Resumen PAV (3)

Excepciones (*exceptions*)

Este resumen ha sido elaborado para ser leído después de haber asistido a clase: pretende fijar y profundizar en los conceptos ya explicados.

Introducción

Un tema frecuente en los lenguajes de programación es cómo se manejan los errores durante la ejecución del programa. ¿Qué sucedería si las condiciones que he supuesto cuando escribía el código no se cumplen?

Un enfoque común es diseñar los métodos o funciones de modo que devuelvan un valor previamente determinado si se da una condición de error. Consideremos por ejemplo la función `fopen` del C++. Esta función abre un archivo y devuelve un "file descriptor" (un valor que representa ese archivo). Si por algún motivo se produjo un error, `fopen` devuelve `NULL`.

Consideremos el siguiente ejemplo[1]:

```
#include <stdio.h>
int main()  {
      File *pFile = fopen("readme.txt", "rw");    // abrimos el archivo
      fputs("Esto se escribirá al archivo...", pFile);
                                 // escribimos una cadena al archivo
      fclose();                  // cerramos el archivo
}
```

Puede ser que `fopen` no pueda abrir el archivo por algún motivo que desconocemos. Si eso sucede, `pFile` será igual a `NULL` (pues es el valor que `fopen` devuelve cuando hay algún error) y cuando se ejecute la siguiente línea, el programa se detendrá, porque `fputs` necesita un archivo válido para poder escribir la cadena.

Lo importante del ejemplo es que **el lenguaje no nos fuerza a enfrentar el hecho de que podría producirse un error**. Queda a la responsabilidad del programador, si es que éste está dispuesto a tomarse la molestia y acabar las cosas hasta los últimos detalles.

Java y otros lenguajes más recientes adoptan otro enfoque al manejo de errores. Las posibles condiciones de error dejan de considerarse un situación casual. El manejo de errores se ha *formalizado* como parte del lenguaje –con palabras reservadas específicas--, de modo que si hay la posibilidad de que se produzca un error el programador **deba tenerlo en cuenta**.

Excepciones y errores

Se considera una excepción en Java a *una situación que está fuera de las condiciones previsibles del problema que estoy resolviendo*. No se trata de errores que, por el problema tengo entre manos, se pueden prever y deben prever.

Por ejemplo, la función `readLine()` de la clase BufferedReader lee la línea siguiente de un stream[2]. Si estoy leyendo un archivo, es previsible que en algún momento leeré la última línea del archivo. Esto no es una excepción, sino una condición que si no manejo dará un error. Pero es un error perfectamente previsible.

Por eso, la definición de función `readLine()` dice que `readLine()` devuelve un `String` con la

1 No interesa tanto entender exactamente qué hace cada función del ejemplo, sino fijarmos en cómo se maneja el error.
2 Un stream... un archivo, del teclado, de cualquier flujo de datos.

siguiente línea del stream (archivo o lo que sea) *y null si ya no hay más líneas que leer*. En cambio, produce una excepción si se da otro tipo de error. Este segundo tipo de error, excepcional, es tal que el programa no puede continuar si alguien no lo arregla. Y Java fuerza a que esto sea así.

El programa sería algo así[3]:

```
String s;         // nuestra variable para guardar la línea
s = in.readLine();
while (s != null)  {
      System.out.println(s);
      s = in.readLine();
}
```

De modo más sintético, se suele escribir así:

```
String s;
while ( (s = in.readLine()) != null) {
      System.out.println(s);
}
```

Sin embargo, este código no compilará, pues `readLine()` puede generar una excepción y Java nos fuerza a que encaremos ese hecho. Veremos más adelante cómo se hace. Interesa notar en el ejemplo cómo se está manejando la posibilidad de que no hayan más líneas que leer: comparando s con `null`, y terminando elegantemente el loop, si se verifica esa condición. Pero, repetimos, esa no es una situación excepcional. Excepcional sería que en plena lectura del archivo derramemos una jarra de agua en el teclado o, menos dramático, se malograse el CD o el disco duro del que estamos leyendo.

Otro ejemplo es la *división entre cero*.

Hay problemas en los que el denominador de una fórmula previsiblemente puede ser cero (por ejemplo, en un sistema amortiguado, hay una frecuencia de resonancia. Si el sistema opera a esa frecuencia, se autodestruye... justamente éso quiere decir denominador cero en este caso). Pero esto no es una excepción: es el comportamiento normal (y que corresponde a lo que sucede en la realidad) del modelo matemático. Por eso, antes de hacer la operación sería normal que verifiquemos que el denominador no es cero.

En cambio, en otros casos el denominador cero es un tema verdaderamente excepcional: no estaba previsto en el curso de la resolución del problema. No podemos estar comparando el denominador con cero cada vez que hacemos una división. Es más sencillo dejar que se produzca una excepción, y manejarla.

Generación y manejo de excepciones

Cuando un programa está ejecutándose, hay que considerar dos "momentos" distintos:

a) **la situación excepcional**, que es detectada por el código que alguien o uno mismo ha escrito. El programador decide *generar una excepción* porque su código, en lo que a él respecta, no puede continuar. Generar una excepción es disparar una bengala de emergencia para indicarle al compilador que nuestro buque (nuestro programa) se está hundiendo.

b) **el manejo de la excepción**: la operación de rescate del buque. De esto se encargarán

3 Suponiendo que `in` es un objeto `BufferedReader`.

otros, no los que están en el buque[4].

Generación de excepciones

Una excepción se genera en Java usando la palabra reservada `throw`. En el siguiente código, el programador se quiere asegurar de que `lastRoom` no sea `null` antes de proseguir con su programa:

```
if (lastRoom == null)  {
      throw new Exception();
}
else {
      /* sigue la ejecución normalmente */
}
```

`lastRoom == null` es un hecho que no debió darse, según la lógica de resolución del problema. Si se da, no podemos seguir. Estamos desconcertados: ¿cómo es que ha pasado esto? No nos queda otra alternativa que generar una excepción.

Cuando escribimos un método que puede generar una excepcion (en términos prácticos, cuando en alguna de sus líneas el método usa la expresión `throw`) hay que decirlo así en la declaración del método. De lo contrario, el compilador se quejará. Para esto, se usa otra palabra reservada: `throws`. Siguiendo con el código anterior[5]:

```
public void volver( ) throws Exception {

      if ( lastRoom == null) {
            // imposible... lastRoom nunca debería ser null
            throws new Exception( );
      } else {
            // El código sigue ejecutándose normalmente
            Room tmp = currentRoom;
            currentRoom = lastRoom;
            lastRoom = tmp;
      }
}
```

La línea `throw new Exception( )` merecería un poco más de comentarios. Como se puede deducir, `Exception( )` es simplemente el constructor de la clase[6] `Exception`, que está definida en la librería estándar de Java[7]. Por tanto, cuando decimos `new Exception( )`, estamos creando un nuevo objeto de tipo `Exception`[8]. Hablaremos más de esto después de explicar cómo se manejan las excepciones.

---

4 Dentro de los límites de esta analogía, por supuesto. En la práctica puede ser que el que genere la excepción y escriba el código para manejarla sean la misma persona.

5 Se sobreentiende que `lastRoom` y `currentRoom` son variables de la clase que contiene al método `volver`, que ya están definidas.

6 Es una clase que implementa una serie de métodos determinados, y por eso se puede usar con `throw`. Pero no es tema de nuestro resumen, de modo que no nos detendremos en este detallito. Simplemente diremos que no todas las clases se pueden usar con `throw`, y nosotros nos limitaremos por ahora a usar a `Exception` y sus parientes.

7 Está definida en `java.lang`, que se importa siempre automáticamente, de modo que Exception está siempre disponible en los programas Java.

8 De hecho, podríamos escribir el código así, pues `Exception` es una clase común y silvestre:

```
Exception e = new Exception( );
throw e;
```

Manejo de las excepciones

¿Qué sucede cuando en un programa que está ejecutándose Java se encuentra con la línea `throw new Exception( )`?

1. El programa se detiene.
2. Java toma el objeto `Exception( )` y "lo saca del contexto actual". Sacar del contexto quiere decir que *sube a un nivel de anidamiento superior en el programa*, lo que en la práctica normalmente significa salir del método que se está ejecutando.
3. El mecanismo de manejo de excepciones trata de buscar un lugar adecuado para que continúe la ejecución del programa. Este lugar adecuado se llama *exception handler* (manejador de excepciones). Su cometido es que el programa se recupere del accidente y pueda continuar normalmente, o termine de un modo elegante. El manejador de excepciones recibe el objeto `Exception` que creó la parte del programa que generó la excepción.

Para entender cómo se maneja la excepción en la práctica, vamos a ver las cosas desde el punto de vista del que *usa* un método que puede generar una excepción (en vez del punto de vista del que escribe el método que genera la excepción).

Para capturar la excepción, debemos poner el código que podría generarla dentro de una región protegida (*guarded region*). Se usa la palabra reservada `try` para señalar la región. La "región protegida" no es más que un bloque de código encerrado entre llaves ({ }). A continuación debe ir otra región marcada con la palabra reservada `catch`. Éste es el bloque al que saltará el programa si se genera una excepción dentro del bloque `try`.

Siguiendo con el ejemplo anterior, si queremos usar el método `volver()` en nuestro código, no podemos obviar el hecho de que puede generar una excepción[9]. Necesitamos encerrar la zona del código que llame a volver en un área restringida:

```
/* ... arriba más código */
if ( commandWord.equals( "volver" ) {
      try {
            volver( );

      } catch (Exception e) {

            System.out.println("Error inesperado.");

      }
}
```

`volver( )` está encerrada en un bloque `try`, pues puede generar una excepción.

Este bloque de código sólo se ejecutará si volver produce una excepción..

Por supuesto, podemos escribir todo el código que necesitemos (y llamar a otros métodos, crear objetos, etc.) dentro de los bloques `try` y `catch`. Pero lo ideal es restringir el bloque a las líneas de código que pueden generar la excepción, y no a todo el método.

Hay otro modo de lidiar con la excepción que puede arrojar `volver( )` : podríamos no manejarla, sino pasársela al método que llamó a nuestro método. Si el código anterior estuviera dentro de un método `proccesCommand`, tendríamos que declarar que ése método arroja una excepción.

antes: `public boolean processCommand( Command command) {`

9 Pues está definido como `public void volver( ) throws Exception { ... }`.

```
        (...)      }
```

ahora[10]:

```
public boolean processCommand( Command command)
throws Exception  {
 (...)      }
```

La clase Exception

Ya hemos dicho que `Exception` es una clase, y por eso podemos decir `new Exception( )`. De este hecho se derivan algunas consecuencias interesantes:

a) *La clase Exception puede tener más de un constructor*, y de hecho los tiene:

`public Exception( )`, el constructor sin parámetros que hemos venido usando.

`public Exception( String message )`, un constructor que permite almacenar un String dentro del objeto `Exception`. Este "mensaje" se puede recuperar usando el método `String getMessage( )` de la clase `Exception`.

¿De qué nos serviría poder almacenar un mensaje dentro del objeto que se crea cuando se lanza una excepción? Por ejemplo, para que el código que maneje la excepción pueda tener alguna información adicional sobre el error producido. Como el método que genera la excepción no tiene idea ni de quién lo ha llamado ni de quién va a manejar la excepción, el único modo de pasar alguna información es el objeto `Exception`.

Podríamos reescribir el código de `volver( )` y `processCommand( )`:

```
public void volver( ) throws Exception {

      if ( lastRoom == null) {
            // imposible... lastRoom nunca debería ser null
            throws new Exception("Error! lastRoom es null" );
      } else {
            // El código sigue ejecutándose normalmente
            Room tmp = currentRoom;
            currentRoom = lastRoom;
            lastRoom = tmp;
      }
}


public boolean processCommand( Command command ) {

      /* ... arriba más código */
      if ( commandWord.equals( "volver" ) {
            try {
                  volver( );
            } catch (Exception e) {
                  System.out.println( e.getMessage() );

            }
      }
}
```

b) `Exception` *tiene subclases.* El hecho de que las situaciones que llevan a generar una excepción sean inesperadas no quiere decir que estas situaciones no se puedan ordenar de algún modo.

Cuando creamos una sublcase, estamos especializando la superclase, delimitando con

10 La palabra throw puede ir también en la misma línea que `processCommand( ...)`. A Java realmente no le importa.

más precisión su campo de acción. Es justamente lo que hace Java al definir subclases de `Exception`.

Por ejemplo, los errores que pueden producirse cuando se lee o escribe a un archivo o, más en general, a un stream, son errores de entrada/salida (Input/Output). Java define, por tanto, una excepción llamada `IOException`. Esta subclase de `Exception` no añade ninguna funcionalidad, simplemente nos permite identificar, por su nombre, el tipo de error que se ha producido. De hecho readLine( ) de BufferedReader no dice `throw new Exception( );`, sino `throw new IOException( );`

Java define en su librería estándar muchísimos tipos de excepciones. Nosotros podemos saber qué tipo de excepción genera un método determinado fijándonos en la documentación de la librería de clases.

Así, la documentación del método `readLine( )` dice:

readLine

```
public String readLine()
          throws IOException
```

Read a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

Returns:
A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

Throws:
`IOException` - If an I/O error occurs

c) *Podemos definir nuestras propias clases de excepciones derivando subclases de* `Exception` *o de sus subclases.* No necesitamos añadir ninguna funcionalidad, basta crear un nombre único:

```
public VolverException extends Exception   {  /* vacío */   }
```

Dado que hay distintos tipos de excepciones, un método puede generar distintos tipos de excepciones dependiendo de problema que se haya producido:

```
public unMetodo( )  throws Exception  {
      if (            )  {
            /* en algún punto del método, se da un error, y entonces...
            generamos la excepción, y aquí termina la ejecución del
            método */
            throw new IOException ( );
      }

      /* en otro punto del método, podría ocurrir un error distinto
      que nos fuerza a generar una excepción, distinta a la anterior */
      throw new NullPointerException( );
}
```

El código que maneja la excepción puede manejar cada excepción por separado:

```
try  {
      System.out.println("Vamos a ejecutar un método");
      unMetodo( );
} catch( IOException e)  {
      /* este bloque se ejecuta si se produjo una IOException */
      (...)
} catch(NullPointerException e) {
      /* este bloque se ejecuta si se produjo
       una NullPointerException */

} catch(Exception e) {
      /* por último, podríamos comparar si
      hay algún tipo de excepción que no es de ninguno
      de los tipos anteriores.  Esto funciona porque
       Exception es la excepción más general[11]... pero tiene que
      ser la última comparación */
}
```

Un ejemplo más, usando `readLine` y `BufferedReader`:

```
public String leeArchivo(String filename) throws IOException {

      String s;
      try {
            BufferedReader in = new BufferedReader(
                        new FileReader( filename ) );
      } catch (IOException e) {
            // esta excepción ha sido arrojada por FileReader.
            throw new IOException("No pude abrir el archivo" + filename
                        + ". Se produjo una excepción "
                        + e.getMessage( ) );
      }

      try {
            while ( (s += in.readLine( ) ) != null ) {
                  ;  // no hago nada, sólo acumulo... ojo con el += en
                     // la línea de arriba
            }
      } catch (IOException e) {
            // Esta excepción ha sido arrojada por readLine( );
            // Aquí imprimo el error, pero no genero una excepción:
            // decido devolver s "cómo esté"
            System.out.println("Se produjo un error al leer el archivo" +
                  e.getMessage( ) );
      }

      in.close( );  // cierro el stream
      return s;
}
```

Comentarios, correcciones y sugerencias: Roberto Zoia (roberto.zoia@gmail.com)

This work is licensed under the Creative Commons Attribution-ShareAlike 2.5 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/2.5/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

11 Así como todos los `Gatos` son `Animales`, todos las clases derivadas de `Exception` son también de tipo `Exception`.