ELO329 - Diseño y Programación Orientados a Objetos

Manejo de errores: Excepciones en Java

Agustín González
Patricio Olivares

Excepciones (o errores)

- Los lenguajes orientados a objeto han buscado la forma de facilitar la programación de las **condiciones de error** en un programa.
- Muchas cosas pueden generar excepciones (o errores): Errores de hardware (falla de disco), de programa (acceso fuera de rango en arreglo), apertura de archivo inexistente, ingreso de un depósito negativo, probabilidad mayor que 1, etc.
- En lugar de mezclar el código asociado a la lógica principal del programa con el tratamiento de excepciones, lo cual dificulta la claridad de la tarea principal del programa, los lenguajes orientados a objetos como **Java y C++ disponen un mecanismo de excepciones que separa la parte fundamental del código (mayor % de los casos) de las situaciones de error**.
- U**na excepción es un evento que ocurre durante la ejecución de un programa que rompe el flujo normal de ejecución**. Cuando se habla de excepciones nos referimos a un evento excepcional.

Ejemplo: Motivación

- Supongamos que queremos leer un archivo completo a memoria:

```
// Lógica del programa
readFile () {
    Abrir el archivo; // (1)
    Determinar el largo del archivo; // (2)
    Localizar esa cantidad de memoria; // (3)
    Leer el archivo en memoria; // (4)
    Cerrar el archivo; // (5)
}
```

Ejemplo: implementación sin excepciones

```
errorCodeType readFile () { // Comentarios marcan la lógica del código
    initialize errorCode = 0;
    Abrir el archivo; // (1)
    if (theFileIsOpen) {
        Determinar el largo del archivo; // (2)
        if (gotTheFileLength) {
            Localizar esa cantidad de memoria; // (3)
            if (gotEnoughMemory) {
                Leer el archivo en memoria; // (4)
                if (readFailed) errorCode = -1;
            } else errorCode = -2;
        } else errorCode = -3;
        Cerrar el archivo; // (5)
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else errorCode = errorCode and -4;
    } else errorCode = -5;
    return errorCode;
}
```

Ejemplo: implementación con excepeciones

```
void readFile () {
    try {
        abrir un archivo;
        determinar su tamaño;
        localizar esa cantidad de memoria;
        leer el archivo en memoria;
        cerrar el archivo;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Ejemplo: implementación con excepeciones

- Cuando el código lanza una excepción, se detiene la secuencia del código restante en el `try` y se continúa en el `catch` correspondiente.
- Si no hay `try`, se retorna del método (esto es relanzar la excepción).

Captura de Excepciones (completo)

- El manejo de excepciones se logra con el bloque `try`

```
try {
    //lógicao normal del código
} catch (e-clase1 e1 ) {
    // sentencias tratamiento error e-clase1
} catch (e-clase2 e2 ) {
    // sentencias tratamiento error e-clase2
} ...
finally { // esta parte es opcional. Si está, se ejecuta siempre
    //sentencias
}
```

Captura de Excepciones (completo)

- La cláusula `finally` es ejecutada con posterioridad cualquiera sea la condición de término del `try` (con o sin error, `return`, `break`, `continue`). Esta sección permite dejar las cosas consistentes antes del término del bloque `try`.

Tipos de Excepciones

- Las hay de dos tipos
  - Aquellas generadas por el lenguaje Java. Éstas se generan cuando hay errores de ejecución, como al tratar de acceder a métodos de una referencia no asignada a un objeto, división por cero, etc. En este caso no se exige el bloque `try`.
  - Aquellas incluidas por el programador vía paquetes o sus propias clases. Aquí sí se exige bloque `try`.
- El compilador chequea por la captura de las excepciones lanzadas por los métodos invocados en el código.
- Si una excepción no es capturada (con sentencia `try-catch`), debe ser **relanzada**.

Reenviando Excepciones: dos formas

```
public static void doio (InputStream in, OutputStream out)throws IOException {
    int c; // en caso de más de una excepción throws exp1, exp2
    while (( c=in.read()) >=0 ) {
        c = Character.toLowerCase((char) c);
        out.write(c);
    }
}
```

- Si la excepción no es manejada con `try`, debe ser reenviada.

Reenviando Excepciones: dos formas

- Alternativamente

```
public static void doio (InputStream in, OutputStream out)throws IOException {
    int c;
    try {
        while (( c=in.read()) >=0 ) {
            c= Character.toLowerCase( (char) c);
            out.write(c);
        }
    } catch ( IOException t )
    throw t;
}
```

- En este caso, el método envía una excepción - que aquí corresponde al mismo objeto capturado - por lo tanto, se debe declarar en la cláusula `throws`.
- Si un método usa la sentencia `throw` debe indicarlo en su declaración con la cláusula `throws`.
- En este caso es responsabilidad de quien llame a `doio()` atrapar la excepción o relanzarla. Así esto suba hasta posiblemente llegar al método main.

Creación de tus propias excepciones

- Siempre es posible lanzar alguna excepción de las ya definidas en Java (`IOException` por ejemplo).
- También se puede definir nuevas excepciones creando clases derivadas de las clases `Error` o `Exception`.

```
class ZeroDenominatorException extends Exception {
    private int n;
    public ZeroDenominadorException(String s) {
        super(s);
    }
    public setNumerator(int _n) { n = _n; }
}
```

Creación de tus propias excepciones

- Luego la podemos usar como en este constructor:

```
public Fraction (int n, int d) throws ZeroDenominatorException {
    if (d == 0) {
        ZeroDenominatorException myExc = new
        ZeroDenominatorExceptio("Fraction: Fraction with 0 denominator?");
        myExc.setNumerator(n);
        throw (myExc);
    }
}
```

Ventajas de las Excepciones

- Claridad y simplicidad de la tarea a realizar más frecuentemente.
- Propaga los errores hacia atrás hasta el punto donde se puede tomar una acción.
- Se agrupan los errores según su naturaleza.
- Ej:
  - Si hay más de un archivo que se abre, basta con un código para capturar tal caso.
  - Si se lanzan excepciones que son todas subclases de una base, basta con capturar la base para manejar cualquiera de sus instancias derivadas.

Cosas a tomar en cuenta

- Las excepciones consumen tiempo, no usarlas cuando hay alternativas mejores, ejemplo (verlo en casa) `ExceptionalTest.java`
- Agrupar el manejo de varias excepciones en un único `try` ...es bueno.
- En cada caso evaluar si es mejor atrapar la excepción o reenviarla a código llamador
    - Ejemplo: Quien llame a `readStuff` puede manejar la excepción de mejor forma que aquí.

```
public void readStuff(String name) throws IOException {
    FileInputStream in= new FileInputStream(name);
    ...
}
```

- Para redefinir un método que no lanza excepciones, la redefinición tampoco debe hacerlo.