UD5

LENGUAJE DE PROGRAMACIÓN JAVA

5.2 Excepciones personalizas

MP_0485
Programación

Introducción

Cuando se produce una condición de error (excepción para Java), la máquina virtual genera un tipo de objeto especial con información sobre el suceso ocurrido; un objeto de tipo Exception.

La jerarquía de clases para la gestión de excepciones en Java se puede resumir con la siguiente imagen.

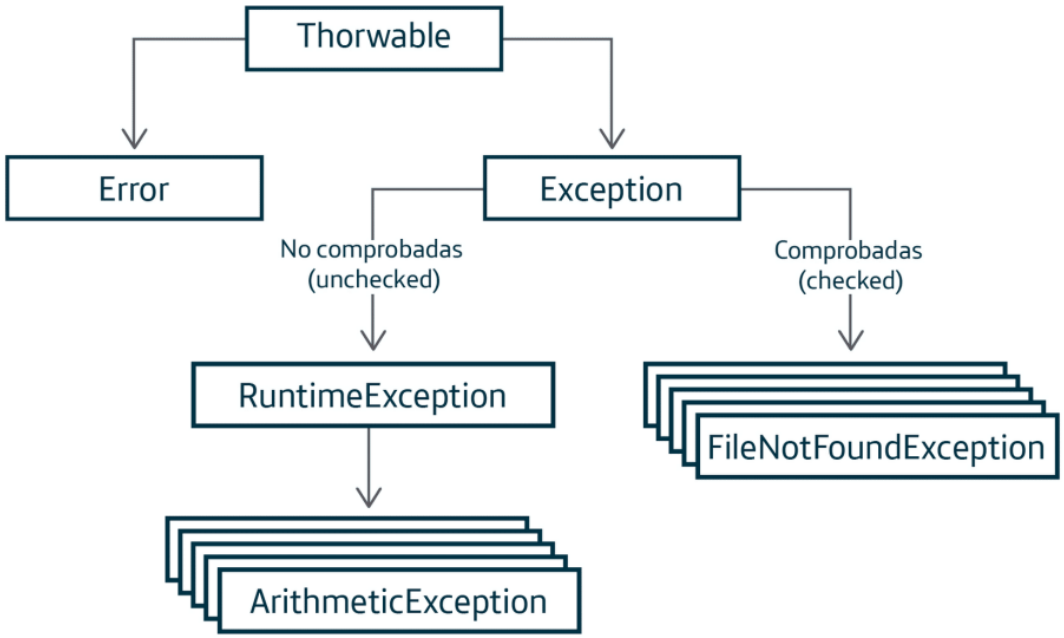

Figura 1: Estructura jerárquica de las clases de Excepción en Java.

Podemos crear nuestras propias excepciones personalizadas. Una clase personalizada de excepción debe heredar de `Exception` o de `RuntimeException`. Lo más habitual es heredar de `Exception`.

Si creamos una clase que hereda de `Exception` será una excepción comprobada, es decir, es obligatorio gestionarla usando try ... catch o propagarla como hemos visto en la unidad anterior. Si creamos una clase que hereda de `RuntimeException` será una excepción de tipo "no comprobada", es decir, no es obligatorio gestionarla.

Por convención, es altamente conveniente que las clases de excepción personalizadas terminen con la palabra Exception.

Crear excepción no comprobada

Vamos a estudiar la creación de excepciones no comprobadas, para ellos veremos por medio de un ejercicio guiado cómo se crea una excepción personalizada de tipo "no comprobada".

Una clase Coche compuesta por las propiedades marca, modelo y velocidad tiene los métodos `acelerar()` y `frenar()`. Queremos considerar una situación excepcional el hecho de que la velocidad exceda de los 120 km/hora. Para ello crearemos una excepción llamada `ExcesoVelocidadException` de tipo "no comprobada".
Partiremos de esta sencilla clase Coche que todavía no considera el exceso de velocidad como una situación excepcional:

```
public class Coche {
	private String marca;
	private String modelo;
	private int velocidad;

	public Coche(String marca, String modelo) {
		this.marca = marca;
		this.modelo = modelo;
		this.velocidad = 0;
	}

	@Override public String toString() {
		return "Coche [marca=" + marca + ", modelo=" + modelo + ",
		velocidad="+velocidad];
	}

	public void acelerar(int cuanto) {
		this.velocidad = this.velocidad + cuanto;
	}
	public void frenar(int cuanto) {
		this.velocidad = this.velocidad - cuanto;
	}
}
```

Ahora creamos la clase Principal, con método main y comprobar que evidentemente puedes acelerar el coche todo lo que quieras.

```
public class Principal {
	public static void main(String[] args) {
		Coche miCoche = new Coche("Ford", "Fiesta");
		miCoche.acelerar(100);
		miCoche.acelerar(50);
		System.out.println(miCoche);
	}
}
```

Para solventar esta situación y considerar el exceso de velocidad una situación excepcional vamos a seguir los siguientes pasos:

Crear una clase que extienda de Exception o RuntimeException. Como queremos que nuestra excepción sea de tipo "no comprobada", extenderemos de RuntimeException. Nuestra nueva clase de excepción podría quedar así:

```
public class ExcesoVelocidadException extends RuntimeException {
	private static final long serialVersionUID = 1L;
	private int nuevaVelocidad;

public ExcesoVelocidadException(int nuevaVelocidad) {
	super("Exceso de velocidad");
	this.nuevaVelocidad = nuevaVelocidad;
}

@Override public String toString() {
	return "ExcesoVelocidadException [nuevaVelocidad=" +
nuevaVelocidad]”;
}
}
```

En primer lugar, observa que nuestra nueva clase `ExcesoVelocidadException` termina con la palabra Exception. Esto es una buena práctica, y aunque sea opcional puedes considerarlo como

obligatorio, ya que es la forma de que se distinga claramente que se trata de una clase que representa una excepción.

La clase ExcesoVelocidadException extiende de `RuntimeException`, lo que la convierte en una excepción de tipo "no comprobada". En el constructor, con la sentencia `super("Exceso de velocidad")` estamos pasando un valor al constructor de `RuntimeException`. El texto pasado como argumento será lo que finalmente devuelva el método `getMessage()` del objeto de excepción.

Nuestra clase de excepción, como cualquier otro tipo de clase, puede tener todas las propiedades y todos los métodos que sea necesario para cumplir correctamente con su función. En el ejemplo, hemos añadido la propiedad `nuevaVelocidad`, que contendrá la velocidad del coche después de la aceleración que provocó la excepción. El valor para la propiedad nuevaVelocidad se lo estamos pasando al constructor por medio de un parámetro.

La sentencia `private static final long serialVersionUID = 1L;` es necesaria para evitar un warning o alerta que nos aparece en el IDE. Esto se debe a que `RuntimeException` es una clase serializable y necesita un número de versión. Este concepto lo veremos en otra unidad, por ahora confórmate con saber que hay que añadir esa línea y ya está.

Después queremos desencadenar la excepción dentro de la clase Coche. La clase tendrá que desencadenar la excepción cuando la velocidad sea superior a 120 km/hora y lo hará creando un nuevo objeto de la clase `ExcesoVelocidadException` con un tipo de declaración específica para este fin que tiene el siguiente formato:

```
throw new ClaseDeExcepción(parámetros);
```

La excepción se desencadenará dentro del método acelerar. Modi ca el código del método acelerar de la siguiente manera:

```
public void acelerar(int cuanto) {
        this.velocidad = this.velocidad + cuanto;
        if (this.velocidad > 120) {
                throw new ExcesoVelocidadException(this.velocidad);
        }
}
```

Finalmente, debemos de provocar la excepción. Puesto que es una excepción de tipo "no comprobada" no estamos obligados a gestionarla, el IDEA no nos está dando ninguna pista sobre la posible excepción.

```
public class Principal {
	public static void main(String[] args) {
		Coche miCoche = new Coche("Ford", "Fiesta");
		System.out.println(miCoche.toString());
		miCoche.acelerar(100);
		miCoche.acelerar(50);
		System.out.println(miCoche);
	}
}
```

Si ejecutas el programa, como no hemos gestionado la excepción, se abortará justo en la segunda aceleración y el programa terminará lanzando la traza de la excepción, tal y como puedes a continuación.

```
Coche [marca=Ford, modelo=Fiesta, velocidad=0]
Exception in thread "main" ExcesoVelocidadException [nuevaVelocidad=150]
	at Coche.acelerar(Coche.java:20)
	at Principal.main(Principal.java:6)
```

Llamamos traza de la excepción a un mensaje de texto que muestra el nombre de la excepción, los argumentos si los tiene, el método que provocó la excepción y la pila de llamadas a los distintos métodos de donde proviene. La parte de texto en rojo del cuadro de información anterior es la traza del error.

También, por supuesto, podemos optar por gestionar la excepción usando try ... catch. En ocasiones resulta interesante mostrar la traza del error dentro del bloque catch para obtener mayor información sobre lo ocurrido. Esto se hace con la `sentencia e.printStackTrace()`.

```
public class Principal {
	public static void main(String[] args) {
		Coche miCoche = new Coche("Ford", "Fiesta");
		System.out.println(miCoche.toString());
		try {
			miCoche.acelerar(100);
			miCoche.acelerar(50);
		}
		catch (ExcesoVelocidadException e) {
			e.printStackTrace();
		}
		System.out.println(miCoche);
	}
}
```

Crear excepción comprobada

En esta ocasión realizaremos el mismo ejemplo de antes, pero esta vez la excepción ExcesoVelocidadException será de tipo "comprobada".

```
	public class ExcesoVelocidadException extends Exception {
		private static final long serialVersionUID = 1L;
		private int nuevaVelocidad;

	public ExcesoVelocidadException(int nuevaVelocidad) {
		super("Exceso de velocidad");
		this.nuevaVelocidad = nuevaVelocidad;
	}

	@Override public String toString() {
		return "ExcesoVelocidadException [nuevaVelocidad=" +
	nuevaVelocidad]”;
	}
}
```

Solo hemos modificado el nombre de la clase base `RuntimeException` por `Exception`, el resto del código se ha quedado igual.

Para desencadenar la excepción dentro de la clase Coche el método acelerar no solo tiene que incluir la declaración `throw new` para desencadenar la excepción, sino que además debe propagarla hacia arriba con la declaración `throws ExcesoVelocidadException` en la cabecera de la función para que sea recogida por el método que la provocó. En nuestro ejemplo, la excepción será provocada en el método main de la clase Principal.

```
public void acelerar(int cuanto) throws ExcesoVelocidadException{
        this.velocidad = this.velocidad + cuanto;
        if (this.velocidad > 120) {
                throw new ExcesoVelocidadException(this.velocidad);
        }
}
```

A continuación, para provocar la excepción comenzaremos por crear una clase Main, una instancia de Coche y hacer uso del método acelerar sin preocuparnos de nada más.

```
public class Principal {
        public static void main(String[] args) {
                Coche miCoche = new Coche("Ford", "Fiesta");
                System.out.println(miCoche.toString());
                miCoche.acelerar(100);
                miCoche.acelerar(50);
                System.out.println(miCoche);
        }
}
```

Pero en este caso el IDE nos muestra errores de compilación porque ahora tenemos una excepción de tipo "controlado" y estamos obligados a gestionarla. Por lo que, debemos encerrar el código en un bloque *try ... catch* o propagar la excepción. En esta ocasión vamos a utilizar un bloque try ... catch.

```
public class Principal {
	public static void main(String[] args) {
		Coche miCoche = new Coche("Ford", "Fiesta");
		System.out.println(miCoche.toString());
		try {
			miCoche.acelerar(100);
			miCoche.acelerar(50);
		}
		catch (ExcesoVelocidadException e) {
			System.out.println(e.getMessage());
			System.out.println(e.toString());
		}
		System.out.println(miCoche);
	}
}
```

Throw frente throws

Es importante que no confundas las declaraciones tipo `throws` con las declaraciones tipo `throw new`.

Las declaraciones `throws` se utilizan para relanzar o propagar hacia arriba una excepción.

```
	public static void ejecutarTarea2() throws FileNotFoundException {
	FileReader fichero = new FileReader("c:/datos.txt");
	System.out.println("El fichero ha sido abierto");
```

Las declaraciones tipo `throw` new se utilizan para desencadenar una excepción.

```
	public void acelerar(int cuanto) {
		this.velocidad = this.velocidad + cuanto;
		if (this.velocidad > 120) {
			throw new ExcesoVelocidadException(this.velocidad);
		}
	}
```

Try with resources

Una de las características introducidas en Java 7 es la sentencia **try-with-resources**, diseñada para cerrar automáticamente los recursos dentro de la estructura **try-catch-finally**, simplificando así el código. Las variables cuyas clases implementan la interfaz `AutoCloseable` pueden declararse en el bloque de inicialización del **try-with-resources**, y sus métodos `close()` se ejecutarán al finalizar el bloque, como si se hubiera escrito explícitamente.

Un ejemplo de código que lee una línea de un archivo utilizando **try-with-resources** en Java 7 se muestra a continuación. Como se puede ver, no es necesario invocar manualmente el método close() para liberar los recursos de la instancia de **BufferedReader**.

```
public static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

Anteriormente a Java 7 esto se debía hacer de la siguiente manera con unas pocas lineas más de código algo menos legibles.

```
public static String readFirstLineFromFileWithFinallyBlock(String path) throws
IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

En Java 6, cuando trabajas con recursos (como archivos o conexiones), normalmente usas un bloque **try-catch-finally** para asegurarte de cerrar esos recursos. El código que se usa es parecido al de Java 7, pero no exactamente igual.

¿Por qué? Porque en Java 6 **tienes que declarar la variable del recurso (por ejemplo, BufferedReader br) fuera del bloque try**, para poder usarla tanto en el try como en el finally donde la cierras manualmente.

Además, hay un detalle importante sobre las excepciones:

- Si ocurre un error en el bloque **try** y luego otro error en el bloque **finally**, la excepción del finally **oculta** la del try. Es decir, la excepción original se pierde y solo se lanza la del finally. Esto puede complicar la depuración.

En cambio, con **try-with-resources** (Java 7 en adelante), esto se soluciona porque el cierre del recurso se maneja automáticamente y las excepciones se registran sin enmascararse.

La mayoría de las clases relacionadas con operaciones de entrada y salida implementan la interfaz **AutoCloseable**. Esto incluye aquellas que trabajan con el sistema de archivos y flujos de red, como **InputStream**, así como las utilizadas para la conexión a bases de datos mediante **JDBC**, como **Connection**.

Referencias

Apuntes elaborados a partir de la siguiente documentación:

[1] Apuntes Fernando Barber y Ricardo Ferris. Universidad de Valencia.

[2] Apuntes Programación de José Antonio Díaz-Alejo. IES Camp de Morvedre.

[3] Apuntes de Programación de Carlos Cacho y Raquel Torres. Ceedcv.

[4] Apuntes de Programación Edix Digital Workers.