Guía práctica de estudio 10: Excepciones y errores

Elaborado por:
M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:
M.C. Alejandro Velázquez Mena

Guía práctica de estudio 10: Excepciones y errores

Objetivo:

Identificar bloques de código propensos a generar errores y aplicar técnicas adecuadas para el manejo de situaciones excepcionales en tiempo de ejecución.

Actividades:

- Capturar excepciones de un bloque de código seleccionado.
- Capturar errores de un bloque de código seleccionado.

Introducción

La ley de Pareto, también conocida como la regla 80/20, establece que el 80 % de las consecuencias proviene del 20 % de las causas.

Una máxima en el desarrollo de software dicta que el 80 % del esfuerzo (en tiempo y recursos) produce el 20 % del código. Así mismo, en términos de calidad, el 80 % de las **fallas** de una aplicación son producidas por el 20 % del código. Por lo tanto, detectar y manejar errores es la parte más importante en una aplicación robusta.

Una aplicación puede tener diversos tipos de **errores**, los cuales se pueden clasificar en tres grandes grupos:

- **Errores sintácticos**: Son todos aquellos errores que se generan por **infringir las normas de escritura de un lenguaje**: coma, punto y coma, dos puntos, palabras reservadas mal escritas, etc. Normalmente son detectados por el compilador o el intérprete (según el lenguaje de programación utilizado) al procesar el código fuente.

- **Errores semánticos (o lógicos)**: Son errores más sutiles, se producen cuando la sintaxis del código es correcta, pero la **semántica o significado** no es el que se pretendía. Los compiladores e intérpretes sólo se ocupan de la estructura del código que se escribe y no de su significado, por lo tanto, un error semántico puede hacer que el programa termine de forma anormal, con o sin un mensaje de error.

No todos los errores semánticos se manifiestan de una forma obvia. Un programa puede continuar en ejecución después de haberse producido errores semánticos, pero su estado interno puede ser distinto del esperado. Quizá las variables no contengan los datos correctos, o bien es posible que el programa siga un camino distinto del pretendido. Eventualmente, la consecuencia será un resultado incorrecto. Estos errores se denominan lógicos, ya que, aunque el programa no se bloquea, la lógica que representan contiene un error.

- **Errores de ejecución**: Son errores que se presentan cuando la aplicación se está ejecutando. Su origen puede ser diverso, se pueden producir por un **uso incorrecto del programa** por parte del usuario (si ingresa una cadena cuando se espera un número), o se pueden presentar debido a **errores de programación** (acceder a localidades no reservadas o hacer divisiones entre cero), o debido a algún **recurso externo** al programa (al acceder a un archivo o al conectarse a algún servidor o cuando se acaba el espacio en la pila (stack) de la memoria).

Un **error en tiempo de ejecución** provoca que la aplicación termine abruptamente. Los lenguajes orientados a objetos proveen mecanismos para **manejar errores de ejecución**.

NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.

Excepciones

El término **excepción** hace referencia una condición excepcional que cuando ocurre **altera el flujo normal** del programa en ejecución. Estos errores pueden ser generados por la lógica del programa, como un índice de un arreglo fuera de su rango, una división entre cero, etc., o errores generados por los propios objetos que denuncian algún tipo de estado no previsto o condición que no pueden manejar.

Desde el punto de vista del tratamiento de una excepción dentro de un programa, hay que tener en cuenta que todas estas clases de excepción se dividen en dos grandes grupos:

- **Excepciones marcadas**
  - Aquellas cuya captura es obligatoria.
- **Excepciones no marcadas**
  - Las excepciones en tiempo de ejecución (RuntimeException y sus subclases). No es obligatorio capturarlas.

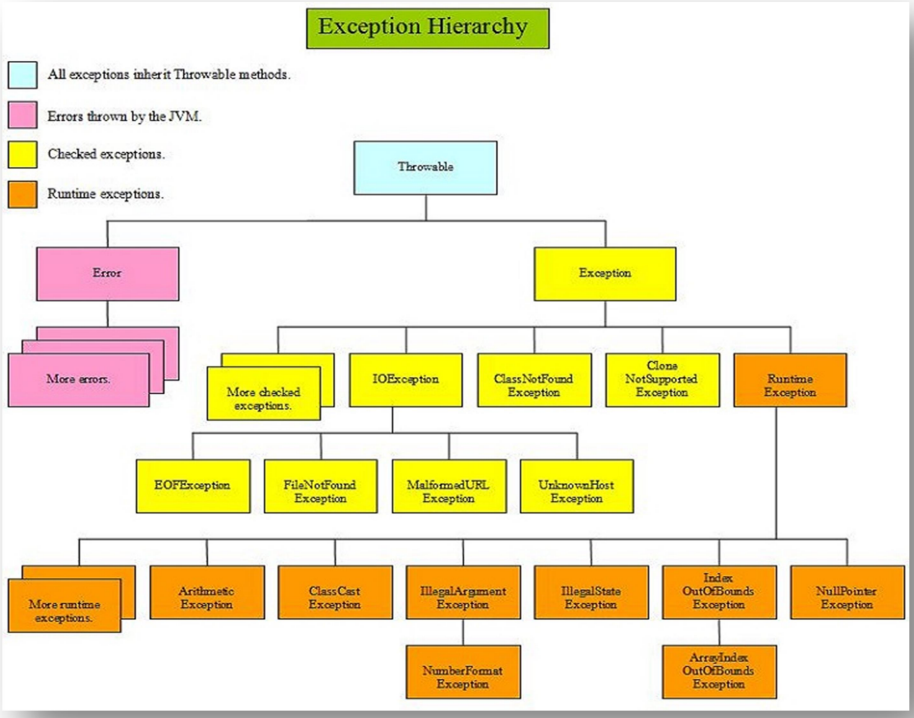

Figura 1. Jerarquía de clases de las Excepciones y Errores.

En Java, cuando un evento excepcional ocurre se dice que se **lanza una excepción**. Las **excepciones** son el mecanismo mediante el cual se pueden controlar los errores producidos en tiempo de ejecución. El código responsable de hacer alguna acción cuando se arroja una excepción es llamado manejador de excepciones y lo que hace es **cachar la excepción lanzada**.

Manejo de Excepciones

Para **manejar una excepción** se utilizan las palabras reservadas **try** y **catch**. El bloque *try* es utilizado para definir el bloque de código en el cual una excepción pueda ocurrir. El o los bloques *catch* son utilizados para definir un bloque de código que maneje la excepción.

Las excepciones son **objetos** que contienen información del error que se ha producido. Heredan de la clase *Exception* que a su vez hereda de la clase *Throwable*.

Ejemplo:

```
public class TryCatchFinally {

    public static void main(String[] args) {

        try {
            String mensajes[] = {"Primero", "Segundo",  "Tercero" };
            for (int i=0; i<=3; i++)
                System.out.println(mensajes[i]);

        } catch ( ArrayIndexOutOfBoundsException e ) {
            System.out.println("Error: apuntador fuera del rango del arreglo.");
        }
    }
}
```

Si no se captura la excepción interviene un manejador por defecto que normalmente imprime información que ayuda a encontrar dónde y cómo se produjo el error, sin embargo, como la excepción no es controlada, ésta se propaga hasta llegar al método principal y termina abruptamente el programa.

Ejemplo:

```
public class TryCatchFinally {

    public static void main(String[] args) {
        String mensajes[] = {"Primero", "Segundo",  "Tercero" };
        for (int i=0; i<=3; i++)
        System.out.println(mensajes[i]);
  }

}
```

La palabra reservada **finally** permite definir un tercer bloque de código dentro del manejador de excepciones. Este bloque le indica al programa las instrucciones a ejecutar de manera independiente de los bloques *try-catch*, es decir, si el código del bloque *try* se ejecuta de manera correcta, entra al bloque *finally*; si se genera un error, después de ejecutar el código del bloque *catch* ejecuta el código del bloque *finally*.

Ejemplo:

```
public class TryCatchFinally {

    public static void main(String[] args) {

    try {
        float equis = 5/0;
        System.out.println("Equis = " + equis);
    } catch ( ArithmeticException e ) {
        System.out.println("Error: división entre cero.");

    } finally {
        System.out.println("A pesar de todo, se ejecuta el bloque finally.");
    }
  }
}
```

Sin generar excepción:

```
public class TryCatchFinally {

    public static void main(String[] args) {

    try {
        float equis = 5/2;
        System.out.println("Equis = " + equis);
    } catch ( ArithmeticException e ) {
        System.out.println("Error: división entre cero.");

    } finally {
        System.out.println("A pesar de todo, se ejecuta el bloque finally.");
    }
  }
}
```

Cómo ya se mencionó, el bloque *catch* permite capturar excepciones, es decir, manejar los errores que genere el código del bloque *try* en tiempo de ejecución impidiendo así que el programa deje de ejecutarse y posibilitando al desarrollador enviar el error generado.

Es posible tener **más de un bloque *catch*** dentro del manejador de excepciones, pero cuidando el orden de captura, es decir, las excepciones se deben acomodar de las más específicas a las más generales, como se muestra a continuación:

catch (ClassNotFoundException e) {...}
catch (IOException e) {...}
catch (Exception e) {...}

Si se invierte el orden, se capturan las excepciones de lo más general a lo más específico, todas las excepciones caerían en la primera (*Exception* es la más general) y no habría manera de enviar un error del tipo *IOExcepction* o *ClassNotFoundException*. Esta situación la detecta el compilador y no permite generar el *bytecode*, es decir, no compila.

Ejemplo:

```
public class TryCatchFinally {

    public static void main(String[] args) {

    try {
        int equis = 5/0;
        System.out.println("Equis = " + equis);
    } catch ( ArithmeticException e ) {
        System.out.println("Error: división entre cero.");
    } catch ( Exception e ) {
        System.out.println("Error: excepción general.");
    } finally {
        System.out.println("El bloque finally siempre se ejecuta.");
    }
  }
}
```

Propagación de excepciones

No es obligatorio tratar las excepciones dentro de un bloque manejador de excepciones, pero, en tal caso, se debe indicar explícitamente a través del método. A su vez, el método superior deberá incluir los bloques *try/catch* o volver a **pasar la excepción**. De esta forma se puede ir propagando la excepción de un método a otro hasta llegar al último método del programa, el método *main*. Ejemplo:

```
public class PropagaExcepcion {

    public static int miMetodo(int a, int b){
        int c =a / b;
        return c;
    }

    public static void main(String[] args) {
        try{
            int division= miMetodo(10, 0);
            System.out.println(division);
        } catch(ArithmeticException e){
            System.out.println("Excepcion aritmetica arrojada: " );
            e.printStackTrace();
        }

    }
}
```

Throws

Existen métodos que obligan a ejecutarse dentro de un manejador de excepciones (es decir, son marcadas), debido a que el método define que **va a lanzar** una excepción.

Para indicar que un método **puede lanzar una excepción** se utiliza la palabra reservada **throws** seguida de la o las excepciones que puede arrojar dicho método. La sintaxis es la siguiente:

[modificadores] valorRetorno nombreMetodo() ***throws*** *Excepcion1, Excepcion2 {*
// Bloque de código del método
}

Esta sintaxis obliga que al utilizar el método se deba realizar dentro de un manejador de excepciones o dentro de un método que indique que va a arrojar la misma excepción.

```
public class PropagaExcepcion {

    public static void main(String[] args) {
        try{
            int division = miMetodo(10, 0);
            System.out.println(division);
        } catch (ArithmeticException e) {
            System.out.println("Excepcion aritmetica arrojada");
        }
    }
    public static int miMetodo(int a, int b) throws ArithmeticException{
        int c = a / b;
        return c;
    }

}
```

Throw

Una **excepción se puede lanzar** de manera explícita (sin necesidad de que ocurra un error) creando una instancia de la misma y arrojándola mediante la palabra reservada **throw**. Una excepción se debe arrojar dentro de un manejador de excepciones o dentro de un método que indique que se va arrojar dicha excepción (o cualquier subclase). La sintaxis es la siguiente:

throw *new Excepcion();*

Ejemplo:

```
public class PropagaExcepcion {

    public static int miMetodo(int a, int b) throws ArithmeticException{
        if(b == 0){
            throw new ArithmeticException();
        }
        int c =a / b;
        return c;
    }

    public static void main(String[] args) {
        try{
            int division= miMetodo(10, 0);
            System.out.println(division);
        } catch(ArithmeticException e){
            System.out.println("Excepcion aritmetica arrojada: " );
            e.printStackTrace();
        }

    }
}
```

Excepciones propias

Cuando se desarrollan aplicaciones es común requerir excepciones que no están definidas en el API de Java, es decir, **excepciones propias del negocio**. Es posible crear clases que se comporten como excepciones (que se puedan arrojar), para ello sólo es necesario heredar de *Exception* o de *Throwable*. Estas excepciones se invocan y se pueden arrojar de la misma manera en la que se utilizan las excepciones del API, pero pueden generar información más concisa del problema debido a que se están programando a la medida del negocio.

Ejemplo:

Para una aplicación bancaria se desea lanzar una excepción propia cuando el saldo de una cuenta sea insuficiente para realizar una operación (por ejemplo un retiro).

Se crea la clase de excepción propia *SaldoInsuficienteException*:

```
public class SaldoInsuficienteException extends Exception {

    public SaldoInsuficienteException() {
        super("Saldo insuficiente");
    }

}
```

Se crea la clase *Cuenta*, la cual podrá lanzar una excepción de tipo *SaldoInsuficienteException* si se intenta retirar un monto mayor al saldo de la cuenta:

```
public class Cuenta {

    private double saldo;

    public Cuenta(){
        saldo=0;
    }

    public void depositar(double monto){
        System.out.println("Depositando " + monto);
        saldo += monto;
    }

    public void retirar(double monto) throws SaldoInsuficienteException{
        System.out.println("Retirando " + monto);
        if(saldo < monto)
            throw new SaldoInsuficienteException();
        else
            saldo -= monto;
    }

    public double getSaldo(){
        return saldo;
    }

}
```

Finalmente, para probar el funcionamiento de la clase *Cuenta* y su correspondiente excepción, se crea una cuenta *Cajero* donde se emulan depósitos y retiros a una cuenta:

```
public class Cajero {

    public static void main(String[] args) {
        Cuenta cuenta = new Cuenta();
        try {
            cuenta.depositar(2000);
            cuenta.retirar(1000);
            cuenta.getSaldo();
            cuenta.retirar(1000);
            cuenta.getSaldo();
            cuenta.retirar(1000);
            cuenta.getSaldo();
            cuenta.depositar(200);
            cuenta.retirar(100);
        } catch (SaldoInsuficienteException e) {
            e.printStackTrace();
        }

    }

}
```

Los métodos **getMessage** y **printStackTrace** se heredan de *Exception*. El método *getMessage* imprime la cadena que se envía al objeto cuando se construye. El método *printStackTrace* imprime la traza del error, es decir, el método donde se generó el problema y todos los métodos que se invocaron ante de éste.

Bibliografía

Barnes David, Kölling Michael
Programación Orientada a Objetos con Java.
Tercera Edición.
Madrid
Pearson Educación, 2007

Deitel Paul, Deitel Harvey.
Como programar en Java
Septima Edición.
México
Pearson Educación, 2008

Martín, Antonio
Programador Certificado Java 2.
Segunda Edición.
México
Alfaomega Grupo Editor, 2008

Dean John, Dean Raymond.
Introducción a la programación con Java
Primera Edición.
México
Mc Graw Hill, 2009