élogos

CONTROL DE EXCEPCIONES. EXCEPCIONES PERSONALIZADAS

élogos

ÍNDICE

CONTROL DE EXCEPCIONES. EXCEPCIONES PERSONALIZADAS

1. Utilización de los bloques try, catch y finally

Descripción genérica de excepciones: try/catch/finally

Estructura Genérica

```
try {
      // código a ejecutarse y qué puede producir
      // algún tipo de excepción
                  }
catch(Excepcion1 e1) {
      // aquí se ejecutaría el código necesario si dentro
      // de try ocurriera la excepción denominada Excepcion1
                  }
catch(Excepcion2 e2) {
      // aquí se ejecutaría el código necesario si dentro
      // de try ocurriera la excepción denominada Excepcion2
                  }
                  ...
finally {
      // aquí se ejecutaría SIEMPRE el código que nos
      // interese, como pueda ser limpieza de variables ...
                  }
```

try: El bloque try constituye un conjunto de sentencias como otras, pero que pueden lanzar excepciones que serán tratadas a continuación.

Catch: El bloque o los bloques catch, van a manejar cada uno el tipo de excepción al que hace referencia si esta ocurre en el bloque try. Su argumento debe ser del tipo Throwable o una subclase de éste.

Tratamiento de múltiple excepciones

Se pueden capturar varias excepciones en un mismo **try** colocando los bloques catch seguidos. Si algunas de estas clases de excepción están relacionadas por la herencia, hay que tener en cuenta que los bloques de las clases más específicas (subclases) deben situarse delante de los de las clases más generales(superclases), pues si se produce una excepción dentro del **try** el programa ejecutará el primer bloque que encuentre coincidente con el tipo de excepción.

Control de excepciones. Excepciones personalizadas

```
try {
      algunMetodoExcepcional();
                  }
catch(NullPointerException e1) {
      // Tratamiento
                  }
catch(RuntimeException e2) {
      // Tratamiento
                  }
catch(IOException e2) {
      // Tratamiento
                  }
catch(IOException e2) {
      // Tratamiento que se realizará siempre en caso de que el código del programa no entre por ninguno
            de los catch anteriores, esto funciona como un comodín, siempre al final ponemos la clase
            superior(Exception) y así nos aseguramos de que siempre se ejecutara algún catch en caso de
            error
                  }
```

Cláusula finally

Permite definir una serie de instrucciones de obligada ejecución. Este bloque se ejecutará tanto si se produce la excepción como si no. Por tanto, si hay una determinada operación que queramos asegurar su ejecución, como el cierre de una conexión con la base de datos, la incluiremos dentro de un bloque finally.

```
algunaClaseArchivo f = new algunaClaseArchivo();
if (f.open("/a/file/name/path")) {      // instrucción para abrir un fichero     try {
algunMetodoExcepcional();
}
finally {
f.close();
// instrucción para cerrar un fichero
 }
 }
```

El cierre del fichero se producirá en cualquier caso se produzca o no la excepción.

Esta forma de trabajar se comporta mejor que la siguiente:

```
algunaClaseArchivo f = new algunaClaseArchivo();
if (f.open("/a/file/name/path")) {
try {
algunMetodoExcepcional();
}
catch (Throwable t) {
f.close();
throw t;
 }
}
```

Características de la cláusula finally

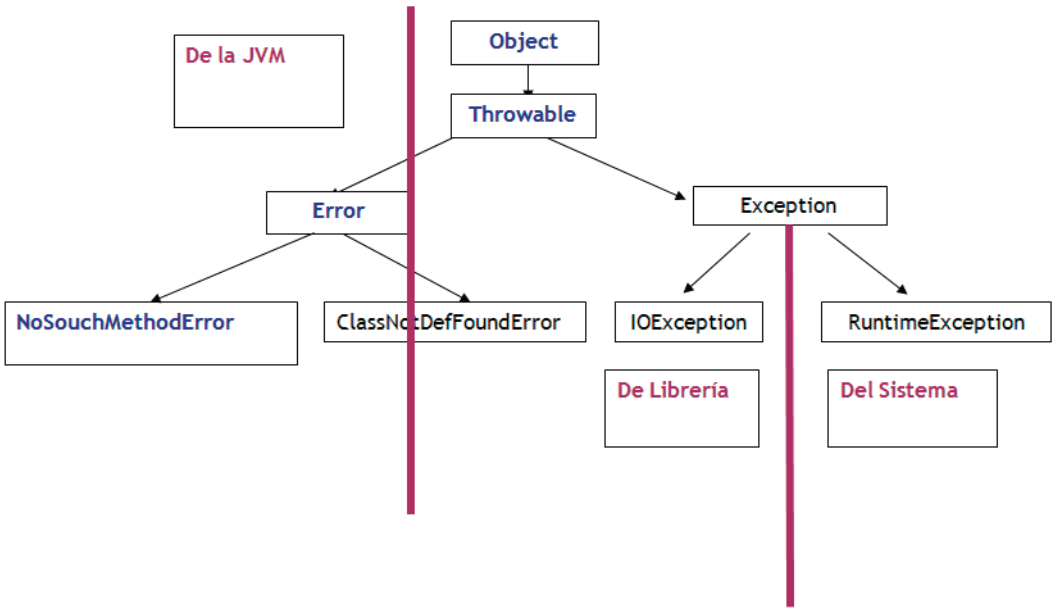
- La cláusula finally se usa generalmente para la limpieza del sistema (cierre de ficheros, liberación de recursos, etc.).

- Una propiedad importante del bloque finally es que siempre se va a ejecutar aunque sólo se ejecutase una porción de try y se saltase con un break o return.

- El orden establecido de ejecución de la estructura completa es: ejecutar try y si ocurre alguna excepción se salta a los catch. Una solución es que se coja aquí la excepción y se trate, otra es pasársela para que la coja el nivel superior subiendo en la jerarquía. Después de esto siempre se ejecutara el bloque finally.

- Una utilidad de la pareja try/finally, y saliéndonos de las excepciones, está en usar try como etiqueta para un conjunto de sentencias y finally como bloque de liberación de recursos después de que éstas se ejecutaran siempre, haya o no dentro del try breaks continues o returns.

Repaso



2. La lista de throws

Si un método es capaz de provocar una excepción que no cubre, deberá especificar este comportamiento en el prototipo para que los métodos que le llaman puedan protegerse frente a la posible excepción.

```
<tipo Retorno> <Nombre Método> (<Lista Parámetro>)
        throws <Lista Throws>
```

Hay que poner la lista de throws para todas las excepciones derivadas de Exception excepto las derivadas de RunTimeException.

```
class ThrowsDemo {
static void Proc1() throws IOException {
System.out.println ("Algo va mal.");
 throw new IOException ("Demo");
            }
public static void main (String [] args) {
            try{
            Proc1();
            }catch(IOException e){
            // Instrucciones para tratar la excepción}
            }
          }
```

Enlace Dinámico

Cuando se redefine un método las excepciones que lance el método redefinido tienen que ser como máximo las excepciones de la base (Enlace Dinámico).

- Si en la derivada se lanza una excepción no prevista en la base se va a producir un error.

- Se pueden quitar excepciones de la lista.

- Los constructores también pueden llevar lista de throws y en caso de que se produzca la excepción en el constructor, el objeto se crea pero a la referencia no se asigna el objeto, con lo cual, el Sistema de Recogida de Java liberará la memoria del objeto creado.

```
Punto p;
try { Punto p=new Punto (-1, -1); }
            // La operación de asignación no se realiza, la referencia
            // no apunta al objeto
catch(IllegarParameterException ex) { . . . }
```

3. Crear excepciones personalizadas

Crear tipo de excepciones personalizadas

Mediante las excepciones personalizadas podemos informar a las clases que van a hacer uso de nuestros métodos de situaciones anómalas ocurridas en los mismos.

Control de excepciones. Excepciones personalizadas

La única condición que deben poseer estas clases es derivar de Exception aunque conviene que dispongan de un constructor personalizado y que sobrescriban toString().

Ejemplo

```
class ExcepcPersonal extends Exception { //Deriva de Exception.
private int valor;
ExcepcPersonal (int valor) {
 this.valor = valor; }
String toString() {
      return "El valor "+valor" es demasiado grande.";
            }
          } // fin de clase
public class ExcepcionDemo {
static void contar (int a) throws ExcepcPersonal {
      System.out.println ("Llamada contar con valor "+a);
 if (a > 10) throw new ExcepcPersonal (a);
      System.out.println ("Salida normal "+a);
            }
public static void main (String [] args) {
  try { contar(1);
        contar(20);
             }
  catch(ExcepcPersonal ex) { System.out.println("Excepción capturada "+a); }
            }
          } // Fin de clase.
```

Control de excepciones. Excepciones personalizadas

Ventajas

La ventaja del uso de excepciones frente al manejo tradicional de errores, es que el manejo de errores en el uso tradicional estaba dentro de la lógica del problema mientras que con el tratamiento de excepciones existe una separación clara entre el manejo de la misma y la lógica del problema.

Manejo tradicional

```
int leerfichero (const *char Nombrefichero) {
int CodError = 0;
FILE fichero;
CodError = Openfile (Nombrefichero,&hfichero);
if (CodError == 0) {
   int longitud;
   CodError = Getfilelength(hfichero,&longitud);
if (CodError == 0) {
....
Algoritmo del problema
....
}
}
return CodError;
}
```

Control de excepciones. Excepciones personalizadas

Manejo de Excepciones

```
static void leerfichero (String nombrefichero) throw IOException {
try {
File  f = new File();
f.openFile(nombrefichero);
int longitud = f.getFileLength();
....
Algoritmo del problema
....
}

catch(FileOPenFailedException       ex)       {
System.out.println ("No existe el fichero");
        }

catch(DeterminationFailedException     ex)     {
System.out.println ("No se sabe la longitud");
}
      }
```