Excepciones en Java

Excepciones en Java

Las excepciones en Java son eventos que pueden alterar el flujo normal de ejecución de un programa. Java proporciona un mecanismo para manejar estas excepciones y asegurar que el programa pueda responder adecuadamente a situaciones inesperadas.

Básicamente una excepcion es un Objeto descendiente de la clase **java.lang.Object**, podemos pensar en ellas como una condición excepcional en nuestro sistema el cual altera la correcta ejecución del mismo, las excepciones nos indican que hay algo anómalo, inconsistente o simplemente un Error, lo cual impide que el sistema se ejecute como debería de ser...

Tal vez se preguntaran si ¿pero Anómalo, inconsistente o Error no es básicamente lo mismo?..... podría ser, pero en este enfoque no necesariamene lo es, ya que lo que vamos a conocer como una excepcion no siempre es un error (hablando como excepcion en general, ya que en java una **Exception** es muy diferente a un **Error**), muchas veces necesitaremos trabajar con excepciones controladas para indicar alguna inconsistencia en nuestro sistema que podría provocar errores.

1. Conceptos Básicos

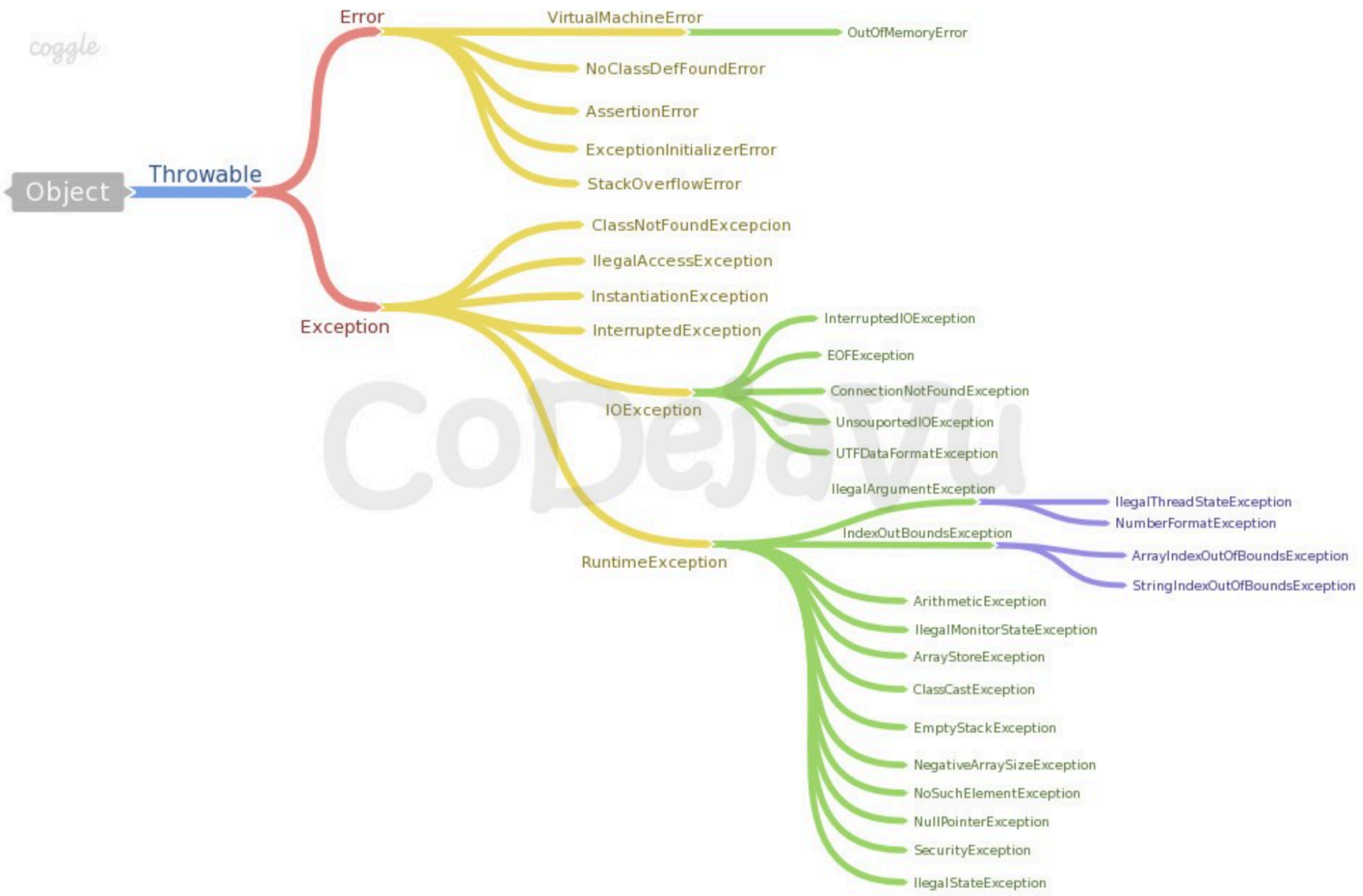
- **Excepción**: Un evento que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de las instrucciones.
- **Manejo de Excepciones**: Proceso de responder a la aparición de excepciones. Esto se realiza mediante los bloques `try`, `catch`, `finally` y `throw`.

A modo de ejemplo, podemos encontrarnos con el famoso **NullPointerException,** el cual nos indica que un objeto se encuentra vació, pero esto no es un error ya que nosotros podemos trabajar con objetos

null, entonces veamoslo como si la excepcion nos dijera "Es un objeto nulo y no se puede efectuar el proceso", mientras que hay errores como, **NoClassDefFoundError,** el cual nos indica que la maquina virtual de Java (**JVM**) no puede encontrar una clase que necesita, debido a por ejemplo que no encuentra un .class, esto si se maneja como un error en java

2. Jerarquía de Excepciones

Para hacer mas claridad sobre el tema veamos la Jerarquía de Excepciones de Java (puede que no se encuentren algunas excepciones, pero se contemplan las mas comunes)



La jerarquía básica es la siguiente:

```
java.lang.Object
  └── java.lang.Throwable
      ├── java.lang.Error
      └── java.lang.Exception
          ├── java.lang.RuntimeException
          └── (otras excepciones chequeadas)
```

Vemos que se tiene un claro árbol de herencia mediante el cual se pueden definir las categorías de Excepciones o de Error que se puede dar en el sistema.

Las excepciones en Java se derivan de la clase `Throwable`. Existen dos categorías principales de excepciones:

- **Checked Exceptions**: Excepciones que son verificadas en tiempo de compilación. El programador debe manejar estas excepciones explícitamente.
- **Unchecked Exceptions**: Excepciones que ocurren durante la ejecución. Incluyen `RuntimeException` y sus subclases.

3. Bloques `try` y `catch`

El manejo básico de excepciones se realiza utilizando los bloques `try` y `catch`.

```
try {
    // Código que puede lanzar una excepción
} catch (TipoDeExcepcion e) {
    // Código para manejar la excepción
}
```

Ejemplo:

Una división sobre 0 puede arrojar un error aritmético, por esa razón la forma de controlarla seria mediante el try catch

```
public class EjemploExcepcion {
    public static void main(String[] args) {
```

```
        try {
            int division = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Error: División por cero.");
        }
    }
}
```

4. Bloque `finally`

El bloque `finally` contiene código que se ejecuta siempre, independientemente de si se lanzó una excepción o no.

```
try {
    // Código que puede lanzar una excepción
} catch (TipoDeExcepcion e) {
    // Código para manejar la excepción
} finally {
    // Código que se ejecuta siempre
}
```

Ejemplo:

```
public class EjemploFinally {
    public static void main(String[] args) {
        try {
            int division = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Error: División por cero.");
        } finally {
            System.out.println("Este bloque siempre se ejecuta.");
        }
    }
}
```

La importancia de Prever!!!

Cuando se esta programando debemos tener claro que nuestro código no es perfecto, así tengamos mucha experiencia en desarrollo siempre esta la posibilidad de que algo falle, sea por nuestro código o por otros factores, por eso de la importancia de contemplar todo desde antes, posibles fallos o lo que pueda afectar el sistema.

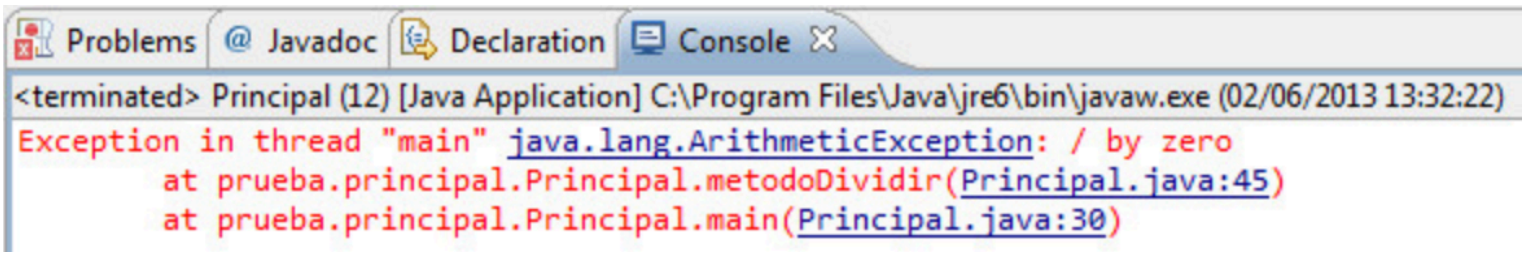
veamos nuevamente en detalle el ejemplo anterior sobre la división, que pasaría si no controlamos la excepción.

```
private void metodoDividir(int dividendo, int divisor){

  String resultado+=dividendo/divisor;
  System.out.println(resultado);

}
```

el **metodoDividir(int, int)** teóricamente esta bien, claro, a simple vista si tenemos : **dividendo = 4** y **divisor = 2** pues el resultado es **2,** básico, pero y si el **divisor es 0**? pues con ese caso puntual el resultado seria el siguiente.



Vemos que nos indican que se produjo una **ArithmeticException** debido a una división por cero, además se muestra cual fue la traza del error pasando por el método main hasta el **metodoDividir().**La anterior es una Excepcion simple, algo que se supone no debería pasar, es obvio, no se puede dividir por cero, o ¿no?pues no, **en programación no podemos asumir ni pensar así**, ya que muchas veces nos olvidamos de las cosas obvias y las pasamos por alto, el problema es que eso tan obvio puede detener toda la ejecución del programa.

Trabajando con try - catch - finally

Con los bloques Try - Catch podemos capturar y procesar una posible excepcion, evitando que el sistema se detenga sin necesidad cuando el motivo de esto puede ser corregido facilmente, la estructura básica es la siguiente.

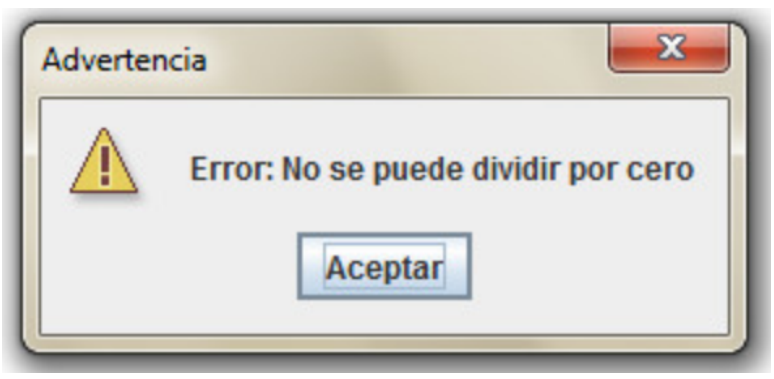
```
try {
  //Bloque de código que vamos a procesar
} catch(excepcion) {
  //Tratamiento que se le da a la posible excepción
} finally {
  //Bloque de código que se ejecutará despues del try o del catch
}
```

Apliquemos esto a nuestro ejemplo anterior.

```
private void metodoDividir(int dividendo, int divisor){
String resultado="";
 try {
   resultado+=dividendo/divisor;
  }catch (Exception e) {
   resultado="Se intentó dividir por cero";
   JOptionPane.showMessageDialog(null,"Error: No se puede dividir por cero ",
      "Advertencia",JOptionPane.WARNING_MESSAGE);
 }
   finally{
    System.out.println("Termino el proceso : el resultado es = "+resultado);
   }
}
```

Como vimos aplicamos la estructura de los bloques y de esta manera nos aseguramos que la excepción anterior fue controlada evitando que el sistema se detenga, en el **catch** podemos hacer el proceso que consideremos conveniente, ya sea solo informar del error o solicitar nuevos parámetros de entrada.

la salida es la siguiente:



Algunas Consideraciones.

Veamos un poco mas lo que debemos tener en cuenta cuando usamos estos bloques:

try: Aquí vamos a escribir todo el bloque de código que posiblemente llegue a lanzar unas excepción la cual queremos manejar, aquí va tanto el código como llamados a métodos que puedan arrojar la excepción.

En este bloque solo se detectara la primera excepcion lanzada, hay que tener en cuenta que por cada try se debe especificar un catch y/o un finally.

catch: en caso de que en el try se encuentre alguna excepción, se ingresara automaticamente al bloque catch donde se encontrara el código o proceso que queremos realizar para controlar la excepción.

Se pueden especificar cualquier cantidad de catch de ser necesario, estos deben ser ubicados después del try y antes del finally (en caso de que este ultimo se especifique), cada catch que se ponga debe manejar una excepcion diferente (no se puede repetir) y el orden de estos depende de la jerarquía de herencia que se tenga, ingresando al primer catch que pueda suplir la necesidad a corregir, por ejemplo.

```
try {
      //Sentencias con posibles errores;
    } catch(InterruptedException e){
      //manejo de la exepcion
    } catch(IOException e){
   //manejo de la exepcion
    } catch(Exception e){
   //manejo de la exepcion
    }
```

Si se genera una excepción en el try, se valida a cual de los 3 catch se ingresa, dependiendo si InterruptedException puede controlarlo se ingresa a esa, sino entonces a IOException o si no a la superClase que seria Exception (ver la jerarquía de herencia anterior).

En el ejemplo que del metodoDividir() trabajamos directamente con Exception e, de esta forma nos aseguramos que capture cualquier excepción, sin embargo se recomienda usar la jerarquía en los catch para poderle dar un mejor manejo.

finally : Este bloque es opcional, lo podremos si queremos ejecutar otro proceso después del try o el catch, es decir, siempre se ejecutara sin importar que se encuentre o no una excepción.

5. Lanzar Excepciones con `throw`

Puedes lanzar excepciones manualmente utilizando la palabra clave `throw`.

```
public class EjemploThrow {
    public static void main(String[] args) {
        try {
            validarEdad(15);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    public static void validarEdad(int edad) throws Exception {
        if (edad < 18) {
            throw new Exception("Edad no válida para votar.");
        }
    }
}
```

6. Declarar Excepciones con `throws`

Cuando un método puede lanzar una excepción, debes declararlo con la palabra clave `throws`.

```
public class EjemploThrows {
```

```
    public static void main(String[] args) {
        try {
            metodoQueLanzaExcepcion();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    public static void metodoQueLanzaExcepcion() throws Exception {
        throw new Exception("Esta es una excepción lanzada por el método.");
    }
}
```

7. Crear Excepciones Personalizadas

Puedes crear tus propias excepciones extendiendo la clase `Exception`.

```
class MiExcepcion extends Exception {
    public MiExcepcion(String mensaje) {
        super(mensaje);
    }
}

public class EjemploExcepcionPersonalizada {
    public static void main(String[] args) {
        try {
            metodoQueLanzaMiExcepcion();
        } catch (MiExcepcion e) {
            System.out.println(e.getMessage());
        }
    }

    public static void metodoQueLanzaMiExcepcion() throws MiExcepcion {
        throw new MiExcepcion("Esta es una excepción personalizada.");
    }
}
```

Conclusión

Las excepciones en Java son una herramienta poderosa para manejar errores y situaciones inesperadas en tus programas. Utilizando `try`, `catch`, `finally`, `throw` y `throws`, puedes escribir código robusto y resistente a fallos. Crear excepciones personalizadas te permite manejar casos específicos de error de manera más clara y manejable.

Instructor: Cristian David Henao Hoyos