5 – Mecanismo de tratamiento de excepciones y errores

5.1 – INTRODUCCIÓN. PATRÓN DE SOPORTE DE ERRORES Y EXCEPCIONES

En la ejecución de sentencias de un programa, pueden producirse situaciones diferentes de las deseadas:

- •la imposibilidad de abrir un fichero por diversos motivos,
- •la imposibilidad de efectuar una división porque la expresión denominador da cero como resultado de su evaluación,
- •la imposibilidad de acceder a un determinado objeto por utilizar erróneamente una referencia nula,
- •etc.

Es necesario un mecanismo de detección y control de modo que para cada situación no deseada se actúe en consecuencia.

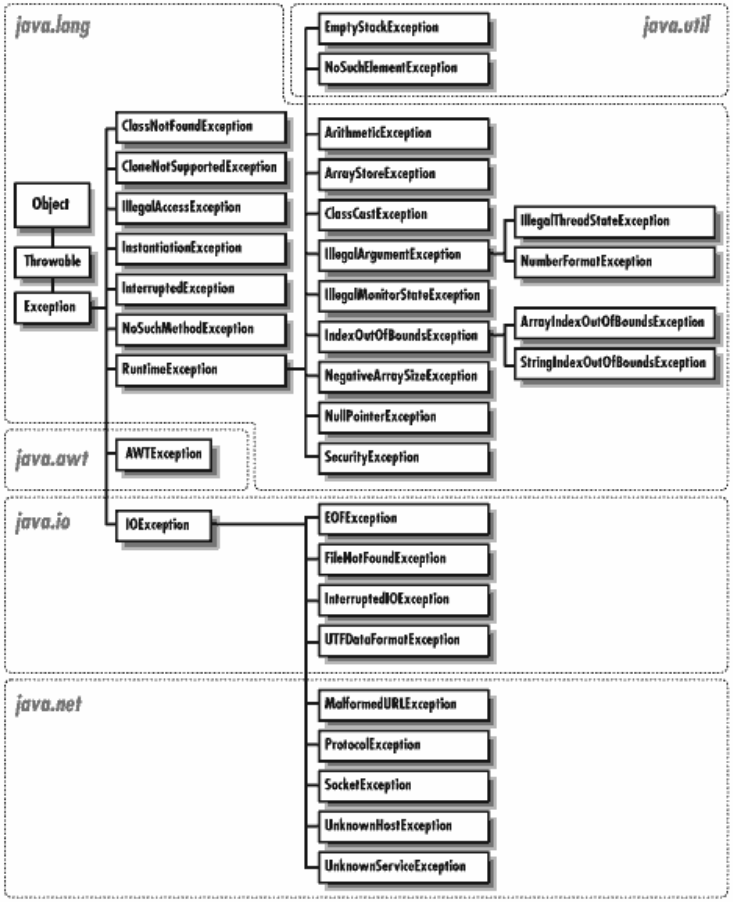
En los lenguajes de programación "clásicos" no se suele disponer de otro mecanismo que la sentencia condicional, que permite establecer dos caminos de ejecución, uno para la situación "deseada", y otro para la "problemática". De este modo ambos caminos forman parte del algoritmo, que ya no es sólo el algoritmo de ejecución de una determinada tarea, sino que es esto más el control de toda la combinatoria de posibles situaciones "problemáticas".

Java permite separar el algoritmo de ejecución de una tarea de los algoritmos de solución de las diversas situaciones "problemáticas":

Cuando se da una de dichas situaciones, la ejecución del algoritmo "principal" (camino de éxito) se aborta y se trasfiere a otro punto donde se encuentra el algoritmo de gestión del problema concreto producido, pasándole información de lo sucedido encapsulada en un objeto.

Terminología: los objetos con información del problema son "arrojados" en el momento de la excepción y "capturados" por el bloque de código que se hace cargo de la situación.

Todas las clases correspondientes se encuentran en un subárbol de la jerarquía que comienza con "`Throwable`" y que tiene a su vez dos subárboles cuyas raíces son "`Error`" y "`Exception`".



Errores

(Normalmente) no es posible recuperarse de un error.

El mecanismo try-catch/arrojar puede usarse para llevar a un cierre controlado del programa.

Los errores son causados principalmente por el entorno en el que se ejecuta el programa.

Todos los errores son de tipo “no chequeado” (unchecked).

Ejemplos:
StackOverflowError,
OutOfMemoryError

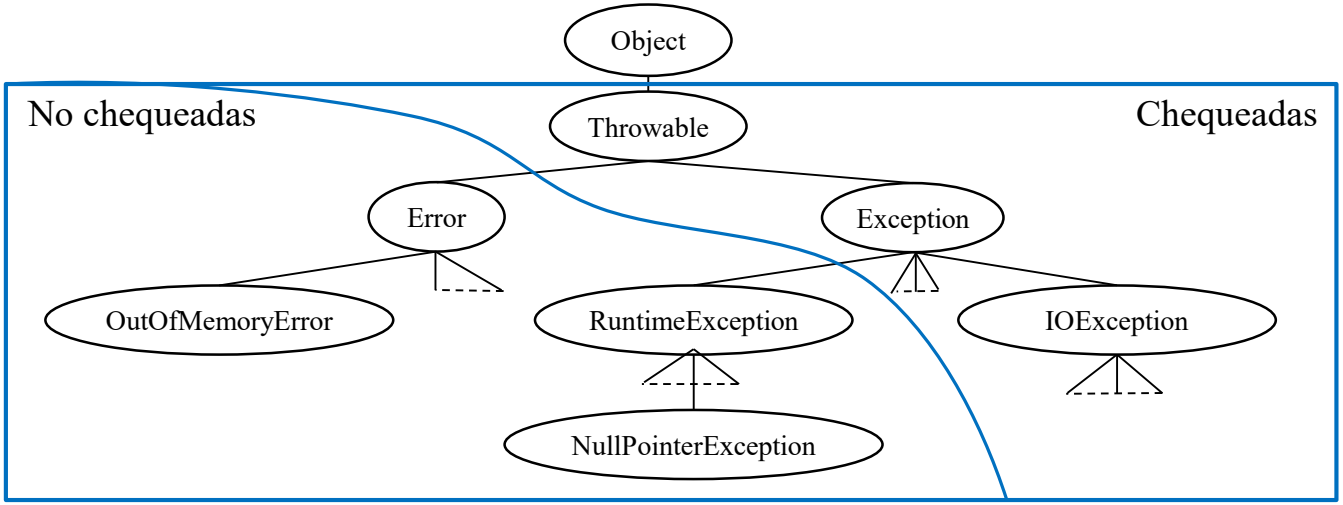
Excepciones

(En general) podemos recuperarnos de las excepciones utilizando el mecanismo try-catch/arrojar.

El programa en sí es responsable de provocar excepciones.

Hay excepciones “chequeadas” y otras que no lo son.

Ejemplos de excepciones chequeadas:
SQLException, IOException
Ejemplos de excepciones no chequeadas:
ArrayIndexOutOfBoundException,
NullPointerException, ArithmeticException.



5.2 – CONTROL DE EXCEPCIONES

Sentencia "try – catch"

La separación de los bloques de código para una ejecución correcta y sus posibles situaciones excepcionales se lleva a cabo mediante una sentencia que se dejo sin ver en su momento para hacerlo ahora:

```
try bloque_sentencia [catch (clase_arrojable id) bloque_sentencia]+ finally bloque_sentencia
```

Palabras reservadas en Java				
abstract	assert(1.4)	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum(5.0)	extends	final	finally	float
for	goto*	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp(1.2)	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

Analizaremos el siguiente ejemplo:

```
 1- FileInputStream f=null;
 2- int d;
 3- try {
 4-    f=new FileInputStream("pepe");
 5-    d=f.read();
 6-    //aquí el restro de acciones que se quieran realizar con f
 7- } catch (FileNotFoundException ex) {
 8-    //aquí el código a ejecutar en caso de que no existiera el fichero "pepe"
 9- } catch (IOException ex) {
10-    //aquí el código a ejecutar en caso de haya cualquier otro problema con el fichero
11- } finally {
12-    if (f!=null) f.close();
13-    //aquí otro código que deba ejecutarse en todo caso
14- }
```

`FileInputStream` es capaz de abrir un fichero para realizar lecturas de datos. "f" es una referencia que utilizaremos para un objeto de este tipo.

1. En las lineas 3 a 7 se encuentra un bloque de código tras el término **`try`** que recoge la ejecución que queremos realizar con nuestro fichero (abrir uno de nombre "pepe", leer un entero sobre la variable d, etc.)

2.a. En el momento de instanciar el objeto "f" (línea 3) puede producirse un error de clase `FileNotFoundException`, de modo que el bloque de las líneas 7 y 8, recibe un objeto de esta clase en caso de producirse el problema y actua en consecuencia (nótese que el código de este **`catch`** puede hacer uso del objeto recibido o no).

2.b Igualmente, en el método `read()` pueden producirse indeterminados errores de entrada/salida identificados con un objeto de clase `IOException` (p.ej. faltas de permisos de acceso, ausencia de datos sin fin de fichero, etc). Las acciones a ejecutar en tal caso se llevan a cabo en el bloque 9-10.

3. Por último, el fragmento **`finally`** de las líneas 11-14 se ocupará de terminar con todo aquello que pueda haber quedado en situación inconclusa, como por ejemplo de cerrar el fichero en caso de que se hubiese llegado a abrir.

Detalles:

•El bloque finally se ejecuta SIEMPRE, incluso si nos encontramos dentro de un método y hay un `return` en los bloques `try/catch` (la ÚNICA posibilidad de que no sea así es que la aplicación termine en uno de los bloques mediante `System.exit(.)`)

•Si tenemos varios `catch` asociados a clases de una misma línea de jerarquía, es preciso situarlos ordenadamente del más específico al más general.

Sentencia “try – catch con recursos”

Esta variante del “try” tiene la siguiente sintaxis:

```
try (recursos) bloque_sentencia [catch (clase_arrojable id) bloque_sentencia]+ finally bloque_sentencia
```

El ejemplo anterior:

```
FileInputStream f=null;
int d;
try {
        f=new FileInputStream(”pepe”);
        d=f.read();
        //aquí el restro de acciones que se quieran realizar con f
} catch (FileNotFoundException ex) {
        //aquí el código a ejecutar en caso de que no existiera el fichero “pepe”
} catch (IOException ex) {
        //aquí el código a ejecutar en caso de haya cualquier otro problema con el fichero
} finally {
        if (f!=null) f.close();
        //aquí otro código que deba ejecutarse en todo caso
}
```

Puede escribirse con esta versión del “try” como

```
int d;
try (FileInputStream f=new FileInputStream(”pepe”);) {
        d=f.read();
        //aquí el restro de acciones que se quieran realizar con f
} catch (FileNotFoundException ex) {
        //aquí el código a ejecutar en caso de que no existiera el fichero “pepe”
} catch (IOException ex) {
        //aquí el código a ejecutar en caso de haya cualquier otro problema con el fichero
}
```

Ahora el recurso (el fichero) está declarado para el bloque “try-catch” y queda “bajo su control”. No es necesario “cerrarlo” ya que de ello se encarga la sentencia (esto es así para toda clase que implemente AutoCloseable)

5.2 – CONTROL DE EXCEPCIONES (cont.)

Una vez que se produce la excepción, se aborta la ejecución del código, de modo que si se encontraba dentro de un bloque **`try`** pasará a ejecutarse el correspondiente **`catch`**. De no ser así, terminará el método y este "arrojará" el objeto "hacia" el método le que llamó.

Cuando un método deja "escapar" objetos arrojables es preciso que se indique en su declaración, para lo que basta con añadir a su prototipo, después de los parámetros la palabra reservada **`throws`** y la lista, separada con comas, de todas aquellas clases de las que puede lanzar objetos. P.ej:

Palabras reservadas en Java				
abstract	assert(1.4)	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum(5.0)	extends	final	finally	float
for	goto*	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp(1.2)	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

```
1- public void desahogada() throws IOException, NumberFormatException, ArithmeticException {
2-   //cuerpo del método, donde se producen y no se atienden las excepciones indicadas.
3- }
```

Atención!!: Ante las situaciones en que puedan producirse excepciones (o errores) tendremos que plantearnos cuidadosamente qué será lo adecuado, si atenderlas en un bloque **`catch`** o dejarlas salir del método declarándolo explícitamente.

Sobreescritura de métodos que arrojan excepciones o errores

Puede sobreescribirse un método que en la clase padre arrojaba excepciones variando el comportamiento al respecto, pero sólo en el sentido de hacerla más restrictiva, es decir tratando internamente ciertas excepciones que el método de la clase padre no trataba.

```
1- import java.io.*;
2- import java.text.*;
3-
4- public class A {
5-
6-     public void metodo() throws IOException, ParseException {
7-         // ...
8-         int i=f.read(); //IOException no atendida
9-         //...
10-        Double d=df.parse(s); //ParseException no atendida
11-        //...
12-    }
13-
14-    //....
15- }
```

```
1- import java.io.*;
2-
3- public class B extends A{
4-
5-     public void metodo() throws FileNotFoundException {
6-         // ...
7-         FileInputStream f=new FileInputStream();
8-         //...
9-     }
10-
11-    //....
12- }
```

Nota: El compilador se encarga bastante de que hagamos bien la gestión de errores ya que avisa posibles excepciones no consideradas, de ordenaciones incorrectas de los catch, nos obliga a tomar una determinación de atender o declarar que una excepción sale del método… No obstante permite que algunas excepciones no sean tratadas ya que entiende que obedecen a situaciones que el programador ha podido evitar (por ejemplo `ArithmeticException` puede darse en una división de enteros cuando el denominador es cero, pero puede que el algoritmo no permita esta situación, por lo que no es razonable obligar a "controlarla").

5.3 – Generación de excepciones

Nuestros programas pueden generar excepciones:

Detectada la situación excepcional puede activarse el mecanismo instanciando un objeto de la subclase "arrojable" que se considere adecuada, y utilizando "`throw`":

```
throw objeto_arrojable
```

Palabras reservadas en Java				
abstract	assert(1.4)	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum(5.0)	extends	final	finally	float
for	goto*	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp(1.2)	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

Es habitual no añadir información alguna ya que la identidad de la clase en sí misma puede ser suficiente información.

```
1- if (condición_de_error) throw new IOException();
```

En ocasiones es interesante incluir un texto informativo:

```
1- if (condición_de_error) throw new IOException("Rebasado límite autorizado en salida");
```

Pero en caso de querer introducir información específica deberemos generar nuestras propias instrucciones

5.3 – DEFINICION DE NUEVAS EXCEPCIONES

Definir una nueva excepción o error no es otra cosa que definir una clase que herede de `Exception` o `Error` respectivamente o de alguna de sus clases descendientes si se considera preciso.

Típicamente esta definición suele ser meramente "de trámite" ya que su comportamiento puede ser suficiente con el heredado, de modo que solo se reescriben los constructores:

```
public class MyException extends Exception {
public MyException(){}
public MyException(String s){super(s);}
public MyException(Throwable t){super(t);}
public MyException(String s, Throwable t){super(s,t);}
}
```

Naturalmente nada se opone a que hagamos una definición de la excepción "a nuestra medida" añadiéndole las características que deseemos.

Información de excepciones

Si se produce una excepción que aborta la ejecución de un programa (el compilador no nos ha obligado a considerarla y no lo hemos hecho), se nos muestra la "traza de ejecución"

```
java.lang.NullPointerException
        org.apache.jsp.MiCursoJava.tema5.error_jsp._jspService(error_jsp.java:56)
        org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:97)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
        org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:332)
        org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:314)
        org.apache.jasper.servlet.JspServlet.service(JspServlet.java:264)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
```

Hay excepciones que contienen excepciones:

excepción

```
org.apache.jasper.JasperException
        org.apache.jasper.servlet.JspServletWrapper.handleJspException(JspServletWrapper.java:510)
        org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:393)
        org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:314)
        org.apache.jasper.servlet.JspServlet.service(JspServlet.java:264)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
```

causa raíz

```
java.lang.NullPointerException
        org.apache.jsp.MiCursoJava.tema5.error_jsp._jspService(error_jsp.java:56)
        org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:97)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
        org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:332)
        org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:314)
        org.apache.jasper.servlet.JspServlet.service(JspServlet.java:264)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
```

Si atendemos a una excepción y evitamos que la aplicación se aborte, podemos mostrar esta misma información valiéndonos del método "printStackTrace()" sobre el objeto arrojado.
(un programa "en producción" no debe contener llamadas a este método)

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

En ocasiones sólo queremos evitar que la aplicación se aborte. En estos casos conviene dejar claro de algún modo que es intencionado.

```
catch (IOException ignore) {}
```

Aserciones

Las “aserciones” permiten chequear una condición de modo que si no se cumple se arroja un error (AssertionError).

```
assert expresión1;
assert expresión1: expresión2;
```

La “expresión1” es la condición booleana, y “expresión2” puede ser añadida para que sea proporcionada por la excepción.

Son de utilidad en algunas situaciones en que se quieren asegurar condiciones que se presuponen ciertas (en este sentido pueden “sustituir” a comentarios y ser utilizadas en combinación con la posibilidad de deshabilitarlas en tiempo de ejecución).

Palabras reservadas en Java				
abstract	assert(1.4)	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum(5.0)	extends	final	finally	float
for	goto*	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp(1.2)	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

```
private void setRefreshInterval(int interval) {
assert interval > 0 && interval <= 1000/MAX_REFRESH_RATE : interval;
... // Set the refresh interval
}
```

“funcionalmente” equivale a…

```
if (! (assert interval > 0 && interval <= 1000/MAX_REFRESH_RATE)) throw new AssertionError(interval);
```

En tiempo de ejecución pueden activarse/desactivarse determinas aserciones con“-ea” (enable assertions) y “-da” (disable assertions), de modo que puede eliminarse su coste de ejecución y ser reactivadas si surgen problemas. (téngase en cuenta que el test puede ser costoso)

```
java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat BatTutor
```

- El manejo de aserciones puede complicarse considerablemente, y está ligado a técnicas de informática teórica como el chequeo de pre/post-condiciones, de invariantes, etc.
- No nos extendemos más en este punto. Puede estudiarse en http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html