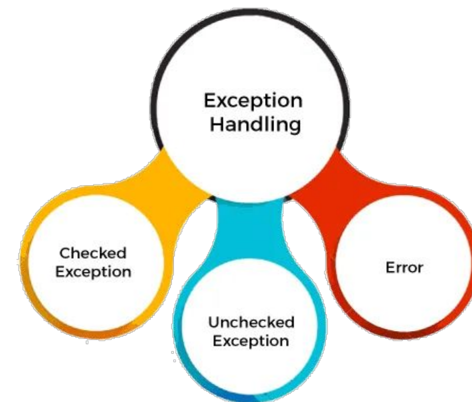


Gestión de excepciones

UNJu - Desarrollo y Arquitecturas Avanzadas

Ing. José Zapana

Concepto

- Una excepción es un evento que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de las instrucciones.
- El objetivo principal de las excepciones es separar la lógica del programa de la gestión de errores.

Clasificación

En función de su naturaleza una excepción puede ser:

- **Unchecked:**
  - No son verificadas en compilación; se detectan en tiempo de ejecución.
  - Generalmente representan errores de lógica del programador.
  - Heredan de RuntimeException.
  - Ejemplos:
    - NullPointerException
    - ArithmeticException (división por cero)
    - ArrayIndexOutOfBoundsException
- **Checked:**
  - El compilador obliga a declararlas (con throws) o manejarlas (con try-catch).
  - Son verificadas en tiempo de compilación.
  - Representan condiciones previsibles que pueden ocurrir fuera del control del programa.
  - Ejemplos:
    - IOException (error de entrada/salida)
    - SQLException (errores en bases de datos)
    - FileNotFoundException

Excepciones Unchecked

- **ArrayIndexOutOfBournException:** intento de acceder fuera de los límites de un array
- **NullPointerException:** Acceso a métodos de un objeto con referencia null
- **SecurityException:** producida por una violación de seguridad
- **ClassCastException:** error de conversión de tipos de datos
  - `Object ob = new String("34");`
  - `Integer nro = (Integer) ob;`
- **ArithmeticException:** operación aritmética incorrecta.
  - `int n = 5 / 0;`
  - `double r = 3 / 0.0;`
- **IllegalArgumentException:** un método recibe como parámetro un valor no válido: `thread.sleep(-100)`

Errores

- A diferencia de una excepción, un error es una situación que se produce en un programa de la que este no se puede recuperar, como fallo en la JVM, falta de espacio en memoria, etc.
- Sin embargo los errores también están por clases que heredan de Error: OutOfMemoryError, StackOverflowError, InternalError, etc

Ventajas sobre las excepciones

- **Separación de lógica y errores:** el código principal se concentra en la funcionalidad, mientras que el manejo de errores se gestiona aparte.
- **Reutilización:** se pueden definir jerarquías de clases de excepción para representar diferentes tipos de errores, facilitando la extensión del sistema.
- **Recuperación controlada:** el programa puede continuar ejecutándose tras un fallo, en lugar de detenerse abruptamente.

Limitaciones de las Excepciones

- **Sobrecarga en el rendimiento:** lanzar y capturar excepciones implica un costo adicional en tiempo de ejecución.
- **Uso inapropiado:** si se utilizan para controlar flujo normal (y no errores), el código se vuelve ineficiente y difícil de entender.
- **Complejidad excesiva:** demasiados tipos de excepciones pueden generar una jerarquía difícil de mantener.
- **Dependencia del programador:** el buen manejo depende de cómo se definan y capturen las excepciones; un catch mal usado puede ocultar errores críticos.

Clases de excepciones

Throwable

Error

Exception

IOException

ClassNotFoundException

RuntimeException

IllegalArgumentException

NullPointerException

ArithmeticException

NumberFormatException

Captura de excepciones

```
try{
    //instrucciones
}catch(TipoExcepcion1 ex){
    //tratamiento de la excepción
}catch(TipoExcepcion2 | TipoExcepcion2 ex){
    //tratamiento de las dos excepciones
}
```

- Si los catch tienen excepciones en relación de herencia, las subclases deben ir antes que las superclases
- Si no están en relación de herencia se pueden agrupar en un multicatch mediante |

Métodos de exception

Todas las clases de excepción heredan los siguientes métodos de Exception:

- **String getMessage():** devuelve una cadena de caracteres con un mensaje de error asociado a la excepción
- **void printStackTrace():** genera un volcado de error que es enviado a la consola

Bloque finally

- Se ejecuta siempre se produzca o no la excepción

```
try{
   int n = 4 / 0;
}catch(ArithmeticException ex){
    System.out.println(“Error división por cero”);
    return;
}finally{System.out.println(“Final”)}
```

- Si se produce una excepción y no hay ningún catch para capturarla, se propagara la excepción al punto de llamada, pero antes ejecutará el bloque finally

Propagación de Excepciones

Cuando ocurre una excepción:

- Se crea un objeto de excepción en el lugar del fallo.
- La JVM busca un bloque catch adecuado en el mismo método.
- Si no lo encuentra, la excepción se propaga al método que lo llamó.
- El proceso se repite recorriendo la pila de llamadas (call stack).
- Si ningún método maneja la excepción → la JVM la captura y termina el programa mostrando la traza (stack trace).

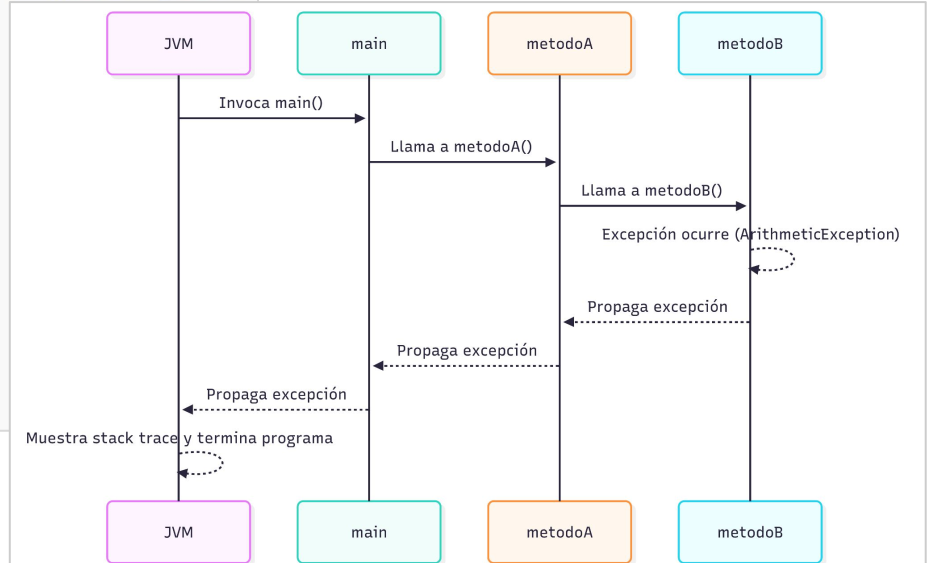
Ejemplo

```java
public class PropagacionEjemplo {

    public static void main(String[] args) {
        metodoA();
        System.out.println("Fin del programa"); // No se ejecuta si no se captura
    }

    static void metodoA() {
        metodoB();
    }

    static void metodoB() {
        int resultado = 10 / 0; // ArithmeticException
    }
}
```



Propagación de una excepción

- Si un método que debe capturar una excepción y no desea hacerlo, puede propagarla al lugar de la llamada del método
- Se debe declarar la excepción en la cabecera del método con la instrucción throws:

```
void metodo(){
    BufferedReader bf = …;
    try{
        String s = bf.readLine()
    }catch(IOException ex){
    }
}
```

Propagación

```
void metodo()throws IOException{
    BufferedReader bf = …;
    String s = bf.readline();
}
```

Ejemplo

Devolver el valor de un elemento de un vector dado su índice

```
/**
 *
 * @param vector: con elementos enteros
 * @param indice: posición desde donde se quiere obtener un valor
 * @return: elemento del vector en el indice indicado
 * @throws java.lang.IndexOutOfBoundsException
 */
public int getElementoVector(int[]vector, int indice)
                                         throws IndexOutOfBoundsException{

       if(indice < 0)
           throw new IndexOutOfBoundsException("Indice incorrecto");

       return vector[indice];
}
```

Excepciones personalizadas

Se puede crear una excepción personalizada definiendo una clase que herede de Exception

```
class TestException extends Exception{

}
```

```
class C1{
 public void metodo() throws TestException{
        …
        throw TestException();
 }
}
```

```
C1 c = new C1();
try{
   c.metodo();
}catch(TestException ex){
    …
}
```

Ejemplo: Realizar extracciones de una cuenta bancaria

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

```
public class CuentaBancaria(){
      public void retirar(double cantidad)throws SaldoInsuficienteException,
      IllegalArgumentException{
         if(cantidad < 0)
            throw new IllegalArgumentException("La cantidad debe ser mayor a cero");
         if (cantidad > saldo)
            throw new SaldoInsuficienteException("Saldo insuficiente para el retiro");
         saldo -= cantidad;
      }
}
```

Ejemplo (continuación)

```
public void metodo()throws {
    CuentaBancaria cuenta = new CuentaBancaria(1000);
    try{
        cuenta.retirar(500);
        cuenta.retirar(800);
    }catch(SaldoInsuficienteException ex){
        System.err.println(ex.getMessage());
    }catch(IllegalArgumentException ex){
        System.err.println(ex.getMessage());
    }
}
```

Mockito Introducción

- Es un framework de código abierto para pruebas unitarias en Java.
- Su objetivo principal es ayudar a crear objetos simulados o "mocks" de las dependencias de una clase para que puedas probarla de forma aislada

¿Cómo Funciona Mockito?

- El funcionamiento de Mockito se puede resumir en 3 pasos clave, que corresponden a las 3 "A's" del testing: Arrange, Act, y Assert.

Mockito - Funcionamiento

- **Arrange (Preparar):** con el @mock se crea una versión simulada de un objeto Definir comportamiento: indicarle al mock qué debe hacer cuando se llama a sus métodos. Esto se hace con la sintaxis when().thenReturn().
- **Actuar (Ejecutar el Método)**: simplemente llamar al método de la clase que se está probando, sin preocuparse por las dependencias reales.
- **Assert (Acertar / verificar):** Verificar las Interacciones:
  Este es el paso más importante en Mockito. En lugar de verificar el estado final de un objeto, se verifica que los métodos del mock fueron llamados de la manera correcta. Esto se hace con la sintaxis verify().

Referencias

- Oracle Class Exception
- Java - Exceptions