INFORMATORIO

Taller de Programación Web

Errores y Excepciones

INTRODUCCIÓN

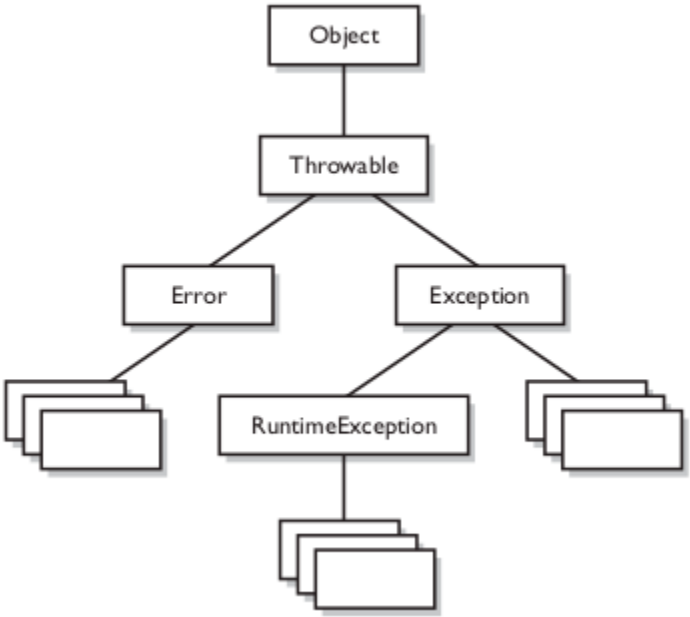
Un error se puede definir en Java como un evento anormal severo donde la aplicación no se puede recuperar (un problema de memoria, sistema operativo, etc).

Una exception, significa una condición excepcional y que altera el flujo normal de la aplicación. Pueden ser muchas las razones que nos lleven a lidiar con excepciones, incluyendo fallas de sistema y bugs. Cuando este evento sucede decimos que una exception es “lanzada” (thrown). El código responsable para lidiar con dicha exception es llamado “exception handler” (gestionador), y atrapa (catches) la exception lanzada.

¿CÓMO FUNCIONA?

Para lidiar con excepciones, usaremos las palabras reservadas `try` y `catch`. El bloque `try` es para definir la región del código en donde la excepción puede llegar a ocurrir (riesgo de que se produzca una exception). Las cláusulas de `catch` (puede ser una o más) comparan el tipo de excepción específica (o perteneciente a un grupo) para ejecutar un bloque de código que lidiara con la misma.

JERARQUÍA DE EXCEPCIONES



Todas las excepciones son subtipo de la clase `Exception`. Esta clase deriva de la clase `Throwable` (que a su vez deriva de `Object`).
Como se puede ver en la figura anterior, dos subclases derivan de `Throwable` (`Error` y `Exception`). Las clases que extiendan (o deriven) de `Error` representan situaciones inusuales que no son causadas por errores de programación, e indican cosas que normalmente no sucederán normalmente durante la ejecución de un programa, como pueden ser errores de memoria, SO, y más. Por lo tanto, la aplicación no se puede recuperar de un `Error`, así que no es necesario controlarlo o manejarlo.
Una excepción no sucede por resultado de un error de programación, sino

porque algún recurso no está disponible o alguna condición requerida para la ejecución correcta no está presente.

Excepciones Chequeadas (Checked Exceptions)

Estas excepciones son las que el compilador de Java nos requiere que controlemos. Podemos controlarlas con un bloque `try/catch` o podemos agregarla a la firma del método (con `throws`) para que sea controlada por quien lo invoca.
Algunos ejemplos de excepciones chequeadas son: `IOException` and `ServletException`.

Excepciones No Chequeadas (Unchecked Exceptions)

Son excepciones que el compilador de Java no requiere que las controlemos. Sencillamente, podemos decir que si una excepción extiende de `RuntimeException`, será No Chequeada; caso contrario chequeada.
Algunos ejemplos de excepciones no chequeadas son: `NullPointerException`, `IllegalArgumentException`, y más.

Manejando Excepciones:

Varias librerías dentro de la API de Java contienen métodos con comentarios (javadoc) anunciando que algo puede salir mal.

Ejemplo:

```
/**
 * @exception FileNotFoundException ...
 */
public Scanner(String nombreArchivo) throws FileNotFoundException {
   // ...
}
```

Se puede decir que estos métodos podrían ser riesgosos si se produce la excepción, pero hay varias formas de manejarlos.

throws

La forma más sencilla de manejar una excepción es relanzarla.

```
public int getSalarioEmpleado(String archivoEmpleado)
    throws FileNotFoundException {

        Scanner contents = new Scanner(new File(archivoEmpleado));
        return Integer.parseInt(contents.nextLine());
}
```

Al ser `FileNotFoundException` una excepción chequeada (checked), esta es la forma más simple de manejar para satisfacer al compilador. Pero significa que el que llame a la función `getSalarioEmpleado` deberá manejarla ahora (sea con `try/catch` o colocar `throws` en su firma).

Atención:
El método parseInt puede lanzar NumberFormatException, pero al ser esta una excepción no chequeada (unchecked) no debemos manejarla.

try/catch

Si queremos manejar la excepción en nuestro código, podemos usar un bloque try/catch.

Podemos relanzar la excepción:

```
public int getSalarioEmpleado(String archivoEmpleado) {
    try {
        Scanner contents = new Scanner(new File(archivoEmpleado));
        return Integer.parseInt(contents.nextLine());
    } catch (FileNotFoundException noFile) {
        throw new IllegalArgumentException("Archivo no encontrado");
    }
}
```

O podemos realizar pasos de recuperación:

```
public int getSalarioEmpleado(String archivoEmpleado) {
    try {
        Scanner contents = new Scanner(new File(archivoEmpleado));
        return Integer.parseInt(contents.nextLine());
    } catch ( FileNotFoundException noFile ) {
        logger.warn("Archivo no encontrado, retornando valor por
defecto.");
        return 0;
    }
}
```

finally

Ocurren ocasiones en donde necesitamos ejecutar código sin importar si ocurre una excepción o no. Para eso utilizaremos la palabra reservada `finally`.

En los anteriores ejemplos no se tuvo en consideración una operación de limpieza (cleanup), luego de ocupar el objeto `Scanner` deberías cerrar (close) el mismo sea que hayamos leído o no algo (muchas de las clases para operaciones de entrada-salida contienen métodos close para “informar” que no se utilizara más).

Ejemplo sin catch:

```
public int getSalarioEmpleado(String archivoEmpleado)
throws FileNotFoundException {
  Scanner contenidos = null;
  try {
    contenidos = new Scanner(new File(archivoEmpleado));
      return Integer.parseInt(contenidos.nextLine());
  } finally {
      if (contenidos != null) {
        contenidos.close();
      }
  }
}
```

Aquí podemos observar que si la excepción `FileNotFoundException` se produce, antes de lanzarla a quien invoco este método, primero deberá ejecutar el contenido del bloque `finally`.

Ejemplo con catch:

```
   public int getSalarioEmpleado(String archivoEmpleado) {
        Scanner contenidos;
        try {
            contenidos = new Scanner(new File(archivoEmpleado));
            return Integer.parseInt(contenidos.nextLine());
        } catch (FileNotFoundException noFile ) {
            logger.warn("Archivo no encontrado, retornando valor por
defecto.");
            return 0;
        } finally {
            try {
                if (contenidos != null) {
                    contenidos.close();
                }
            } catch (IOException io) {
                logger.error("No se pudo cerrar el objeto reader!", io);
            }
        }
   }
```

Podemos observar que aquí también agregamos un `try/catch` para el método close, porque es posible que se produzca una excepción antes de que se cree el objeto de tipo `Scanner`, así que para agregar un nivel más de seguridad hemos agregado este bloque.

try-with-resources

A partir de Java 7, se puede simplificar la sintaxis con las clases que heredan de `AutoCloseable`.

```
public int getSalarioEmpleado(String archivoEmpleado) {
    try (Scanner contenidos = new Scanner(new File(archivoEmpleado))) {
      return Integer.parseInt(contenidos.nextLine());
    } catch (FileNotFoundException e ) {
      logger.warn("Archivo no encontrado, retornando valor por defecto.");
      return 0;
    }
}
```

Cuando referenciamos que hay un tipo `AutoCloseable` en la declaración del `try` , no necesitaremos cerrarlo nosotros. Podemos agregar un bloque `finally` para otras operaciones si lo deseamos.