Excepciones

OCTUBRE 2015

Introducción

Una **excepción** es un error que ocurre en el tiempo de ejecución de un programa

En la ejecución de programas se producen errores, con diferentes niveles de gravedad

En lenguajes de programación de alto nivel (p.ej. Java) es posible gestionar y manejar estos errores de forma estructurada y controlada, manipulando los errores en tiempo de ejecución

Una de las ventajas del manejo de excepciones es se automatiza gran parte del código para **manejar errores** que previamente han sido ingresados dentro de un programa

Antes de las excepciones…. (1/2)

En varios lenguajes de programación, los códigos de error son retornados cuando un método falla

Los códigos regresados por un método resulta ser tediosa y propensa a un error, por ejemplo:

```
double calcularRaizCuadrada(double p){
   //En la función existe una variable resultado que regresa un número mayor o igual a 0 si la función se ejecutó
   correctamente, sin embargo, regresa -1 si ocurrió un error en el proceso, y regresa -2 si el valor del parámetro p es
   un número menor a 0
   return resultado;

}
```

Desde el lugar donde se invoca dicho método habría que verificar el resultado regresado:

```
int main(…..){
            double r = calcularRaizCuadrada(-1);
            if (r ==-1){
                        //La función tuvo un error para calcular raíz cuadrada: Error
            }else if (r == -2){
                        //El número enviado de parámetro es menor a 0: Error
            }else{
                        //El cálculo se hizo correctamente

            }
            return 0;
}
```

Esta forma de controlar el error resulta ineficiente, y puede ser confuso en el código de un programa

Antes de las excepciones…. (2/2)

Otro ejemplo:

```
double dividir(int dividendo, int divisor){
   //En la función existe una variable resultado que regresa uno si la división se realizó correctamente, sin embargo,
   regresa -1 si ocurrió un error en el proceso,  y regresa -2 si el valor del parámetro divisor es un número igual 0
   (no es posible la división entre 0)
   double resultado = dividendo/divisor;
   return resultado;

}
```

Desde el lugar donde se invoca dicho método habría que verificar el resultado regresado:

```
int main(…..){
            double r = dividir(10,0);
            if (r ==-1){
                        //La función tuvo un error para calcular la división
            }else if (r == -2){
                        //El número enviado de parámetro es 0: Error
            }else{
                        //El cálculo se hizo correctamente

            }
            return 0;
}
```

Nuevamente esta gestión de error resulta ineficiente

Manejador de Excepciones

El manejo de excepciones, controla los errores de líneas de flujo, permitiendo que un programa defina un código de bloque de nombre **manejador de excepciones**

El manejador de excepciones es ejecutado automáticamente cuando ocurre un error, sin ser necesario comprobar manualmente el éxito o falla de cada operación o método específico

- Si ocurre un error, éste será procesado por el manejador de excepciones

En Java, se definen excepciones estándar para errores comunes en programas, como dividir por cero (**ArithmeticException**) o la de un archivo no encontrado (**FileNotFoundException**)

- Para responder a estos errores, el programa debe buscar y manejar estas excepciones
- Ser un programador exitoso en Java significar estar capacitado para navegar en el subsistema de manejo de excepciones de Java

Jerarquía de Excepciones en Java

En Java, todas las excepciones son representadas por clases

Todas las clases de excepción son derivadas de la clase **Throwable**

Por lo tanto, cuando en el programa ocurre una excepción, un objeto de alguna clase excepción es generado

Hay dos subclases directas de **Throwable: Exception** y **Error**

- Error: Son todos los errores que ocurren en la misma máquina virtual de Java, y no en el programa propio. Estos tipos de excepciones, están fuera de nuestro control, y nuestro programa generalmente no tratará con ellos
- Exception: Son los errores ocurridos por una división por cero, límites de un arreglo o errores de un archivo. En general, nuestro programa debe manejar excepciones de este tipo.

Fundamentos del Manejo de Excepciones (1/2)

El manejo de excepciones en Java es dirigido por cinco palabras clave: **try, catch, throw, throws** y **finally**

Estas palabras claves forman parte de un subsistema interrelacionado donde el uso de un implica el uso de otra.

Las declaraciones del programa que se desea monitorear en busca de una excepción están contenidas en el bloque **try**, si una excepción ocurre dentro del bloque **try**, esta es lanzada

- El código de nuestro programa puede atrapar esta excepción utilizando **catch** y manejarla de alguna manera racional
- Las excepciones generadas por el sistema son automáticamente lanzadas por el sistema de tiempo de ejecución de Java

Fundamentos del Manejo de Excepciones (2/2)

Para lanzar manualmente una excepción se utiliza la palabra clave **throw**

En algunos casos, una excepción que se lance fuera de un método se debe especificar por medio de una cláusula **throws**
Cualquier código que deba ser ejecutado después de haber salido de un bloque **try** se coloca en un bloque **finally**, no obstante, el bloque **finally** es opcional, ejemplo:

```
try{
        //Bloque de código para el control de errores
}catch(ExcepcionTipo1 exObj){
        //Manejador de excepciones para ExcepcionTipo1
}catch(ExcepcionTipo2 exObj){
        //Manejador de excepciones para ExcepcionTipo2
}
```

*Cuando una excepción es lanzada, es atrapada por su correspondiente sentencia catch, y luego la excepción es procesada.
Como se puede observar, en el formato general puede haber más de una sentencia catch asociada con un try, y el tipo de excepción determina la sentencia catch ejecutada.

Si el tipo de excepción especificada por una sentencia catch coincide con la excepción, la sentencia catch se ejecuta (y todas las otras son omitidas), cuando la excepción es atrapada, exObj recibirá su valor.
Si no se lanza ninguna excepción, entonces el bloque try finaliza normalmente y todas las sentencias catch son omitidas.
Las sentencias catch serán ejecutadas sólo si una excepción es lanzada.

Ejemplo Sencillo en Java (1/2)

En este ejemplo se intenta dividir entre 0

```
public class DemoZero{
	public static double dividir(int dividendo, int divisor){
		double resultado = dividendo/divisor;
		return resultado;
	}

	public static void main(String args[]){
		try{
			double resultado = DemoZero.dividir(10,0);
			System.out.println("Resultado: " + resultado);
		}catch(ArithmeticException e){
			System.out.println("División Inválida entre 0");
		}
	}
}
```

```
C:\Programacion\práctica 5 - Flujos>javac DemoZero.java

C:\Programacion\práctica 5 - Flujos>java DemoZero
Divisi¾n Invβlida entre 0

C:\Programacion\práctica 5 - Flujos>
```

Ejemplo Sencillo en Java (2/2)

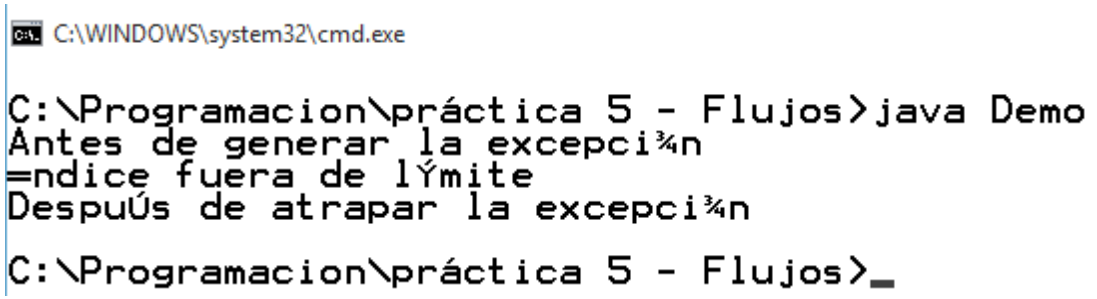
En este ejemplo se intenta acceder a una posición de un arreglo fuera de sus límites

Cuando esto ocurre la máquina virtual de Java lanza un **ArrayIndexOutBoundsException**

En el ejemplo se controla esta excepción

```
public class Demo{
    public static void main(String args[]){
            int nums[] = new int[4];
            try{
                        //Antes de generar la excepción
                        System.out.println("Antes de generar la excepción");
                        nums[10] = 20;
            }catch(ArrayIndexOutOfBoundsException e){
                        //Atrapa la excepción
                        System.out.println("Índice fuera de límite");
            }
            System.out.println("Después de atrapar la excepción");
  }

}
```



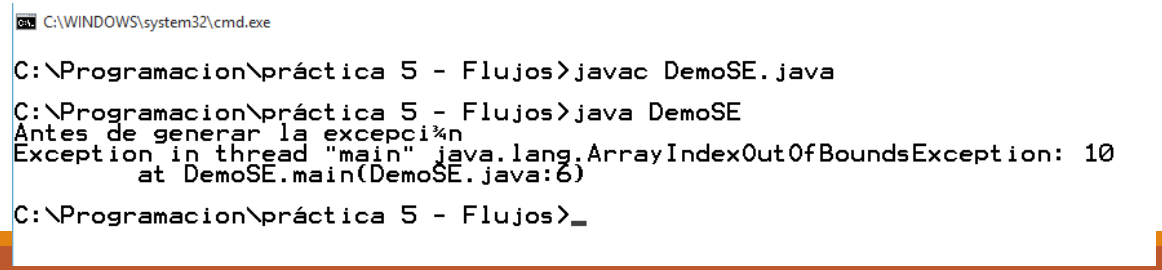
Consecuencias de NO Atrapar una Excepción

Atrapar una excepción estándar de Java, como se hizo anteriormente, tiene un lado bueno:

- Previene la terminación anormal del programa
- Si un programa no atrapa una excepción entonces ella será atrapada por la **máquina virtual**
- El problema es que el manejador de excepciones por defecto de la máquina virtual de java termina la ejecución del programa y muestra un rastreo de pila y un mensaje de error, por ejemplo en esta versión sin manejo de la excepción **ArrayIndexOutBoundsException**:

```
public class Demo{
	public static void main(String args[]){
		int nums[] = new int[4];
		//Antes de generar la excepción
		System.out.println("Antes de generar la excepción");
		nums[10] = 20;
		System.out.println("Después de atrapar la excepción");
	}
}
```

Aunque no se maneje la excepción un mensaje de error es mostrado
Sin embargo, este mensaje es útil para el programador mientras depura, pero no es ideal que los otros lo vean (otros desarrolladores o usuarios finales)



Uno de los beneficios clave del manejo de excepciones consiste en permitir al programa responder a un error, y luego continuar

Atrapar Excepciones Subclase

Hay un punto importante en relación con las sentencias de catch múltiples relativas a subclases

Una cláusula catch para un superclase también coincidirá con cualquier de sus subclases

Por ejemplo, puesto que la superclase de todas las excepciones es **Throwable**, para atrapar todas las excepciones posibles, se podría utilizar **Throwable**

Si se desea atrapar excepciones de ambos tipos, superclases y subclases, se coloca primero la subclase en la secuencia catch, de lo contrario la superclase en **catch** también atrapará todas la excepciones derivadas y generará código inalcanzable puesto que la subclase en catch nunca se ejecutaría, ocasionando un error de compilación

Los bloques **try** pueden ser anidados. Una excepción generada en el interior del bloque **try** que no es atrapada por un **catch** asociado con ese **try** es propagada al exterior del bloque **try**, por ejemplo, si el **ArrayIndexOutBoundsException** no es atrapado por un bloque **try** interno, este sería atrapado por un **try** externo

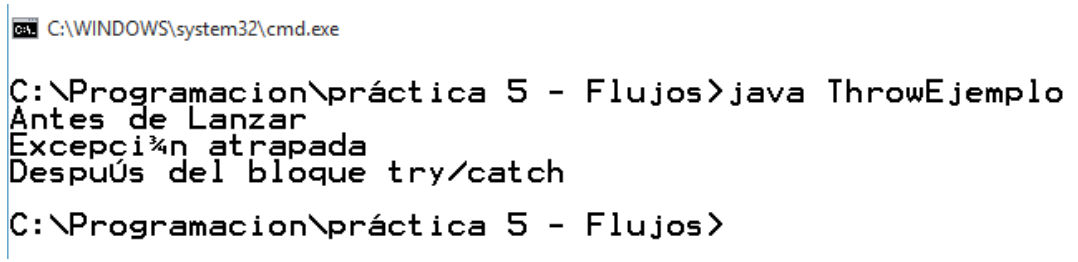
Lanzamiento de Excepción

Los ejemplos anteriores han estado atrapando excepciones generadas automáticamente por la máquina virtual de java, sin embargo, es posible lanzar manualmente una excepción por medio de la sentencia **throw**, con la siguiente estructura:

- **throw** *objetoExcepción* *// ObjetoExcepción debe ser un objeto de una clase de excepción derivada de Throwable*

Ejemplo para lanzar manualmente una **ArithmeticException**:

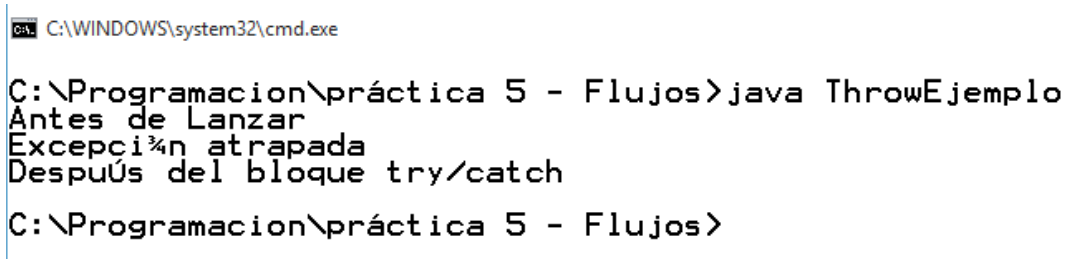
```
class ThrowEjemplo{
 public static void main(String args[]){
        try{
                    System.out.println("Antes de Lanzar");
                    throw new ArithmeticException();
        }catch(ArithmeticException e){
                    //Atrapa excepción
                    System.out.println("Excepción atrapada");
        }
        System.out.println("Después del bloque try/catch");
 }
}
```



Lanzamiento de Excepción

Otro ejemplo para lanzar manualmente una **ArithmeticException**:

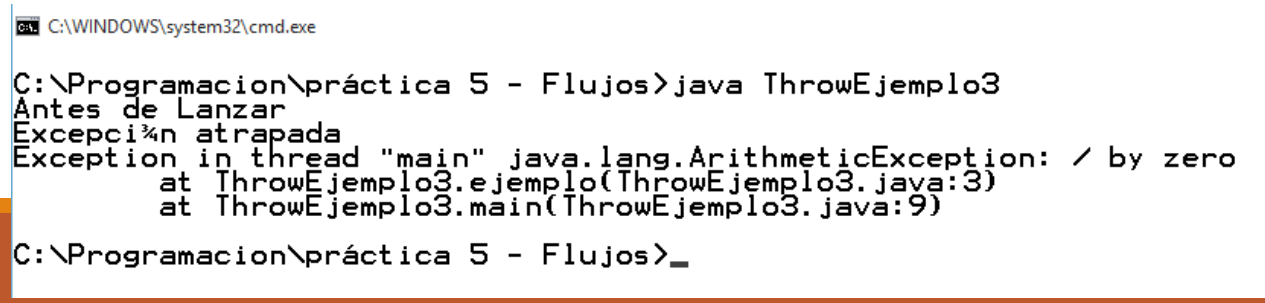
```
class ThrowEjemplo2{
	public void ejemplo(){
		throw new ArithmeticException();
	}
	public static void main(String args[]){
		try{
			System.out.println("Antes de Lanzar");
			ThrowEjemplo2 ejemplo = new ThrowEjemplo2();
			ejemplo.ejemplo();
		}catch(ArithmeticException e){
			//Atrapa excepción
			System.out.println("Excepción atrapada");
		}
		System.out.println("Después del bloque try/catch");
	}
}
```



Re-lanzamiento de Excepción

Ejemplo para re-lanzar manualmente una **ArithmeticException**:

```
class ThrowEjemplo3{
	public void ejemplo(){
		int x = 10/0;
	}
	public static void main(String args[]){
		try{
			System.out.println("Antes de Lanzar");
			ThrowEjemplo3 ejemplo = new ThrowEjemplo3();
			ejemplo.ejemplo();
		}catch(ArithmeticException e){
			//Atrapa excepción
			System.out.println("Excepción atrapada");
			throw e;
		}
		System.out.println("Después del bloque try/catch");
	}
}
```



Uso de throws

En algunos casos, si un método genera una excepción que no maneja, debe declarar esa excepción con la cláusula **throws**

```
modificador_acceso tipo_retorno nombre_método(parámetros) throws lista_excepciones{
        //Cuerpo

}
```

Java:

```
public class SalidaOutErr{
        public static void main(String args[]) throws FileNotFoundException{
                System.setOut(new PrintStream(new FileOutputStream("salida_normal.txt")));
                System.setErr(new PrintStream(new FileOutputStream("salida_error.txt")));
                System.out.println("Hola!");
                System.err.println("Error");
        }
}
```

La lista de excepciones son separadas por comas, y son las que el método puede lanzar fuera de sí mismo

Ejemplo de throws (1/3)

Hasta este momento podría preguntarse porqué no se necesitó especificar la cláusula **throws** para algunos ejemplos anteriores que lanzaron excepciones fuera de los métodos:

```
public class ThrowEjemplo4{
      public static void main(String args[]){
            System.out.println("Antes de Lanzar");
            throw new ArithmeticException();
      }
}
```

Ejemplo de throws (2/3)

La respuesta es porqué las excepciones que son subclases de **Error** o **RuntimeException** no necesitan ser especificadas en una lista **throws**, sin embargo, todos los otros tipos de excepciones necesitan ser declarados (p. ej. `FileNotFoundException`). En caso de no hacerlo se tendría un error de compilación:

```
import java.io.*;
public class ThrowEjemplo5{
	public static void main(String args[]){
		System.out.println("Antes de Lanzar");
		throw new FileNotFoundException();
	}
}
```

```
C:\Programacion\práctica 5 - Flujos>javac ThrowEjemplo5.java

C:\Programacion\práctica 5 - Flujos>javac ThrowEjemplo5.java
ThrowEjemplo5.java:5: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
                throw new FileNotFoundException();
                ^
1 error
```

Ejemplo de throws (3/3)

Para que el código mostrado en la diapositiva anterior compile correctamente es necesario usar la cláusula **throws** en la firma del método:

```
import java.io.*;
public class ThrowEjemplo5{
      public static void main(String args[]) throws FileNotFoundException{
            System.out.println("Antes de Lanzar");
            throw new FileNotFoundException();
      }
}
```

Finally

Algunas veces se deseará definir un bloque de código que se ejecute cuando se abandona el bloque **try/catch**, por ejemplo:

- Una excepción podría causar un error que termine el método actual, lo cual provocaría su retorno prematuro, sin embargo, ese método puede tener abierto un archivo o una conexión de red que necesita ser cerrada. Estas circunstancias son convenientes de utilizar en **finally**

- La estructura de un **try/catch** que incluye un **finally** es la siguiete:

```
try{
  //Código Try
}catch(ExcepcionTipo1 e){
  //Manejador ExcepcionTipo1
}catch(ExcepcionTipo2 e){
  //ManejadorExcepcionTipo2
}finally{

}
```

El bloque finally será ejecutado cada vez que la ejecución abandone un bloque try/catch, sin importar qué condiciones lo causen

Ejemplo con finally

```
public class ThrowEjemplo6{
        public void ejemplo(){
                int x = 10/0;
        }
        public static void main(String args[]){
                try{
                        System.out.println("Antes de Lanzar");
                        ThrowEjemplo3 ejemplo = new ThrowEjemplo3();
                        ejemplo.ejemplo();
                }catch(ArithmeticException e){
                        //Atrapa excepción
                        System.out.println("Excepción atrapada");
                        throw e;
                }finally{
                        System.out.println("En Finally");
                }
                System.out.println("Después del bloque try/catch");
        }
}
```

```
C:\WINDOWS\system32\cmd.exe

C:\Programacion\práctica 5 - Flujos>java ThrowEjemplo6
Antes de Lanzar
Excepci¾n atrapada
En Finally
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ThrowEjemplo3.ejemplo(ThrowEjemplo3.java:3)
        at ThrowEjemplo6.main(ThrowEjemplo6.java:9)

C:\Programacion\práctica 5 - Flujos>
```