Manejo de Excepciones y otros

Agustín J. González

ELO-326: Seminario II

2do. Sem. 2001

Aspectos varios en una comparación con C++

- Sobrecarga de Operadores
  - Esta característica no es soportada por Java. Los diseñadores del lenguajes optaron por esta opción para hacer el lenguaje y su compilador mas simple.
- Administración de Memoria
  - En comparación a C++, Java es mucho mas simple. Los programadores no necesitan preocuparse por esta.
  - Todos los objetos son almacenados en el heap con la sentencia new. El programador no necesita preocuparse por liberar esta memoria.
  - La recolección de “basura” (garbage collection) se encarga de recuperar la memoria no referenciada.
  - Este proceso es en general lo suficientemente rápido para el el usuario no lo note.

Aspectos varios en una comparación con C++ (cont)

- Parametrización de Clases
  - En C++ estudiamos las plantillas (templates) las cuales son un mecanismo de programación genérica.
  - En Java no existe tal mecanismo. El mecanismo usado por Java fue definir la case Object como la clase base para cualquier clase.
  - Este mecanismo posee como desventaja la necesidad de hacer un cast cuando retornamos objetos de clases como Vector. Esto puede causar errores al almacenar objetos de distinto tipo en un mismo vector.
  - Otra desventaja es la dificultad para leer el código.
    Vector elements; // no nos dice nada sobre los elementos que deseamos almacenar.
  - La gran ventaja de no usar templates es la simplificación del compilador

Excepciones

- Como en C++, Java posee mecanismos para enfrentar detectados durante la ejecución.
- En métodos siempre es posible retornar un código de error.
- Se puede asignar una variable de error “global”
- Se puede imprimir un mensaje de error.
- Las soluciones previas pueden ser usadas en tareas en la U pero generalmente no son aceptable en productos comerciales.
- Como en C++, Java ofrece un mecanismo para señalar excepciones.
- En Java los objetos lanzados deben ser instancias de clases derivadas de Throwable.

  Ej.

  ```
  Throwable e = new IllegalArgumentException(“Stack underflow”);
  throw e;
  ```

  O alternativamente

  ```
  throw new IllegalArgumentException(“Stack underflow”);
  ```

Manejo de Excepciones

- El manejo de excepciones se logra con el bloque try

```
try {
        // código
} catch (StackError e )
{
        // código que se hace cargo del error reportado en e
}
```

- El bloque try puede manejar múltiples excepciones:

```
try {
        // código
} catch (StackError e )
{
        // código para manejar el error de stack
} catch (MathError me)
{
        // código para manejar el error matemático indicado en me.
}
```

Captura de Excepciones (completo)

- El bloque try tiene la forma general:

```
try {
        //sentencias
} catch (e-type1 e ) {
        // sentencias
} catch (e-type2 e ) {
        // sentencias
} ...
finally {
        //sentencias
}
```

Captura de Excepciones: Ejemplo 1

- public static void doio (InputStream in, OutputStream out) {

```
int c;
try { while (( c=in.read()) >=0 )
            { c= Character.toLowerCase( (char) c);
              out.write( c );
            }
} catch ( IOException e ) {
  System.err.println("doio: I/O Problem");
  System.exit(1);
}
}
```

- La cláusula finally cuando cualquiera de las excepciones ocurre. Esta sección permite dejar las cosas consistentes antes del término del bloque try.

Captura de Excepciones: Ejemplo 2

- .....

```
try { FileInputStream infile = new FileInputStream(argv[0]);
        File tmp_file = new File(tmp_name);
        ....
} catch (FileNotFoundException e) {
        System.err.println("Can´t open input file "+ argv[0]);
        error = true;
} catch (IOException e ) {
        System.err.println("Can´t open temporary file "+tmp_name);
        error = true;
}finally {
        if ( infile != null) infile.close();
        if (tmp_file != null) tmp_file.close();
        if (error) System.exit();
}
```

- El código de la sección finally es ejecutado no importando si el bloque try terminó normalmente, por excepción, por return, o break.

Tipos de Excepciones

- Las hay de dos tipos
- Aquellas generadas por el interpretador de Java. Estas 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.
- Aquellas no generadas por el interprete y que son usadas por el código del programador.
- El compilador chequea por la captura de las excepciones lanzadas por los objetos usados en el código.
- Si una excepción no es capturada debe ser relanzada.

Reenviando Excepciones

- **public static void doio (InputStream in, OutputStream out)**
  throws IOException // en caso de más de una excepción throws exp1, exp2, expi

```
{
        int c;
        while (( c=in.read()) >=0 )
        {
                c= Character.toLowerCase( (char) c);
                out.write( c );
        }
}
```

Si la excepción no es capturada, se entiende reenviada

- **Alternativamente:**
- **public static void doio (InputStream in, OutputStream out) throws Throwable {**

```
        int c;
        try { while (( c=in.read()) >=0 )
                { c= Character.toLowerCase( (char) c);
                  out.write( c );
                }
        } catch ( Throwable t ) {
                throw t;
        }
}
```

En este caso el método envía una excepción - que aquí corresponde al mismo objeto capturado -por lo tanto debe declarase en la cláusula throws.

- **!!! Si el método unas la cláusula throw debe indicarlo en su declaración con la cláusula throws.**

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 () {}
        public ZeroDenominadorException(String s) {
                super(s);
        }
        public setNumerator(int _n) { n = _n}
        // otros métodos de interés
}
```

- Luego la podemos usar como en:

```
....
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);
....
}
```

Cuando no podemos relanzar una excepción

- Hay situaciones en que estamos obligados a manejar una excepción. Consideremos por ejemplo:

```
class MyApplet extends Applet {
        public void paint (Graphics g) {
                FileInputStream in = new FileInputStream("input.dat"); //ERROR
                ....
        }
}
```

- Se crea aquí un problema porque dado que la intención es sobremontar un método de la clase Applet - método paint- el cual no genera ninguna excepción. Si un método no genera excepciones la función que lo sobremonta no puede lanzar excepciones (problema en Java).
- Lo previo obliga a que debamos hacernos cargos de la excepción.

```
 class MyApplet extends Applet {
        public void paint (Graphics g) {
          try {
                FileInputStream in = new FileInputStream("input.dat"); //ERROR
                ....
        } catch (Exception e) {//.....}
}
```