Captura de excepciones

Definición de excepción

Una excepción es una situación anómala que puede producirse durante la ejecución de un programa, lo que comúnmente se llama error de ejecución.

Cuando se produce una excepción en un punto del programa, se crea un objeto de este tipo de excepción y se dice que se "lanza" al programa. Si este no la trata y la excepción termina llegando a la máquina virtual Java, se interrumpe la ejecución y se muestra en la consola la traza de error:

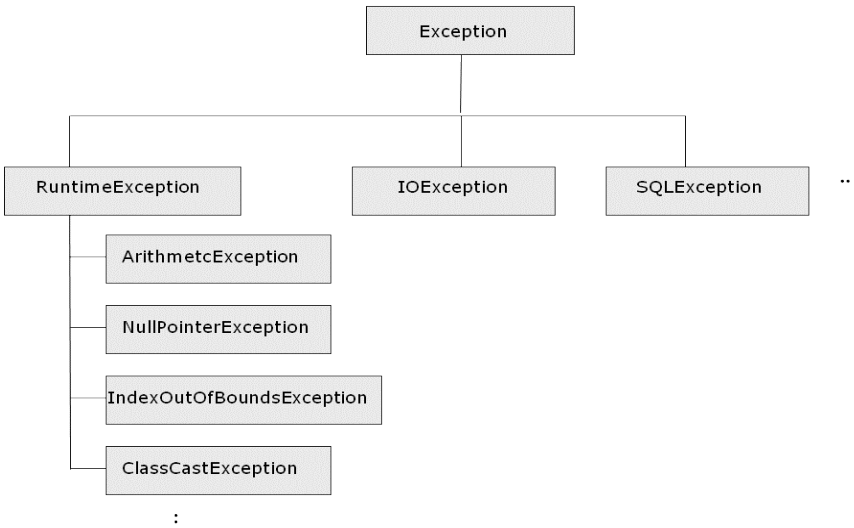
```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 6
        at principal.Primitiva.ordenar(Primitiva.java:35)
        at principal.Primitiva.main(Primitiva.java:18)
```

Las excepciones se producen muchas veces por errores de programación, en esos casos lo que hay que hacer es corregir los errores durante la fase de pruebas de la aplicación. Pero otras veces las excepciones se producen por circunstancias anómalas, que no son fallos de programación y que por tanto no se pueden corregir. En estos casos, el programa deberá proporcionar un control de excepciones de modo que cuando se produzcan, no se interrumpa la ejecución del mismo, sino que se lleven a cabo acciones determinadas dentro del propio programa.

Clases de excepción

Cuando se produce una excepción se crea un objeto del tipo de excepción producida, todos estos tipos de excepción corresponden con clases que heredan **Exception**, la clase de excepción principal de Java.

En la siguiente imagen mostramos un extracto de la jerarquía de clases de excepciones de Java



Tipos de excepción

Desde el punto de vista de la naturaleza de las excepciones, podemos dividir estas en dos grandes grupos:

- **Excepciones verificadas (checked)**. Son excepciones específicas de determinados grupos de clases de Java y que pueden producirse durante el uso de algún método de alguna de estas clases. Como ejemplo tenemos la excepción IOException que puede producirse por el uso de ciertos métodos de las clases de entrada y salida Java, o la excepción SQLException que puede producirse durante el uso de las clases del java.sql para acceso a datos. Estas excepciones deben ser **obligatoriamente capturadas** en un programa, el compilador nos obliga a definir un control de excepciones a la hora de usar los métodos que pueden provocar estas excepciones
- **Excepciones no verificadas (unchecked)**. Son excepciones de uso general, pueden producirse en cualquier tipo de aplicación y son todas subclases de RuntimeException. No hay obligación de capturar estas excepciones, pues la mayoría se producen por fallos de programación que hay que corregir, aunque en algunos casos, como en la NumberFormatException, si suele establecerse un sistema de captura de excepciones.

Captura de excepciones

Capturar una excepción es indicarle al programa que si se produce una excepción, se dirija a un determinado bloque de sentencias donde el error será tratado.

Bloques try catch

La captura de excepciones en Java se realiza a través de los bloques try catch, cuya estructura se indica a continuación:

```
try{

 //instrucciones

}

catch(TipoExcepcion1 ex){

//tratamiento excepción

}

catch(TipoExcepcion2 ex){

//tratamiento excepcion

}
```

Como vemos, el bloque de instrucciones del programa se engloba dentro del bloque try. Si durante la ejecución de este bloque se produce alguna excepción en alguna instrucción, se comprobará si la excepción coincide con alguno de los tipos definidos en los bloques catch y, si es así, el programa continuará por dicho bloque catch.

Tras la ejecución de un catch el control del programa **no vuelve al punto donde se produjo la excepción**, sino que continúa después del último bloque catch.

Indicar también que no se puede escribir ninguna instrucción entre los bloques try y catch:

try{

:

}

int k=10; //error de compilación

catch(IOException ex){

:

}

El programa de ejemplo que indicamos a continuación, realiza la lectura por teclado de dos números y calcula la división entre el mayor y el menor. Para evitar los problemas derivados de la introducción incorrecta de los números o una posible división entre 0, hemos incluido un control de las excepciones InputMismatchException y ArithmeticException, ambas de tipo RuntimeException:

```
public class Test {
	public static void main(String[] args) {
		Scanner sc=new Scanner(System.in);
		int n1,n2,r;
		try{
			System.out.println("Introduce númerador ");
			n1=sc.nextInt();
			System.out.println("Introduce denominador ");
			n2=sc.nextInt();
			r=n1/n2;
			System.out.println("La división es "+r);
		}
		catch(InputMismatchException ex){
			System.out.println("Número no válido");
		}
		catch(ArithmeticException ex){
			System.out.println("División no válida");
		}
	}
}
```

Todo el bloque de instrucciones se ha incluido dentro del try, de modo que la última instrucción solo se ejecutará si no ha habido ningún problema durante la introducción de los números y el cálculo.

Si al hacer una llamada a *nextInt()* se introduce un valor no entero, se produce una InputMismatchException y el programa salta al catch correspondiente donde se muestra el

mensaje de error. Si n2 es 0, entonces la excepción se produce al hacer la división, y será del tipo ArtihmeticException.

Si se produjera una excepción diferente a las indicadas en los catch, al no ser capturada se propagaría a la JVM y esta interrumpiría la ejecución del programa y mostraría en la consola la traza de error.

Agrupación de catch

Si dos o varios catch van a realizar el mismo tratamiento de excepción, se pueden agrupar en uno solo de la forma:

```
catch(InputMismatchException|ArithmeticException ex){
      System.out.println("Error en los cálculos");
}
```

Relación de herencia entre excepciones

Si queremos capturar dos excepciones y una de ellas hereda de otra, el catch de la hija deberá ir antes que el de la padre:

```
catch(InputMismatchException ex){
      //
}

catch(RuntimeException ex){
      //
}
```

Si invertimos el orden de los catch se producirá un error de compilación, ya que en el catch de la hija nunca entraría.

Métodos de Exception

Todos los objetos de error que se envían a los catch cuando se produce una excepción pertenecen a clases que heredan Exception, por tanto, tendrán los mismos métodos que ésta. Entre ellos, destacamos:

String getMessage(). Devuelve el mensaje de error asociado a la excepción. Por ejemplo, si en el código de ejemplo anterior hubiéramos definido el catch de ArithmeticException de la siguiente manera:

```
catch(ArithmeticException ex){
      System.out.println(ex.getMessage());
}
```

Al haberse realizado una división entre 0 se habría mostrado el siguiente mensaje por pantalla:

/ by zero

void printStackTrace(). En algunas ocasiones, sobretodo cuando estamos ante excepciones que estamos obligados a capturar, puede ser interesante mostrar la traza de error en la consola cuando se produce la excepción, a fin de poder analizarla y ver si el problema es por algún error de programación. Esto es precisamente lo que hace el método *printStackTrace()*.

El bloque finally

Además de los bloques catch para la captura de los diferentes tipos de excepciones que se puedan producir en un programa, podemos definir un bloque finally en el que incluir las instrucciones que queramos ejecutar de forma obligada, tanto si se produce una excepción como sino.

La estructura general será:

try{

//instrucciones

}

catch(TipoExcepcion1 ex){

//tratamiento excepción

}

catch(TipoExcepcion2 ex){

//tratamiento excepcion

}

finally{

//instrucciones finales

}

Según lo que acabamos de indicar, si se ejecuta el bloque try sin que se produzca ninguna excepción, el programa continuará después por el bloque finally. Si se produce una excepción dentro del try, se ejecutará el bloque catch correspondiente y después el finally. Si se produce la excepción y no hay ningún bloque catch para su captura, antes de propagarla a la JVM también se ejecutará el finally. Es decir, **el bloque finally siempre se ejecuta**, pase lo que pase.

Por ejemplo, dada la siguiente versión del programa que realiza la división entre dos números:

```
public class Test {
	public static void main(String[] args) {
		Scanner sc=new Scanner(System.in);
		int n1,n2,r;
		try{
			System.out.println("Introduce númerador ");
			n1=sc.nextInt();
			System.out.println("Introduce denominador ");
			n2=sc.nextInt();
			r=n1/n2;
			System.out.println("La división es "+r);
		}
		catch(InputMismatchException ex){
```

```
                System.out.println(ex.getMessage());
                return;
            }
            catch(ArithmeticException ex){
                System.out.println(ex.getMessage());
                return;
            }
            finally{
                System.out.println("final de programa");
            }
        }
}
```

Ocurra lo que ocurra, siempre se mostrará el mensaje "final de programa", incluso a pesar de las instrucciones return de los catch, antes de abandonar el método se produciría la ejecución del bloque finally.

Los bloque finally se utilizan especialmente en aquellos casos en los que trabajamos con recursos que deben ser cerrados una vez que dejan de ser utilizados. Al incluir el cierre de los recursos (ficheros, conexiones con bases de datos, etc.) dentro de los bloques try, nos aseguramos que dicho cierre siempre se producirá.

Excepciones personalizadas

Podemos crear también nuestras propias clases de excepción como una forma de notificar desde un método a otra parte del programa que se ha producido una situación anómala.

Si queremos que sea una excepción verificada, es decir, que sea obligatoria su captura, crearemos esta clase como una subclase de Exception:

```
public class OperacionNoValidaException extends Exception{



}
```

No es necesario definir ningún método adicional, ya que por el hecho de heredar Exception ya será tratada como una excepción. Si queremos asignar a este tipo de excepción un mensaje personalizado, que luego se obtendría mediante getMessage(), sería:

```
public class OperacionNoValidaException extends Exception{

        public OperacionNoValidaException(){

                super("Operación no permitida");

        }

}
```

Creación y lanzamiento de la excepción

Las excepciones son provocadas desde el interior de un método de alguna clase. Para ello, se creará un objeto del tipo de excepción que se va a provocar y se lanzará mediante la instrucción **throw**, cuya sintaxis es:

throw objeto_excepcion;

Por ejemplo, supongamos que vamos a crear una clase que proporcione una serie de métodos para realizar operaciones matemáticas. Queremos que el método que realiza la división entre dos números provoque una excepción del tipo definido anteriormente en caso de que el denominador sea 0:

```
public class Operaciones {
	public int division(int numerador, int denominador){
		if(denominador==0){
			throw new OperacionNoValidaException();
		}
		return numerador/denominador;
	}
}
```

Sin embargo, la clase anterior no compila por un error en la instrucción throw. Y es que, al lanzar una excepción verificada, el compilador obliga a capturarla, pero claro, no tiene sentido que la capturemos en el mismo lugar en el que la lanzamos, por lo que el método deberá propagar la excepción al punto de llamada.

La propagación de una excepción se realiza declarando ésta en la cabecera del método, a través de la instrucción **throws** (no confundir con throw). La clase, por tanto, debería escribirse:

```
public class Operaciones {
	public int division(int numerador, int denominador)
				throws OperacionNoValidaException{
		if(denominador==0){
			//lanza la excepción si no puede dividir
			throw new OperacionNoValidaException();
		}
		return numerador/denominador;
	}
}
```

Captura de la excepción

La clase que haga uso del método *division()* será la que se tendrá que encargar de capturar y tratar la excepción:

```
Operaciones op=new Operaciones();
try {
	System.out.println(op.division(3, 5));
} catch (OperacionNoValidaException e) { //captura de la excepción
	System.out.println(e.getMessage());
}
```