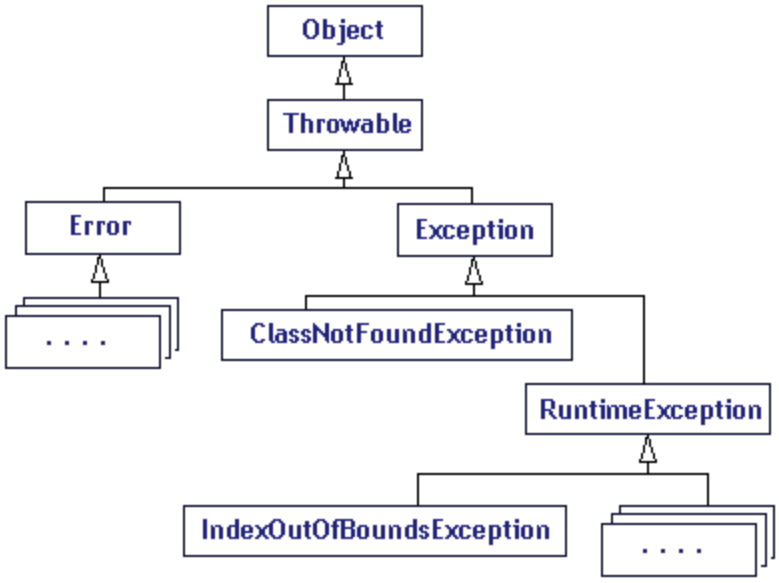
UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS



Excepciones en Programación Orientada a Objetos

2013
Transversal Programación Orientada a Objetos
Proyecto Curricular de Ingeniería de Sistemas

Ítems

1. **Manejo de excepciones**
2. **Capturar excepciones**
3. **Generar excepciones en Java**
4. **Crear excepciones propias**

Introducción

Cuando se produce un error en tiempo de ejecución se debe tener un control sobre el error y poder manejarlo sin que se "rompa" nuestra aplicación.

Las excepciones en Java están destinadas, al igual que en el resto de los lenguajes que las soportan, para la detección y corrección de errores. Si hay un error, la aplicación no debería "romperse". Se debería lanzar una excepción que se pueda capturar y resolver la situación de error. Java permite este mecanismo de control de errores, permitiendo aumentar en gran medida la robustez de nuestros desarrollos.

1. Manejo de excepciones.

Vamos a producir un error en tiempo de ejecución y luego vamos a ver cómo podemos manejarlo, en el siguiente código trataremos de mostrar una posición no existente de un arreglo:

```
public class ArregloSaludo {
	private int i=0;
	private String[] saludo = {"Hola mundo",
				"Hola a todos",
				"Hola java"};
	public void escribirSaludo(){
		System.out.println(saludo[i++]);
		this.escribirSaludo();
	}
	public static void main(String[] args){
		ArregloSaludo a = new ArregloSaludo();
		a.escribirSaludo();
	}
}
```

En el código anterior el método recursivo escribirSaludo() va incrementando el índice del arreglo el cual muestra en pantalla y se llama a sí mismo, generando un error cuando el índice que desea mostrar no existen en el arreglo, en este caso el índice 3, generando la siguiente salida en pantalla:

```
Hola mundo
Hola a todos
Hola java

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
```

Ahora manejemos el error para que no se "rompa" nuestra aplicación, capturando la excepción que se produce al tratar de acceder a un índice del arreglo que no existe, nuestro código queda de la siguiente manera:

```
public class ArregloSaludoAlte {
	private int i=0;
	private String[] saludo = {"Hola mundo",
				   "Hola a todos",
				   "Hola java"};
	public void escribirSaludo(){
		try{
			System.out.println(saludo[i++]);
			this.escribirSaludo();
		}catch(ArrayIndexOutOfBoundsException e){
			System.out.println("Saludo desbordado");
		}catch(Exception e){
			System.out.println(e.toString());
		}finally{
			System.out.println("siempre se escribe");
		}
	}
	public static void main(String[] args){
		ArregloSaludoAlte a = new ArregloSaludoAlte();
		a.escribirSaludo();
	}
}
```

2. Capturando excepciones.

La herramienta que nos permite capturar una excepción es el bloque *try – catch – finally*, de esta manera el código que tratamos de ejecutar dentro del bloque *try* en los bloques *catch* podremos capturar y manejar las excepciones que se produzcan, en este caso *ArrayIndexOutOfBoundsException* que se produce por tratar de usar una posición de arreglo que no existe. Vale la pena apuntar que se pueden capturar tantas excepciones como sea necesario y

que adicionalmente el bloque *finally* siempre será ejecutado. La salida en pantalla del bloque de código anterior será:

```
Hola mundo
Hola a todos
Hola java
Saludo desbordado
siempre se escribe
siempre se escribe
siempre se escribe
siempre se escribe
```

Veamos una explicación de estos bloques

try

Es el bloque de código donde se prevé que se genere una excepción. Es como si le indicáramos a java "intenta este código y mira a ver si se produce una excepción". El bloque try tiene que ir seguido, al menos, por una cláusula catch o una cláusula finally

catch

Es el código que se ejecuta cuando se produce la excepción. En este bloque tendremos que asegurarnos de colocar código que no genere excepciones. Se pueden colocar sentencias catch sucesivas, cada una controlando una excepción diferente. No debería intentarse capturar todas las excepciones con una sola cláusula, como esta:

```
    catch( Excepcion e ) { ...
```

Esto representaría un uso demasiado general, podrían llegar muchas más excepciones de las esperadas. En este caso es mejor dejar que la excepción se propague hacia arriba y dar un mensaje de error al usuario.

finally

Es el bloque de código que se ejecuta siempre, haya o no excepción, por ejemplo, podría servir para hacer un log o un seguimiento de lo que está pasando, porque como se ejecuta siempre puede dejarnos grabado si se producen excepciones y nos hemos recuperado de ellas o no.

Excepciones predefinidas

Los nombres de las excepciones indican la condición de error que representan. Las siguientes son las excepciones predefinidas más frecuentes que se pueden encontrar:

ArithmeticException

Las excepciones aritméticas son típicamente el resultado de una división por 0:

```
        int i = 12 / 0;
```

NullPointerException

Se produce cuando se intenta acceder a una variable o método antes de ser definido:

```
class PruebaBinario{
		public static void main(String[] args){
			Binario bin;
			int x = 15;
			System.out.println(bin.convertir(x,10));
		}
	}
```

IncompatibleClassChangeException
El intento de cambiar una clase afectada por referencias en otros objetos, específicamente cuando esos objetos todavía no han sido recompilados.

ClassCastException
El intento de convertir un objeto a otra clase que no es válida.

```
y = (Prueba)x;		// donde x no es de tipo Prueba
```

NegativeArraySizeException
Puede ocurrir si hay un error aritmético al intentar cambiar el tamaño de un array.

OutOfMemoryException
¡No debería producirse nunca! El intento de crear un objeto con el operador *new* ha fallado por falta de memoria. Y siempre tendría que haber memoria suficiente porque el *garbage collector* se encarga de proporcionarla al ir liberando objetos que no se usan y devolviendo memoria al sistema.

NoClassDefFoundException
Se referenció una clase que el sistema es incapaz de encontrar.

ArrayIndexOutOfBoundsException
Es la excepción que más frecuentemente se produce. Se genera al intentar acceder a un elemento de un array más allá de los límites definidos inicialmente para ese array.

UnsatisfiedLinkException
Se hizo el intento de acceder a un método nativo que no existe. Por ejemplo:

```
public class Prueba {

    native void miMetodoNativo();

    public static void main(String[] args){
        Prueba p = new Prueba();

        p.miMetodoNativo ();
    }

}
```

Se trató de llamar a *p.miMetodoNativo ()*, cuando debería llamar a *Prueba.miMetodoNativo ()*.

> **NOTA:** *Un método nativo no puede tener cuerpo ya que este código debe estar definido en una librería dinámica (dll), estas librerías son archivos de funciones previamente compiladas y generalmente en lenguajes distintos de java (C++, Fortran, etc).*

InternalException

Este error se reserva para eventos que no deberían ocurrir. Por definición, el usuario nunca debería ver este error y esta excepción no debería lanzarse.

3. Generar excepciones en Java.

Cuando se produce un error se debería generar, o lanzar, una excepción. Para que un método en Java, pueda lanzar excepciones, hay que indicarlo expresamente.

```
void MetodoAsesino() throws NullPointerException,CaidaException
```

Se pueden definir excepciones propias, no hay por qué limitarse a las predefinidas; bastará con extender la clase **Exception** y proporcionar la funcionalidad extra que requiera el tratamiento de esa excepción.

También pueden producirse excepciones no de forma explícita como en el caso anterior, sino de forma implícita cuando se realiza alguna acción ilegal o no válida.

Las excepciones, pues, pueden originarse de dos modos: el programa hace algo ilegal (caso normal), o el programa explícitamente genera una excepción ejecutando la sentencia *throw* (caso menos normal). La sentencia throw tiene la siguiente forma:

```
    throw ObtejoExcepction;
```

El objeto ObjetoException es un objeto de una clase que extiende la clase **Exception**. El siguiente código de ejemplo origina una excepción de división por cero:

```
class melon {
    public static void main( String[] args ) {
        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 melon.java
> java melon
     java.lang.ArithmeticException: / by zero
           at melon.main(melon.java:5)
```

Las excepciones predefinidas, como *ArithmeticException*, se conocen como excepciones runtime. 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 sobrescribir algún elemento que ya no se necesite.

4. Crear excepciones propias.

También podemos lanzar nuestras propias excepciones, extendiendo la clase **System.exception**. Por ejemplo, consideremos un programa cliente/servidor. El código cliente se intenta conectar al servidor, y durante 5 segundos se espera a que conteste el servidor. Si el servidor no responde, el servidor lanzaría la excepción de time-out:

```
class ServerTimeOutException extends Exception {}

public void conectame( String nombreServidor ) throws Exception {
    int exito;
    int puerto = 80;

    exito = open( nombreServidor,puerto );
    if( exito == -1 )
        throw ServerTimeOutException;
  }
```

Si se quieren capturar las propias excepciones, se deberá utilizar la sentencia *try*:

```
public void encuentraServidor() {
   ...
   try {
        conectame( servidorDefecto );
        catch( ServerTimeOutException e ) {
            g.drawString(
                "Time-out del Servidor, intentando alternativa",
                5,5 );
            conectame( servidorAlterno );
            }
    ...
    }
```

Cualquier método que lance una excepción también debe capturarla, o declararla como parte de la interface del método. Cabe preguntarse entonces, el porqué de lanzar una excepción si hay que capturarla en el mismo método. La respuesta es que las excepciones no simplifican el trabajo del control de errores. Tienen la ventaja de que se puede tener muy localizado el control de errores y no tenemos que controlar millones de valores de retorno, pero no van más allá.

Bibliografía.

- Bruce Eckel. Piensa en Java. Pretince Hall. 2002
- Agustín Froufe Quintas. Java 2 Manual de usuario y tutorial. Alfaomega.