Revisitando excepciones: crear excepciones propias

Resumen

Hasta ahora hemos visto el tema “excepciones”. Vimos qué son y para qué sirven. Analizamos también los dos tipos de excepciones posibles: las *checked* y las *unchecked exceptions*. Asimismo, vimos cuál es la estructura de código necesaria para lidiar con excepciones y cómo debemos avisar a los usuarios (código cliente) de nuestros métodos cuando estos lanzan excepciones chequeadas.

A modo de repaso, podemos mostrarlo en código:

```
public void readSomeText(String path) throws
FileNotFoundException {
  FileReader fr = null;
  try {
    File f = new File(…);
    FileReader fr = new FileReader(f);
    …
  } catch (FileNotFoundException e) {
    System.out.println(“No file found at ” + path);
    throw e;
  } finally {
    fr.close(); // free resources
  }
}
```

Excepciones propias

Como vimos, podemos manejar excepciones chequeadas y también no chequeadas. La diferencia es que estamos obligados a manejar las chequeadas. En uno u otro caso, sería interesante señalizar errores que nosotros consideremos “fuera del camino feliz”. Asimismo, sería interesante no estar limitados a las excepciones que nos brinda Java. Ahora

que estamos familiarizados con el concepto de “herencia”, podemos decir que estamos en condiciones de crear nuestras propias excepciones. Utilizar herencia nos da una relación **es-un**. Entonces, para crear una clase que es-una excepción, nos queda hacer algo así:

```
public class MyCustomCheckedException extends Exception {
//
}
```

O si queremos una excepción no chequeada:

```
public class MyCustomUncheckedException extends
RuntimeException {
//
}
```

A partir de este momento, podemos crear nuestras propias excepciones chequeadas y no chequeadas a voluntad.

Es importante recalcar que una excepción es una clase como cualquier otra. Por tanto, cuando vayamos a crear nuestras propias excepciones, podemos incluir atributos y métodos que nos sean de utilidad para trabajar con esa excepción. Por ejemplo, podríamos crear una excepción chequeada con un código para indicar la naturaleza del problema:

```
public class CustomCodeException extends Exception {
  private int code;
  public CustomCodeException(int code) {
    this.code = code;
  }
  public int getCode() {
    return this.code;
  }
}
```

Entonces, al momento de utilizarla podríamos hacer lo siguiente:

```
public void readSomeText(String path) throws
CustomCodeException {
```

```
  FileReader fr = null;
  try {
    File f = new File(path);
    fr = new FileReader(f);
    …
  } catch (FileNotFoundException e) {
    throw new CustomCodeException(1001);
  } finally {
    try {
      fr.close(); // free resources | also throws exception
    } catch (IOException e) { /* nothing */ }
  }
}
```

Similarmente, todas las excepciones, al extender exception, heredan la posibilidad de contener dos cosas fundamentales:

- **Un mensaje que podemos utilizar para transmitir información adicional**. De hecho, si observan detenidamente, cada vez que se lanza una excepción se muestra un mensaje. Por ejemplo, cuando vimos el casteo en el código de los perros, si no teníamos cuidado podríamos obtener:

  ```
  Exception in thread "main" java.lang.ClassCastException: Perro cannot be cast to Rottweiler
        at Test.main(Test.java:15)
  ```

  "Perro cannot be cast to Rottweiler" es el mensaje que se adosa a la excepción cuando se lanza. Siempre que recibamos una excepción podemos utilizar el método "getMessage()" para accederlo, si es que al momento de lanzarla elegimos incluir un mensaje.

- **La causa de esta excepción**. La causa de una excepción típicamente es otra excepción. Estrictamente, la causa es un Throwable. Utilizar la causa de una excepción puede ayudar, al momento de lanzarla, a identificar más fácilmente el problema, guiando al programador en la pila de llamadas y dando una idea más clara de dónde se originó el problema. Supongamos que en el código anterior queremos

leer un archivo en /home/user/data.txt y no se encontró el archivo. Entonces, obtendríamos algo del estilo de:ç

```
CustomCodeException
        at Test.readSomeText(Test.java:39)
        at Test.main(Test.java:27)
Caused by: java.io.FileNotFoundException: /home/user/data.txt (No such file or directory)
        at java.io.FileInputStream.open0(Native Method)
        at java.io.FileInputStream.open(FileInputStream.java:195)
        at java.io.FileInputStream.<init>(FileInputStream.java:138)
        at java.io.FileReader.<init>(FileReader.java:72)
        at Test.readSomeText(Test.java:37)
        ... 1 more
```

Ese “caused by” nos ayudará a encontrar el problema mas fácilmente. Se lanzó una CustomCodeException *porque* se lanzó una FileNotFoundException.

Código

Hay algunas cosas para señalar antes de seguir. Las excepciones son clases y, como tales, tienen atributos y métodos. También definen constructores. Y si hay constructores, aplican las restricciones que ya conocemos. Es decir, en el caso de nuestra CustomCodeException, si quisiéramos crear un constructor al que le pasemos el código, entonces perdemos el constructor por *default*. Esto es algo a tener en cuenta si luego planeamos tener subclases de CustomCodeException.

Asimismo, si queremos construir una CustomCodeException con un mensaje, o con una causa, o con las dos cosas, debemos definir los constructores necesarios. Típicamente, y con fines puramente prácticos, cada vez que creamos una excepción, agregamos cuatro constructores típicos:

1. Constructor vacío.
2. Constructor con mensaje.
3. Constructor con causa.
4. Constructor con mensaje y causa.

Si tuviéramos, como en nuestro caso, algún atributo adicional, agregaríamos los constructores que necesitemos para tomarlo en cuenta.

Además, si tenemos atributos privados, tendremos *getters* y *setters* públicos. Sin embargo, en una excepción, no es del todo correcto el *setter*. La idea central es que, una vez que se lanzó la excepción, cualquier información que se recolecte en la pila de llamadas al momento de lanzarla no se pueda cambiar. No sería bueno arriesgarse a que algún usuario del código (voluntaria o involuntariamente) pueda cambiar un atributo (mensaje, valor, etc.) de una excepción en su burbujeo por las llamadas.

En conclusión, nuestra CustomCodeExcepcion quedaría más o menos así:

```
public class CustomCodeException extends Throwable {

    private int code;

    public CustomCodeException() { //1
    }

    public CustomCodeException(String s) { //2
        super(s);
    }


    public CustomCodeException(Throwable throwable) { //3
        super(throwable);
    }


    public CustomCodeException(String s, Throwable throwable) { //4
        super(s, throwable);
        this.code = code;
    }
    public CustomCodeException(int code) {
        this.code = code;
    }

    public CustomCodeException(int code, String s, Throwable throwable) {
        super(s, throwable);
        this.code = code;
    }


    public int getCode() {
        return code;
```

```
    }
}
```

Al momento de usarla, por ejemplo, en un main haríamos:

```
public class Test {
  public void readSomeText(String path) throws CustomCodeException {
    FileReader fr = null;
    try {
        File f = new File(path);
        fr = new FileReader(f);
    } catch (FileNotFoundException e) {
        //constructor con código, mensaje Y causa
        throw new CustomCodeException(1001, "No file at " + path, e);
    } finally {
        try {
            if(fr != null) {
                fr.close(); // free resources
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
  }

  public static void main (String[] args) {
    try                                                                    {
      Test t = new Test();
      t.readSomeText("/home/user/data.txt");
    }          catch          (CustomCodeException          e)          {
      System.out.println("CODE:"+e.getCode());
      e.printStackTrace();
    }


  }
}
```

Y al correrlo obtendríamos algo así:

```
CODE:1001
CustomCodeException: No file at /home/user/data.txt
      at Test.readSomeText(Test.java:40)
      at Test.main(Test.java:27)
Caused by: java.io.FileNotFoundException: /home/user/data.txt (No such file or directory)
      at java.io.FileInputStream.open0(Native Method)
      at java.io.FileInputStream.open(FileInputStream.java:195)
      at java.io.FileInputStream.<init>(FileInputStream.java:138)
      at java.io.FileReader.<init>(FileReader.java:72)
      at Test.readSomeText(Test.java:38)
      ... 1 more
```

Aquí, pudimos (según el orden en que aparecen, en negrita):

1. Imprimir el "código" que dimos al momento de lanzar la excepción.
2. Con el printStackTrace() pudimos ver el mensaje que incluimos al lanzar la excepción ("no file at…").
3. También pudimos ver la causa de esta CustomCodeException: una FileNotFoundException (que a su vez muestra su propio mensaje).

Utilidad de crear excepciones propias

La función fundamental de crear nuestras propias excepciones es **desacoplar**. De hecho, si observan detenidamente, el main de nuestra clase Test pide leer un archivo.

Sin embargo, la realidad es que el main, quien es usuario del código de readSomeText, nunca podrá enterarse si el archivo se lee contra el disco local de la PC o contra una ubicación en un almacenamiento en la nube. Es decir, el usuario del código pidió leer un archivo, pero quedó desacoplado de la solución de almacenamiento del archivo. Nuestro readSomeText puede ser reutilizado, dado que no está atado a un sistema de archivos tradicional. Lanza una CustomCodeException, pero no lanza una FileNotFoundException.

No es responsabilidad de la clase Test lidiar con archivos, sino de leer texto. ¿De dónde? No es relevante.

Evidentemente es una cuestión de nombres, porque efectivamente, estamos yendo a un archivo y la causa es una FileNotFoundException, pero si hiciéramos lo siguiente:

```
public void readSomeText(String path) throws CustomCodeException {
  File f = new File(path);
  try {
    if(f.exists()) {
      this.readSomeFile(path)
    } else {
      this.readSomeCloud(path)
    }
  } catch(Exception e) {
    throw new CustomCodeException(1001, e, "no file at:"+path, e);
  }
}

public void readSomeFile(String path) throws FileNotFoundException {
    FileReader fr = null;
    try {
        File f = new File(path);
        fr = new FileReader(f);
    } catch (FileNotFoundException e) {
      System.out.println(e.getMessage()); //log the exception
      throw e;
    } finally {
        try {
            if(fr != null) {
                fr.close(); // free resources
            }
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }
}


public void readSomeCloud(String path) throws MyResourceNotFoundException
{
  //conectarse a AWS y leer el archivo. Si hay un problema, lanzar
  //MyResourceNotFoundException
}
```

Entonces, readSomeText **solo** lanza CustomCodeException y aísla al código cliente (en este casi nuestro main) de la solución de almacenamiento del archivo a leer. Queda desacoplado nuestro main y su manejo de errores de la forma en que se almacenó el archivo. Si piensan que, en vez de un main, el código cliente de este método fuera otra clase u otro sistema, entonces el poder de desacoplar que tienen las excepciones creadas por nosotros muestra una gran utilidad.

Te invito a crear tus propias excepciones y utilizarlas en tu código.

- ¿Qué tal si lanzas una excepción chequeada ante un problema de casteo?

¡También puedes crear excepciones para validar los valores que pueden tomar los atributos de tus objetos! ¿Dónde más podrías utilizar excepciones que hayas creado?