Material descargable

egg

Excepciones

¡Hola! 👋 Te damos la bienvenida a Excepciones.

Imagínate que estás manejando un coche y repentinamente el motor deja de funcionar. Es un problema grave, ¿no? Ahora, piensa en tu coche como si fuera tu programa. ¿Qué ocurre cuando hay un error durante su ejecución? Necesitamos una manera de manejar estos errores inesperados, como cuando el motor de un coche falla. Y la buena noticia es que Java tiene una solución para esto: las excepciones. Ya hemos visto las excepciones cuando aprendimos estructuras de control, pero ahora que sabemos mucho más podemos profundizar más sobre su funcionamiento.

¡Que comience el viaje! 🚀

Excepciones

En Java, una excepción es un evento que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de instrucciones del programa. Esto puede ser debido a un error en el código, a problemas con los datos de entrada, o a circunstancias imprevistas como una pérdida de conexión a la base de datos.

Palabras clave throw y throws

La primera vez que vimos las excepciones aprendimos que se capturen con el bloque try catch.

```
public class Application {
    public static void main(String[] args) {
        try {
            Integer numero = 10/0;
        } catch (Exception e) {
```

```
            System.out.println("No puedes dividir por 0");
        }
    }
}
```

¿Pero qué sucede si ahora queremos lanzar una excepción para que sea atrapada en otro lugar?, para eso podemos utilizar las palabras reservadas throws y throw:

- **throw** se utiliza para lanzar una excepción explícitamente en el código. Se usa en el cuerpo del método junto con el operador new.
- **throws** se utiliza para declarar que un método puede lanzar una excepción. Se usa en la firma del método.

```
public class Application {
    public static void main(String[] args) throws Exception {
        metodoA();
        metodoB();
        metodoC();
    }

    public static void metodoA() throws Exception {
        try {
            Integer numero = 10/0;
        } catch (Exception e) {
            throw new Exception("No puedes dividir por 0");
        }
    }

    public static void metodoB() throws Exception {
        try {
            Integer numero = 10/0;
        } catch (Exception e) {
            throw e;
        }
    }

    public static void metodoC() throws Exception {
```

```
        throw new Exception("No puedes dividir por 0");
    }

}
```

Como puedes ver en el código *throw* se utiliza tanto para lanzar una instancia que ya existe como también para lanzar una instancia nueva de Exception pasandole un mensaje que luego puede ser recuperado con el método *getMessage()* de la clase Exception. También no hace falta lanzar una excepción dentro del bloque catch, en metodoC() lanzamos directamente la excepción en el cuerpo del método.

La palabra throws se usa solo en la firma del método para señalar que lanza una excepción que debe ser atrapada por en el bloque que invoca el método que lanza la excepción, también se puede volver a delegar el manejo de la excepción volviendo a poner en la firma del método siguiente "throws Exception", como en el caso de nuestro ejemplo, donde en el método main en lugar de atrapar la excepciones que pueden lanzar los métodos metodoA(), metodoB() y metodoC() se vuelve a utilizar la cláusula throws.

Mira este video para entender mejor cómo funcionan estas dos palabras:

👉**Excepciones | Declaración throw - throws | JAVA | Egg**

Las excepciones y la herencia

En Java, todas las excepciones son subclases de la clase java.lang.Throwable. Esta última tiene dos subclases principales: Error y Exception.

Clase Throwable:

- Atributos:
    - detailMessage: Es el mensaje que describe porque se produjo el lanzamiento.
- Métodos principales:
    - getMessage(): Se utiliza para obtener eldetailMessage.
    - printStackTrace(): Este método imprime en la consola la pila de llamadas que llevó a la excepción. Es útil para depurar.

- getCause(): Retorna la causa de la excepción, el Throwable que causó esta excepción. Puede ser null si la causa no se conoce o no se inicializó.
- getStackTrace(): Retorna un array que contiene cada elemento en la pila de llamadas.

Clase Error

La clase Error es utilizada por la JVM para indicar errores serios que no deberían intentar ser manejados por la aplicación, a menudo son errores que suceden a nivel de la máquina virtual de Java y están más allá del control del código que estamos escribiendo. Algunos de los errores más comunes son:

- **OutOfMemoryError**: Se lanza cuando la JVM no puede asignar más memoria para un nuevo objeto debido a la falta de espacio de memoria. Ejemplo de cómo puede ocurrir:

```
int[] matriz = new int[Integer.MAX_VALUE];
```

- **StackOverflowError**: Se lanza cuando una aplicación recursiva consume todo el espacio de la pila de llamadas. Ejemplo de cómo puede ocurrir:

```
void funcionRecursiva() {
    funcionRecursiva();
}
```

Clase Exception

Esta es la clase que normalmente se utiliza para manejar errores en nuestro código. Las subclases de Exception pueden ser Excepciones Marcadas (Checked Exceptions) o no Marcadas (Unchecked Exceptions), y hablaré más sobre ellas en un momento.

Las excepciones marcadas

Son aquellas que debes manejar siempre en tu código. Son excepciones que se espera que probablemente sucedan, por eso son marcadas y el compilador te obliga a manejarlas de alguna manera. Por defecto, todas las Excepciones que heredan de Exception son marcadas. Algunas de ellas son:

- **IOException**: Se lanza durante operaciones de entrada y salida fallidas.

```
try {
    BufferedReader lector = new BufferedReader(new
FileReader("ruta/al/archivo"));
} catch (IOException e) {
    System.out.println("Ocurrió un error al leer el archivo");
}
```

- **ClassNotFoundException**: Se lanza cuando intentamos cargar una clase que no se encuentra.

```
try {
    Class.forName("ClaseNoExistente");
} catch (ClassNotFoundException e) {
    System.out.println("La clase no existe");
}
```

- **InterruptedException**: Se lanza cuando un hilo está esperando, durmiendo, o de otra manera ocupado, y el hilo es interrumpido por por otro hilo.
- **FileNotFoundException**: Se lanza cuando se intenta acceder a un archivo que no existe.

```
try {
    new FileInputStream("archivo_no_existente.txt");
} catch (FileNotFoundException e) {
    System.out.println("Archivo no encontrado");
}
```

- **SQLException**: Se lanza cuando ocurre un error en una operación de Base de Datos.

```
try {
    // Supongamos que tienes un objeto de conexión "conexion"
    conexion.executeQuery("SELECT * from tabla_no_existente");
} catch (SQLException e) {
    System.out.println("Error en la consulta SQL");
}
```

Las excepciones no marcadas

Son conocidas como **RuntimeExceptions**, son problemas que ocurren debido a errores en el código. Las RuntimeException son en su mayoría bugs en el programa, errores que son totalmente prevenibles si se escribe un buen código. Java no obliga a capturar estas excepciones ya que asume que como buenos programadores, escribiremos código que evite estas situaciones. Estas excepciones se heredan de la clase "*RuntimeException*". Algunas de ellas son:

- **NullPointerException**: Se lanza cuando se intenta acceder a un miembro de un objeto nulo.

```
String str = null;
try {
    System.out.println(str.length());
} catch (NullPointerException e) {
    System.out.println("El objeto es nulo");
}
```

- **ArrayIndexOutOfBoundsException**: Se lanza cuando se intenta acceder a un índice de un arreglo que no existe.

```
int[] arr = new int[5];
try {
    int numero = arr[10];
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("El índice está fuera de los límites del
arreglo");
}
```

- **ArithmeticException**: Se lanza cuando se intenta realizar una operación aritmética ilegal, como dividir por cero.

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

- **IllegalArgumentException**: Se lanza cuando se pasa un argumento ilegal o inapropiado a un método.

```
try {
    Thread.sleep(-1000);
} catch (IllegalArgumentException e) {
    System.out.println("Argumento ilegal");
}
```

- **NumberFormatException**: Se lanza cuando se intenta convertir un String a un tipo numérico, pero el formato del String no es apropiado para la conversión.

```
try {
    int num = Integer.parseInt("no_es_un_numero");
} catch (NumberFormatException e) {
    System.out.println("Formato de número inválido");
}
```

- **ClassCastException**: Se lanza cuando se intenta hacer un casting a un tipo de dato que no es el correcto.

```
Object x = new Integer(0);
try {
    System.out.println((String)x);
} catch (ClassCastException e) {
    System.out.println("No se puede convertir a String");
}
```

- **NegativeArraySizeException**: Se lanza cuando se intenta crear un array con un tamaño negativo.

```
try {
    int[] arr = new int[-1];
} catch (NegativeArraySizeException e) {
    System.out.println("El tamaño del arreglo no puede ser
negativo");
}
```

- **StringIndexOutOfBoundsException**: Se lanza cuando se intenta acceder a un índice de un String que no existe.

```
String str = "Hola";
try {
    char ch = str.charAt(10);
} catch (StringIndexOutOfBoundsException e) {
    System.out.println("Índice fuera de los límites del
String");
}
```

Bloque *finally* y *try with resources*

El bloque *finally* se utiliza para ejecutar código importante como la limpieza de recursos, no importa si una excepción fue lanzada o no. Siempre se ejecuta.

```
public class Main {
   public static void main(String[] args) {
        try {
            // Código que puede lanzar una excepción
        } catch (Exception e) {
            // Manejar la excepción
        } finally {
            // Limpiar recursos
        }
    }
}
```

Java 7 introdujo una nueva característica llamada try-with-resources que permite una mejor gestión de los recursos que se utilizan en un bloque try, asegurando que estos recursos se cierren al finalizar el proceso, incluso si se produce una excepción.

```
public class Main {
   public static void main(String[] args) {
        try (FileReader fr = new FileReader("file path")) {
            // Utilizar el recurso
        } catch (IOException e) {
```

```
            // Manejar la excepción
        }
    }
}
```

En este caso, el recurso FileReader se cerrará automáticamente después del bloque try, lo que lo hace más eficiente que el uso del bloque finally para limpiar los recursos.