepciones

```
private List<Integer> lista;
PrintWriter salida = null;
...
try {
    System.out.println("Inicio Try");
    salida = new PrintWriter(new FileWriter("d/out.txt"));
    for (int i=0; i<=lista.size(); i++) {
        salida.println("lista[" + i + "] = " + lista.get(i));
    }
} catch (FileNotFoundException e) {
    System.err.println("FileNotFoundException: "+e.getMessage());
    //relanzamos una excepcion
    throw new MiExcepcion(e);
} catch (IOException e) {
    System.err.println("captura IOException: " + e.getMessage());
}
```




Ordenación de cláusulas `catch`

- ❑ Las excepciones se capturan siguiendo el orden en que escribimos las cláusulas `catch`:
 - Primero escribimos las más específicas (excepciones que no son subclases de las anteriores)
 - Después las más generales (puede que alguna de sus subclases haya sido tratada en un `catch` previo)
 - El compilador lo comprueba y detecta errores
- ❑ En el ejemplo:
 - *FileNotFoundException* (más específica) se trata al lanzando una excepción propia de la aplicación con el operador `throw` (sin `s`, no es lo mismo que `throws`)
 - *IOException* hace de caso más general

Mejoras futuras de catch en JDK 7

- Se pueden tratar distintos tipos de excepción en un solo catch



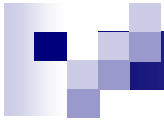
```
catch (FileNotFoundException | SecurityException ex) {  
    logger.log(ex);  
    throw new MiExcepcion (ex);  
    // Nota SecurityException no es obligatoria en  
    // PrintWriter(File f), al ser no comprobada  
}
```

Tratamiento post-excepciones y finalización

```
private List<Integer> lista;
PrintWriter salida = null;
...
try {
    System.out.println("Inicio Try");
    salida = new PrintWriter(new FileWriter("d/out.txt"));
    for (int i=0; i<=lista.size(); i++) {
        salida.println("lista[" + i + "] = " + lista.get(i));
    }
} catch (FileNotFoundException e) {
    ...
} catch (IOException e) {
    ...
} finally {
    if (salida!=null) salida.close();
}
```

El bloque **finally** *siempre* se ejecuta, haya habido excepción o no, siempre.

Es útil para liberar recursos y dejar el proceso bien terminado.



Bloque `finally`

- ❑ El bloque `finally` se ejecuta tras salir del bloque `try`
 - En el ejemplo, la variable `salida` se inicializa fuera a `null` para comprobar si llegó a abrirse el archivo
- ❑ En el bloque `try` podría haber un `return`, `break`, `continue`, o incluso provocar excepciones no previstas ...
- ❑ El bloque `finally` siempre se ejecuta
 - Es útil usar `try` con `finally` incluso si no se prevén excepciones.
- ❑ Nota: El bloque `finally` podría provocar nuevas excepciones, en cuyo caso puede que se ejecute sólo parcialmente

Ejemplo 2: copiando archivos (sin throws)

```
void copiarConRiesgo() {  
    InputStream in = null;  
    OutputStream out = null;  
    ...  
    in = new FileInputStream("in.txt");  
    out = new FileOutputStream("out.txt");  
    byte[] buffer = new byte[1024];  
    int n;  
    while ((n = in.read(buffer)) >= 0) {  
        out.write(buffer, 0, n);  
    }  
    ...  
    in.close();  
    out.close();  
}
```

*Su declaración en
java.io indica que
pueden provocar
IOException*

No compila correctamente sin
declarar, con **throws**, que se
pueden producir excepciones de
tipo **IOException**
que dejamos sin tratar

Ejemplo 2: copiando archivos (sin try)

```
void copiarConRiesgo() throws IOException {  
    InputStream in = null;  
    OutputStream out = null;  
    ...  
        in = new FileInputStream("in.txt");  
        out = new FileOutputStream("out.txt");  
        byte[] buffer = new byte[1024];  
        int n;  
        while ((n = in.read(buffer)) >= 0) {  
            out.write(buffer, 0, n);  
        }  
    ...  
        in.close();  
        out.close();  
    }  
}
```

Así compila correctamente, pero
no es buena solución ignorar todas
las excepciones y errores internos
simplemente poniendo **throws**

Ejemplo 2: copiando archivos (try sin catch)

```
void copiarConRiesgo() throws IOException {  
    InputStream in = null;  
    OutputStream out = null;  
    try {  
        in = new FileInputStream("in.txt");  
        out = new FileOutputStream("out.txt");  
        byte[] buffer = new byte[1024];  
        int n;  
        while ((n = in.read(buffer)) >= 0) {  
            out.write(buffer, 0, n);  
        }  
    } finally {  
        in.close();  
        out.close();  
    }  
}
```

Aunque compile, tampoco es buena solución `try/finally` sin cláusulas `catch`.

Se evita leer/escribir si no se abrió bien el archivo correspondiente, pero si no se abrió, dará error al cerrarlo `NullPointerException`

Ejemplo 2: copiando archivos (try con catch)

```
void copiarConRiesgo() throws IOException {  
    InputStream in = null;  
    OutputStream out = null;  
    try {  
        in = new FileInputStream("in.txt");  
        out = new FileOutputStream("out.txt");  
        byte[] buffer = new byte[1024];  
        int n;  
        while ((n = in.read(buffer)) >= 0) {  
            out.write(buffer, 0, n);  
        }  
    } catch (IOException e) {  
        // tratar la excepción  
    } finally {  
        if (in != null) in.close();  
        if (out != null) out.close();  
    }  
}
```

Las cláusulas `catch` son necesarias para tratar la excepción antes de saltar al bloque `finally`

Y también hemos evitado que al cerrar se produzca `NullPointerException`

Ejemplo 2: copiando archivos (necesita throws)

```
void copiarConRiesgo() throws IOException {  
    InputStream in = null;  
    OutputStream out = null;  
    try {  
        in = new FileInputStream("in.txt");  
        out = new FileOutputStream("out.txt");  
        byte[] buffer = new byte[1024];  
        int n;  
        while ((n = in.read(buffer)) >= 0) {  
            out.write(buffer, 0, n);  
        }  
    } catch (IOException e) {  
        // tratar la excepción  
    } finally {  
        if (in != null) in.close();  
        if (out != null) out.close();  
    }  
}
```

A pesar de las cláusulas `catch` sigue siendo necesario `throws` porque los `close` del bloque `finally` también pueden producir `IOException`

Ejemplo 2: copiando archivos (`try` dentro de `finally`)

```
...  
} finally {  
    if (in != null) {  
        try {  
            in.close();  
        } catch (IOException e) {  
            // tratar la excepción del close  
        }  
        // repetiríamos lo mismo para out.close  
    }  
} // fin del primer try/finally
```

La excepción de `close` significaría un error muy grave de entrada/salida, así que poco o nada se podría hacer, salvo informar, pero ahora ya no necesitamos declarar `throws IOException`

(Parece excesivo tratamiento de excepciones para este caso, pero se trata de aprender lo que se puede hacer cuando sea necesario).



Ejemplo 2: copiando archivos (no necesita throws)

```
void copiarSinRiesgo() {  
    InputStream in = null;  
    OutputStream out = null;  
    try {  
        in = new FileInputStream("in.txt");  
        out = new FileOutputStream("out.txt");  
        byte[] buffer = new byte[1024];  
        int n;  
        while ((n = in.read(buffer)) >= 0) {  
            out.write(buffer, 0, n);  
        }  
    } catch (IOException e) {  
        // tratar la excepción  
    } finally {  
        cerrarIgnorandoExcepcion(in);  
        cerrarIgnorandoExcepcion(out);  
    }  
}
```



Ejemplo 2: copiando archivos (close sin throws)

```
void cerrarIgnorandoExcepcion(Closeable c) {  
    if (c != null) {  
        try {  
            c.close();  
        } catch (IOException e) {  
            // tratar la excepción del close  
        }  
    }  
}
```

El interfaz `Closeable` exige el método:

```
void close() throws IOException
```

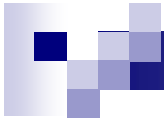
`FileInputStream` y `FileOutputStream` implementan `Closeable`



Mejoras futuras de `try` en JDK 7

- Para usar recursos, admite una sintaxis mejorada que cierra automáticamente los recursos
- Evita usar `finally` sólo para cerrar recursos y no requiere declarar las variables fuera del `try` para inicializarlas a `null`
- El recurso ha de implementar la interfaz `AutoCloseable`
- Si en `try` se lanza una excepción (no capturada) y en el `finally` implícito, se suprime la última `Throwable.getSuppressed()`

```
try (PrintWriter salida =  
    new PrintWriter(new FileWriter("d/out.txt"))) {  
    ...  
}  
catch (...) {  
    ...  
}
```



Indice

- Introducción.
- Funcionamiento.
- Tipos.
- Creando nuevos tipos de excepciones.
- Tratamiento de excepciones.
- **Lanzando excepciones.**



Especificando excepciones lanzables

- Clausula throws después de la declaración del método/constructor y antes de la implementación

```
public void escribeLista()  
        throws IOException, ArrayIndexOutOfBoundsException {  
}
```

- También se puede indicar en Interfaces o métodos abstractos
- Las no controladas no es necesario indicarles.

```
public void escribeLista() throws IOException{  
}
```

Solo IOException, ya que ArrayIndexOutOfBoundsException es no controlada



Lanzando excepciones

- Se lanzan con `throw objetoExcepcion`
- `objetoExcepcion` se crea con `new ClaseExcepcion()`
(hay constructores con argumentos)
- Normalmente se usan subclasses de `Exception`
- Las subclasses de `Error` no se suelen capturar ni lanzar
(fallos graves de la máquina virtual, etc.)

```
public Object pop() {  
    Object obj;  
    if (size == 0) {  
        throw new EmptyStackException();  
    }  
    obj = objectAt(size - 1);  
    setObjectAt(size - 1, null);  
    --size;  
    return obj;  
}
```

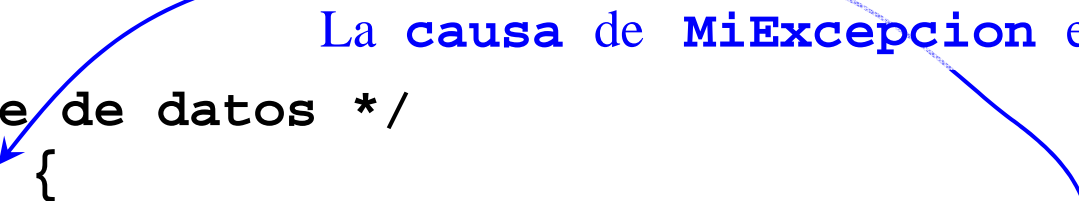
Ejemplo de implementacion de pop()
con lanzamiento de excepcion

Excepciones en cadena

- Ocurre cuando una aplicación responde a una excepción con otra
- `Throwable.getCause()` devuelve la excepción causante
- `Throwable.initCause(Throwable)` fija la causa de la excepción (solo se puede fijar una vez)
- Normalmente se usa el constructor de la excepción
`Throwable(String mensaje, Throwable causa)`

```
try {  
    /* carga de la base de datos */  
} catch (IOException e) {  
    throw new MiExcepcion("Error abriendo base de datos", e);  
}
```

La causa de `MiExcepcion` es ...





Tratamiento de excepciones

- Imprimiendo la pila de llamadas

```
catch(Exception e){  
    e.printStackTrace();  
}
```

- Obteniendo la pila de llamadas

```
StackTraceElement[] stackElemets = e.getStackTrace();  
for (StackTraceElement se : stackElemets ) {  
    System.err.println(se.getFileName()+": "  
                        +se.getLineNumber()+"\t"  
                        +se.getMethodName+"( )")  
}
```



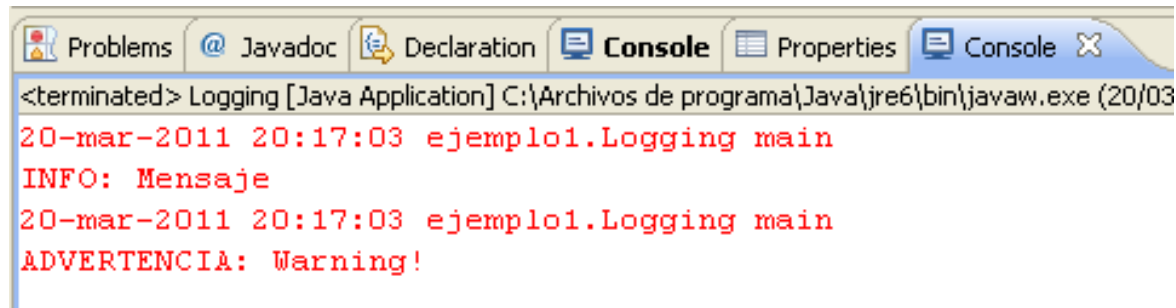
Servicio de Logging

- Se usa para guardar los errores o mensajes para depuración en archivo, usando la configuración de la aplicación
- Puede haber un logger global del sistema, por aplicación o un paquete específico
- Sistema jerárquico, nombres separados por puntos
 - Ej: `Logger logger= Logger.getLogger("java.net");`
- `logger.log(Level, mensaje [, objeto o excepción asociada])`
 - Level (de más a menos grave): SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST
 - Ej: `logger.log(Level.WARNING, "Tiempo de espera agotado, reintentando...", excepcion);`

Servicio de Logging

```
import java.io.IOException;
import java.util.logging.FileHandler;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Logging {
    public static void main(String[] arg) throws IOException {
        Logger log = Logger.getLogger(CuentaBancaria.class.getName());
        log.addHandler(new FileHandler("out.xml"));
        log.setLevel(Level.INFO);
        log.info("Mensaje");
        log.warning("Warning!");
    }
}
```





```
<?xml version="1.0" encoding="windows-1252"
standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2011-03-20T20:17:03</date>
  <millis>1300648623093</millis>
  <sequence>0</sequence>
  <logger>ejemplo1.CuentaBancaria</logger>
  <level>INFO</level>
  <class>ejemplo1.Logging</class>
  <method>main</method>
  <thread>10</thread>
  <message>Mensaje</message>
</record>
<record>
  <date>2011-03-20T20:17:03</date>
  <millis>1300648623140</millis>
  <sequence>1</sequence>
  <logger>ejemplo1.CuentaBancaria</logger>
  <level>WARNING</level>
  <class>ejemplo1.Logging</class>
  <method>main</method>
  <thread>10</thread>
  <message>Warning!</message>
</record>
</log>
```

Servicio de Logging

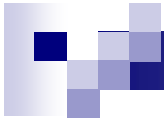
Fichero out.xml

¿Cuándo no?

- No es un reemplazo a las condiciones *normales*
- Ejemplo: Inicialización *perezosa* de listas

```
List<Elemento> lista; // sin inicializar, por ahora
Elemento x;
try {
    lista.add(x);
}
catch (Exception e){
    lista= new ArrayList<Elemento>();
    lista.add(x);
}
```

```
//Mejor con if
if (lista == null) { lista = new ArrayList<Element>(); }
lista.add(x);
```

Ventajas de las excepciones

- Separación entre código normal y tratamiento de errores
- Propagación sencilla en la pila de llamadas
 - Se tratan en el nivel que pueda controlarlas mejor
- Agrupación de errores por tipo y categorización
- Cada método indica que excepciones puede lanzar (Controladas/Checked exceptions)