Portfolio: Control y Manejo de Excepciones en Java

Jhonal Roca

1º DAW - 24/25

Concepto de Excepciones

Las excepciones son eventos inesperados que pueden interrumpir el flujo de ejecución de un programa. En otras palabras, son esos errores que hacen que tu programa se caiga si no los manejas correctamente.

Por ejemplo, imagina que intentas acceder a una variable que no ha sido inicializada:

```
2
3   List lista = null;
4   lista.add("A");  // Genera NullPointerException
5   System.out.println(lista);
6
```

Esto provocará una **`NullPointerException`**, porque la lista es **`null`** y no puedes agregarle elementos.

Jerarquía de Excepciones en Java

En Java, las excepciones tienen diferentes niveles de gravedad:

- **Error:** Problemas graves del sistema, como falta de memoria (**`OutOfMemoryError`**). No se suelen manejar.
- **Exception:** Errores recuperables que podemos gestionar.
    - **Checked Exceptions:** Errores que dependen del uso del programa y que Java nos obliga a manejar (**`IOException`**).
    - **Unchecked Exceptions:** Errores de programación, como dividir entre cero, que Java lanza automáticamente (**`NullPointerException`**).

Miembros de una Excepción

Las excepciones en Java tienen métodos que nos ayudan a entender qué pasó:

- **`getMessage()`:** Muestra el mensaje del error.
- **`printStackTrace()`:** Imprime la traza del error.
- **`toString()`:** Nos da una descripción rápida de la excepción.

Ejemplo de cómo capturarlas:

```
try {
    FileReader fichero = new FileReader("archivo.txt");
} catch (FileNotFoundException e) {
    System.out.println("Error: " + e.getMessage());
    e.printStackTrace();
}
```

Cómo manejar excepciones

Si nuestro código falla, podemos capturar la excepción y evitar que el programa se cierre inesperadamente.

Captura de Excepciones (`try-catch-finally`)

```
try {
    int dividendo = 5;
    int divisor = 0;
    System.out.println("Resultado: " + dividendo / divisor);
} catch (ArithmeticException ex) {
    System.out.println("Imposible dividir por 0");
} finally {
    System.out.println("Fin del programa");
}
```

Propagación de Excepciones (`throws`)

Si no queremos manejar la excepción en el mismo método, podemos pasarla a quien nos llamó:

```
27  public static int division(int dividendo, int divisor) throws ArithmeticException {
28      return dividendo / divisor;
29  }
```

Lanzamiento de Excepciones (`throw`)

Si queremos forzar un error en ciertas condiciones, usamos `throw`:

```
31
32  if (x == null) {
33      throw new IllegalArgumentException("No se admiten valores nulos");
34  }
35
```

Buenas Prácticas

Para que nuestro código sea sólido y fácil de mantener, es importante seguir algunas reglas:

- Captura sólo las excepciones necesarias y no las ignores.
- Usa **`finally`** o **`try-with-resources`** para liberar recursos.
- Evita usar excepciones genéricas, ya que no aportan suficiente información.
- Prioriza un código bien estructurado para reducir el uso de excepciones.

Aserciones en Java

Las aserciones son útiles para comprobar condiciones mientras desarrollamos, pero no se deben usar en producción.

```
36  int x = -15;
37  assert x > 0 : "El valor debe ser positivo";
38  System.out.println("Valor positivo: " + x);
39
```

Si ejecutamos el programa con **`-ea`**, nos dará un **`AssertionError`**, lo que ayuda a detectar errores temprano.

Ejercicios Aplicados

Para poner en práctica lo aprendido, puedes hacer ejercicios como:

- **Crear una excepción personalizada** llamada **`NumeroRepetido`** para controlar valores duplicados en una lista.
- **Modificar un programa de división** para capturar la excepción de dividir por cero.
- **Mejorar la calculadora** para evitar errores al introducir valores no numéricos y manejar las excepciones correctamente.

Comparación de Tipos de Excepciones en Java

Tipo de dato	Uso en excepciones	Ventajas	Desventajas
Exception	Errores recuperables(IOException)	Obligatoria de manejar	Puede hacer el código más complejo
RuntimeException	Errores de programación (NullPointerException)	No requiere throws	Puede provocar fallos inesperados
ArithmeticException	Dividir por cero	Evita cálculos inválidos	Necesita control para evitar cierre del programa
IllegalArgumentException	Parametro invalido en un método	Facilita validación previas	Puede cerrar el programa si no se maneja
Excepciones personalizadas	numeroRepetido, EceptionEdadInvalida	Mejor identificación de errores	Puede generar código innecesario.