UD12- EXCEPCIONES

Contenido

1 Manejo de excepciones

Las excepciones ayudan a afrontar cualquier situación inesperada o excepcional que se produce cuando se ejecuta un programa. Hemos visto que el bloque ***try-catch-finally*** permite el manejo de excepciones cuando éstas se producen en la ejecución de un programa:

- ***try***, en este bloque incluiremos el código cuya ejecución queremos controlar
- ***catch***, permite responder a un tipo de excepción. Puede haber más de un bloque catch cuando queremos controlar varios tipos de excepciones.
- ***finally***, para limpiar recursos tanto para la ejecución correcta del código como si se producen excepciones.

En el ejemplo realizamos la lectura de un fichero de texto y mostramos el contenido por la consola. Controlamos la situación de que el directorio no exista o que el fichero no exista, podría ocurrir que sucedieran otras excepciones como que no tuviéramos acceso al fichero que no estamos controlando. Ocurra o no una excepción siempre se ejecutará el bloque *finally*.

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
public class Main {
    public static void main(String[] args) {
        // Ruta del archivo a leer
        String path = "C:\\Test\\Prueba.txt";
        // Bloque try, tratamos de leer el fichero
        try {
            System.out.println("Archivo: " + path);
            // Creamos un objeto File y un BufferedReader para leer el archivo
            File file = new File(path);
            FileReader fileReader = new FileReader(file);
            BufferedReader bufferedReader = new BufferedReader(fileReader);
            // Leemos todo el contenido del archivo
            String content;
            StringBuilder fileContent = new StringBuilder();
            while ((content = bufferedReader.readLine()) != null) {
                fileContent.append(content).append("\n");
            }
            // Mostramos el contenido del archivo
            System.out.println(fileContent.toString());
            // Cerramos los recursos
            bufferedReader.close();
        }
        catch (FileNotFoundException ex0) {
            System.out.println("Archivo no encontrado: " + ex0.getMessage());
        }
        catch (IOException ex1) {
            System.out.println("Error de E/S: " + ex1.getMessage());
        }
        finally {
            System.out.println("Fin del bloque");
        }
    }
}
```

1. Jerarquía de Excepciones en Java

En Java, todas las excepciones heredan de la clase Throwable. Dentro de Throwable, existen dos tipos principales:

- Error: Usado para condiciones graves del sistema (no se recomienda atraparlas).
- Exception: Usado para excepciones que pueden ser manejadas por el programador.
    - RuntimeException: Subclase de Exception, que agrupa excepciones relacionadas con errores en tiempo de ejecución, como NullPointerException o ArrayIndexOutOfBoundsException.
    - IOException, FileNotFoundException, SQLException, etc., son subclases de Exception.

2. Manejo de Excepciones en Java con try, catch, y finally

- try: Contiene el código que puede generar una excepción.
- catch: Captura y maneja la excepción si ocurre dentro del bloque try.
- finally: Siempre se ejecuta, sin importar si ocurrió una excepción o no. Es útil para liberar recursos, como cerrar archivos o conexiones a bases de datos.

3. Propiedades de las Excepciones en Java

En Java, las excepciones también contienen propiedades similares a las que mencionas en C#.

Mensaje de Error (getMessage()):

En Java, la clase Throwable tiene el método getMessage() que devuelve una cadena descriptiva del error.

```
try {
   throw new IllegalArgumentException("Argumento inválido");
} catch (IllegalArgumentException e) {
   System.out.println("Error: " + e.getMessage());
}
```

Causa Interna (getCause()):

Java tiene un método getCause() que devuelve la excepción que causó la excepción actual.

```
try {
   try {
      throw new NullPointerException("Objeto nulo");
   } catch (NullPointerException e) {
      throw new IllegalArgumentException("Argumento incorrecto", e);
   }
} catch (IllegalArgumentException e) {
   System.out.println("Causa: " + e.getCause().getMessage());  // Mostrará "Objeto nulo"
}
```

Pila de Llamadas (printStackTrace()):

El método printStackTrace() imprime la pila de llamadas, es decir, el rastro de ejecución hasta el punto donde se produjo la excepción.

```
try {
   throw new ArithmeticException("División por cero");
} catch (ArithmeticException e) {
   e.printStackTrace();  // Muestra el rastro de la pila
}
```

4. Lanzar Excepciones en Java con throw

En Java, también puedes lanzar excepciones de forma explícita utilizando la palabra clave throw.

```
public void validarEdad(int edad) {
   if (edad < 18) {
      throw new IllegalArgumentException("La edad debe ser mayor de 18");
   }
}
```

En este caso, se lanza una excepción IllegalArgumentException si el parámetro edad es menor que 18.

5. Bloques try, catch y finally en Java:

try: El bloque donde se coloca el código susceptible de generar una excepción.

catch: Captura y maneja la excepción que se lanzó dentro del bloque try.

finally: Se ejecuta siempre, independientemente de si ocurrió o no una excepción. Se usa para limpiar recursos o liberar memoria.

Ejemplo completo en Java:

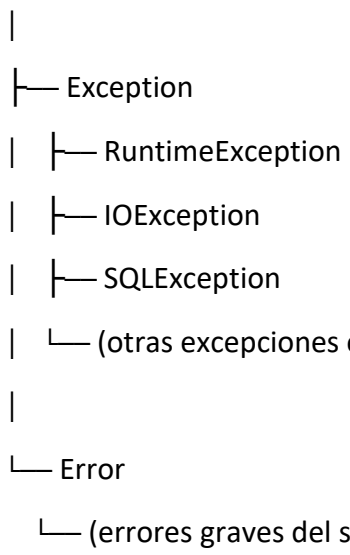
```
public class ExceptionHandlingExample {
    public static void main(String[] args) {
        // Usamos un bloque try-catch-finally
        try {
            // Lanzamos una excepción intencionadamente
            throw new IllegalArgumentException("Error: El argumento es inválido");
        } catch (IllegalArgumentException ex) {
            // Manejo de la excepción
            System.out.println("Mensaje de excepción: " + ex.getMessage());
            System.out.println("Causa de la excepción: " + ex.getCause());
            ex.printStackTrace();  // Muestra la pila de llamadas
        } finally {
            // Código que se ejecuta siempre, incluso si hubo excepción
            System.out.println("Bloque finally ejecutado");
        }
    }
}
```

2 Jerarquía de excepciones

las excepciones también siguen una jerarquía similar. La jerarquía de excepciones en Java tiene la clase Throwable como la superclase de todas las excepciones y errores. Desde ahí, se divide en dos grandes ramas: Exception (que es para excepciones que un programa puede manejar) y Error (para condiciones graves, normalmente no manejadas).

Jerarquía de Excepciones en Java

Throwable



Jerarquía y orden de los bloques catch

La jerarquía de excepciones establece que debemos capturar primero las excepciones más específicas y luego las más generales, porque, en caso de que una excepción sea capturada por un bloque catch más general, los bloques catch más específicos serán "ocultados", y no se ejecutarán.

Ejemplo de orden incorrecto en bloques catch:

```
try {
   // Código que puede generar una excepción
   throw new ArithmeticException("Error de cálculo");
} catch (Exception e) {
   System.out.println("Excepción capturada: " + e);
} catch (ArithmeticException e) {
   System.out.println("Excepción de cálculo capturada: " + e);
}
```

En este ejemplo, la excepción ArithmeticException es más específica que Exception. Sin embargo, debido al orden de los bloques catch, la excepción será capturada por el bloque catch de tipo Exception, y el bloque catch de tipo ArithmeticException nunca se ejecutará.

Ejemplo de orden correcto en bloques catch:

```
try {
   // Código que puede generar una excepción
   throw new ArithmeticException("Error de cálculo");
} catch (ArithmeticException e) {
   System.out.println("Excepción de cálculo capturada: " + e);
} catch (Exception e) {
   System.out.println("Excepción capturada: " + e);
}
```

Explicación:

- Orden de captura: En este caso, el bloque catch de ArithmeticException captura las excepciones más específicas primero. Si no se encuentra una coincidencia, el flujo pasa al bloque catch que captura excepciones más generales como Exception.
- Evitar capturas generales primero: Siempre es mejor colocar primero las excepciones más específicas, porque si pones Exception antes de ArithmeticException, capturará todas las excepciones antes de que el bloque catch más específico tenga oportunidad de ejecutarse.

Bloques catch múltiples:

Puedes usar múltiples bloques catch para manejar diferentes tipos de excepciones. Aquí te dejo un ejemplo con varios tipos de excepciones:

```
try {
   int result = 10 / 0;
} catch (ArithmeticException e) {
   System.out.println("Excepción de cálculo: " + e);
} catch (NullPointerException e) {
   System.out.println("Excepción de puntero nulo: " + e);
} catch (Exception e) {
   System.out.println("Excepción general: " + e);
}
```

En este caso:

- Si ocurre una excepción de tipo ArithmeticException (como la división por cero), el primer bloque catch la captura.
- Si ocurre una NullPointerException, la segunda captura la excepción.
- Finalmente, si ocurre cualquier otra excepción (que no sea ArithmeticException o NullPointerException), el bloque catch más general captura esa excepción.

3 Excepciones creadas por el usuario

Crear excepciones personalizadas es una práctica común cuando se desea lanzar errores específicos en una aplicación que no se manejan adecuadamente con las excepciones estándar proporcionadas por el sistema.

En Java, las excepciones personalizadas se crean mediante la creación de una nueva clase que hereda de Exception (o de alguna de sus subclases).

Pasos para crear una excepción personalizada:

1. Heredar de Exception: Crea una clase que extienda Exception o RuntimeException (dependiendo de si deseas que la excepción sea comprobada o no).
2. Definir constructores: Al menos, se deberían definir tres constructores:
   - Un constructor predeterminado.
   - Un constructor que reciba un mensaje de error.
   - Un constructor que reciba tanto un mensaje como una excepción interna (para encapsular excepciones originales).

```
// Definimos una excepción personalizada

public class MiExcepcion extends Exception {

   public MiExcepcion() {// Constructor predeterminado

      super("Se ha producido un error");

   }

   public MiExcepcion(String mensaje) {// Constructor que establece el mensaje de la excepción

      super(mensaje);    }

   // Constructor que establece el mensaje y la excepción interna

   public MiExcepcion(String mensaje, Throwable causa) {

      super(mensaje, causa);

   }

}
```

4 Buenas prácticas

Manejar correctamente las excepciones y seguir estos lineamientos asegurará que tu código sea más confiable y fácil de mantener. A continuación, te proporciono un desglose de las mejores prácticas que mencionas y cómo puedes implementarlas en tu código:

1. Verificación temprana (Early validation)

Detectar problemas lo antes posible evita que los errores se propaguen y causen fallos mayores. Esto incluye la validación de los datos de entrada y el manejo de excepciones de manera proactiva.

- Ejemplo en Java: Verificación de parámetros de entrada de un método.

```
public void procesarDatos(String archivo) {
   if (archivo == null || archivo.isEmpty()) {
      throw new IllegalArgumentException("El archivo no puede ser nulo o vacío");
   }
   // Continuar con el procesamiento
}
```

En este ejemplo, estamos validando de manera temprana si los parámetros del método son válidos antes de proceder con el procesamiento de datos.

2. No confiar en datos externos

Los datos que provienen de fuentes externas (como archivos, bases de datos o usuarios) pueden ser corruptos o mal formateados. Siempre hay que verificar y validar estos datos antes de usarlos.

- Ejemplo en Java: Validación de datos leídos desde un archivo.

```
public void leerArchivo(String ruta) {
   try {
      File archivo = new File(ruta);
      if (!archivo.exists()) {
         throw new FileNotFoundException("El archivo no existe: " + ruta);
      }
```

```
        // Leer datos del archivo...


    } catch (IOException e) {
        System.out.println("Error al leer el archivo: " + e.getMessage());
    }
}
```

Aquí estamos verificando que el archivo exista antes de intentar leerlo, lo que previene errores en tiempo de ejecución si el archivo no está disponible.

3. Las excepciones no deben usarse para cambiar el flujo del programa

Las excepciones deben ser utilizadas para manejar errores inesperados y no para controlar el flujo normal de la aplicación. Usar excepciones para flujos normales puede hacer que el código sea menos eficiente y más difícil de entender.

- Mal uso de excepciones (No recomendado):

```
try {
    int resultado = (int) obj;  // Lanzará una excepción si obj no es entero
} catch (ClassCastException e) {
    // Código para manejar la excepción, pero es un mal diseño usar excepciones para el control de flujo
}
```

En lugar de esto, es mejor usar otras técnicas de control de flujo como condicionales (if, switch, etc.) en lugar de usar excepciones para condiciones previsibles.

4. No devolver excepciones como valores de retorno

Es importante que las excepciones sean usadas para controlar errores, no como valores de retorno. Si una función lanza una excepción, debe ser tratada adecuadamente.

- Ejemplo adecuado:

```
public void dividir(int a, int b) throws ArithmeticException {
   if (b == 0) {
      throw new ArithmeticException("No se puede dividir por cero");
   }
   System.out.println(a / b);
}
```

En este caso, en vez de devolver una excepción como valor, la función lanza una excepción para indicar que ocurrió un error.

5. No generar excepciones genéricas (System.Exception, System.NullReferenceException, etc.)

Es preferible generar excepciones específicas que describan claramente el tipo de error que ocurrió.

- Excepción específica (recomendado):

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

Utilizar excepciones personalizadas o más específicas ayudará a entender mejor el error y a gestionarlo de manera más eficiente.

6. No "comerse" excepciones

Capturar excepciones de manera general con un bloque catch(Exception) vacío es una mala práctica. Esto oculta los errores y hace que el programa falle sin dar detalles.

- Evitar este patrón:

```
try {
   // Código que puede lanzar excepciones
} catch (Exception e) {
   // No hacer nada y comer la excepción es una mala práctica
}
```

- Correcto manejo de excepciones:

```
try {
   // Código que puede lanzar excepciones
} catch (IOException e) {
   System.out.println("Error al acceder al archivo: " + e.getMessage());
} catch (Exception e) {
   System.out.println("Ocurrió un error inesperado: " + e.getMessage());
}
```

En este ejemplo, capturamos las excepciones de forma específica y mostramos un mensaje útil que permitirá al desarrollador o al usuario identificar el problema.

7. Uso de finally para la limpieza y liberación de recursos

Siempre es recomendable usar un bloque finally para asegurarse de que los recursos se liberen correctamente, incluso si ocurre una excepción.

- Ejemplo en Java: Liberación de recursos como archivos o conexiones de base de datos.

```
public void leerArchivo(String ruta) {
   BufferedReader reader = null;
   try {
      reader = new BufferedReader(new FileReader(ruta));
      String line;
      while ((line = reader.readLine()) != null) {
         System.out.println(line);
      }
```

```
    } catch (IOException e) {
        System.out.println("Error al leer el archivo: " + e.getMessage());
    } finally {
        try {
            if (reader != null) {
                reader.close();
            }
        } catch (IOException e) {
            System.out.println("Error al cerrar el archivo: " + e.getMessage());
        }
    }
}
```

Aquí, el bloque finally asegura que el archivo se cierre correctamente, incluso si ocurre una excepción en el bloque try.

8. Uso de try-with-resources o using

El uso de try-with-resources en Java simplifica el manejo de recursos, asegurando que se liberen correctamente.

- Ejemplo en Java con try-with-resources:

```
public void leerArchivo(String ruta) {
    try (BufferedReader reader = new BufferedReader(new FileReader(ruta))) {
        String line;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
    } catch (IOException e) {
        System.out.println("Error al leer el archivo: " + e.getMessage());
}}
```

El bloque try-with-resources se asegura de cerrar automáticamente el BufferedReader cuando ya no sea necesario, sin necesidad de escribir código adicional de limpieza.