DESARROLLO DE APLICACIONES MULTIPLATAFORMA

PROGRAMACIÓN

TEMA 7

CONTROL DE ERRORES, EXCEPCIONES Y DOCUMENTACIÓN

CONTENIDOS

7.1. INTRODUCCIÓN.

Cuando desarrollamos un programa se pueden producir diversos tipos de errores que debemos controlar y solucionar:

- Errores de compilación: normalmente errores de sintaxis que impiden su ejecución, afortunadamente los IDEs ayudan a reconocerlos y nos dan una idea de lo que puede estar pasando, en última instancia, el compilador del lenguaje nos mostrará el error producido.

  Ej.: comandos mal escritos, orden incorrecto de operaciones, variables con tipos no coincidentes, omisión de elementos necesarios,…

- Errores de ejecución: ocurren cuando el programa se está ejecutando, impidiendo que el programa pueda continuar. Son más difíciles de detectar.

  Ej.: acceso a memoria restringida, división por cero, recursos no disponibles,…

- Errores lógicos: el programa compila y se ejecuta correctamente pero no muestra los resultados esperados. El programa no tiene por qué detenerse, normalmente se deben a un mal diseño de los algoritmos utilizados.

7.2. DEPURACIÓN DE PROGRAMAS.

La depuración de programas (*debugging*) es el proceso de identificar y corregir errores de programación.

- *El término "bug" (bicho) proviene de la época de las computadoras de "válvula termoiónica", en las cuales, los problemas se generaban por los insectos que eran atraídos por las luces y estropeaban el equipo.*
- *Si bien existen técnicas para la revisión sistemática del código fuente y se cuenta con medios computacionales para la detección de errores (depuradores), facilidades integradas en los sistemas "lower CASE" y en ambientes de desarrollo integrado IDEs, sigue siendo en buena medida una actividad manual, que desafía la paciencia, la imaginación y la intuición de programadores.*
- *Muchas veces se requiere incluir en el código fuente instrucciones auxiliares que permitan el seguimiento de la ejecución del programa, presentando los valores de variables y direcciones de memoria y ralentizando la salida de datos ("modo de depuración").*

7.2. DEPURACIÓN DE PROGRAMAS.

El depurador de la mayoría de IDEs permite:

- Ejecutar el código fuente paso a paso.
- Ejecutar métodos del JDK paso a paso.
- Utilizar *breakpoint* (puntos de ruptura) para detener la ejecución del programa y poder observar el estado de las variables.
- Conocer el valor que toma cada variable o expresión según se van ejecutando las líneas de código.
- Modificar el valor de una variable sobre la marcha y continuar la ejecución.

Opciones (*Apache NetBeans*):

Debug Main Project (Ctrl+F5).
Step Over (F8).
Step Into (F7).
Step Out (Ctrl+F7).
Run to Cursor (F4).
Continue (F5) …

* *Breakpoints.*
* *Breakpoints condicionales.*
* *Ventana de variables locales.*
* *Watches.*
* *Pila de llamadas.*
* *…*

7.3. EXCEPCIONES EN JAVA.

7.3.1. CONCEPTO DE EXCEPCIÓN.

El tratamiento de excepciones en Java es un mecanismo del lenguaje que permite gestionar errores y situaciones “excepcionales”.

Una excepción es un error o situación excepcional que se produce durante la ejecución de un programa.

Ejemplos:

Leer un fichero que no existe.

Acceder al valor N de una colección que contiene menos de N elementos.

Enviar/recibir información por red mientras se produce una pérdida de conectividad.

Todas las excepciones en Java se representan a través de objetos que heredan, en última instancia, de la clase *java.lang.Throwable*.

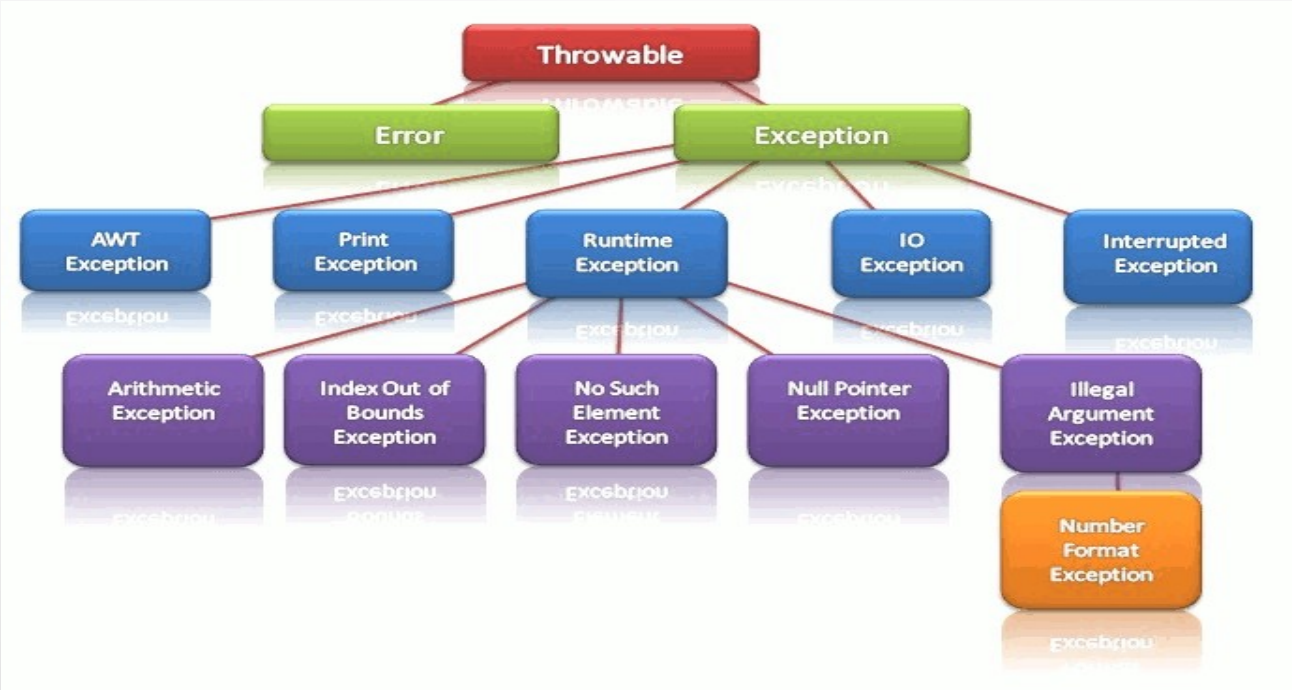
7.3. EXCEPCIONES EN JAVA.

7.3.2. TIPOS DE EXCEPCIONES.

El lenguaje Java diferencia claramente entre 3 tipos de excepciones:

- Errores.
- Comprobadas (o *checked*).
- No comprobadas (o *unchecked*).

El siguiente gráfico muestra el árbol de herencia de clases de excepciones en Java:



7.3. EXCEPCIONES EN JAVA.

7.3.2. TIPOS DE EXCEPCIONES.

La clase principal de la cual heredan todas las excepciones Java es *Throwable*. De ella nacen dos ramas: *Error* y *Exception*.

La primera representa errores de una magnitud tal, que una aplicación nunca debería intentar realizar nada con ellos (como errores del JVM, desbordamientos de buffer, etc.). La segunda rama, encabezada por *Exception*, representa aquellos errores que normalmente sí debemos gestionar.

De *Exception* nacen múltiples ramas: *ClassNotFoundException*, *IOException*, *ParseException*, *SQLException* y otras muchas, todas ellas de tipo *checked*.

La única excepción (valga la redundancia) es *RuntimeException* que es tipo *unchecked* y encabeza todas las de este tipo.

7.3. EXCEPCIONES EN JAVA.

7.3.3. EXCEPCIONES CHECKED.

Las excepciones de tipo “checked” representa un error del cual técnicamente podemos recuperarnos. Por ejemplo, una operación de lectura/escritura en disco puede fallar porque el fichero no exista, porque éste se encuentre bloqueado por otra aplicación, etc.

Todas estas situaciones, además de ser inherentes al propósito del código que las lanza, son totalmente ajenas a nuestro código, y deben ser declaradas y manejadas mediante excepciones de tipo “checked” y sus mecanismos de control.

En ciertos momentos, nuestro código no estará preparado para gestionar la situación de error o simplemente no será su responsabilidad. En estos casos, lo más razonable es relanzar la excepción y confiar en que un método superior en la cadena de llamadas sepa gestionarla.

Por tanto, todas las excepciones de tipo “checked” deben ser capturadas (bloque *try-catch*) o relanzadas (*throws*).

7.3. EXCEPCIONES EN JAVA.

7.3.3. EXCEPCIONES CHECKED.

```java
import java.io.FileWriter;
import java.io.IOException;

public class Test01_excepciones {

    public static void main(String[] args) {
        guardarEnFichero();
    }

    private static void guardarEnFichero() { // CAPTURA DE EXCEPCIÓN
        FileWriter fichero;
        try {
            // Las siguientes dos líneas pueden lanzar una excepción de tipo IOException
            fichero=new FileWriter("datos.txt");
            fichero.write("Datos a guardar en el fichero");
            fichero.close();
        } catch (IOException e) {
            // Aquí capturamos cualquier excepción IOException que se lance (incluidas sus subclases)
            e.printStackTrace();
        }
    }

}
```

7.3. EXCEPCIONES EN JAVA.

7.3.3. EXCEPCIONES CHECKED.

```java
import java.io.FileWriter;
import java.io.IOException;

public class Test02_excepciones {

    public static void main(String[] args) {
        try {
            guardarEnFichero();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static void guardarEnFichero() throws IOException { // LANZAMIENTO DE EXCEPCIÓN
        FileWriter fichero;
        fichero=new FileWriter("datos.txt");
        fichero.write("Datos a guardar en el fichero");
        fichero.close();
    }

}
```

7.3. EXCEPCIONES EN JAVA.

7.3.3. EXCEPCIONES CHECKED.

Cuando se lanza una excepción, forzamos al código cliente de nuestro método a capturarla o relanzarla. Una excepción que sea relanzada una y otra vez, terminará llegando al método "primigenio" y, en caso de no ser capturada por éste, producirá la finalización de su hilo de ejecución.

¿Cuándo capturar una excepción?

Cuando podemos recuperarnos del error y continuar con la ejecución.

Cuando queremos registrar el error.

Cuando queremos relanzar el error con un tipo de excepción distinto.

¿Cuándo relanzar una excepción?

Cuando no es competencia nuestra ningún tratamiento de ningún tipo sobre el error que se ha producido.

7.3. EXCEPCIONES EN JAVA.

7.3.4. EXCEPCIONES UNCHECKED.

Una excepción de tipo “unchecked” representa un error de programación.

Un ejemplo típico, es intentar leer de un array de N elementos un elemento que se encuentra en la posición mayor o igual a N, esto producirá la excepción “unchecked” *ArrayIndexOutOfBoundsException*.

Otro ejemplo, es la división por cero, que producirá la excepción “unchecked” *ArithmeticException*.

Las excepciones tipo “unchecked” no deben ser forzosamente declaradas ni capturadas (es decir, no son comprobadas). Por ello no son necesarios bloques *try-catch* ni declarar formalmente en la firma del método el lanzamiento de excepciones de este tipo (*throws*).

Lo que se debería hacer es controlar el problema por código ya que son consecuencia de errores de programación.

7.3. EXCEPCIONES EN JAVA.

7.3.4. EXCEPCIONES UNCHECKED.

```java
/* EJEMPLO ARITHMETICEXCEPTION */

// Código NO CONTROLADO
int a=10, b=0;
double c;
c=a/b;
System.out.println("Resultado: "+c);


// Código SI CONTROLADO
int a=10, b=0;
double c=0;
try {
    c=a/b;
} catch (ArithmeticException e) {
    System.out.println("Error: "+e.getMessage());
}
System.out.println("Resultado: "+c);
```

7.3. EXCEPCIONES EN JAVA.

7.3.4. EXCEPCIONES UNCHECKED.

```java
/* EJEMPLO ARRAYINDEXOUTOFBOUNDSEXCEPTION */

// Código NO CONTROLADO
int[] a={1,2,3};
int i=3;
System.out.println(a[i]);


// Código SI CONTROLADO
int[] a={1,2,3};
int i=3;
try {
    System.out.println(a[i]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Error: "+e.getMessage());
}
```

7.3. EXCEPCIONES EN JAVA.

7.3.4. EXCEPCIONES UNCHECKED.

```java
/* EJEMPLO NULLPOINTEREXCEPTION */

// Código NO CONTROLADO
Alumno a=null;
System.out.println(a.getNombre());


// Código SI CONTROLADO
Alumno a=null;
try {
    System.out.println(a.getNombre());
} catch (NullPointerException e) {
    System.out.println("Error: "+e.getMessage());
}
```

7.3. EXCEPCIONES EN JAVA.

7.3.5. ESTRUCTURA DE CONTROL DE EXCEPCIONES.

La estructura de control de excepciones es:

```
try {
    ...
} catch () {
    ...
} catch () {
    ...
} finally {
    ...
}
```

La parte *catch* puede repetirse tantas veces como excepciones diferentes se deseen capturar. La parte *finally* es opcional y sólo puede aparecer una vez.

7.3. EXCEPCIONES EN JAVA.

7.3.5. ESTRUCTURA DE CONTROL DE EXCEPCIONES.

Cuando se produce una excepción se compara si coincide con la excepción del primer *catch*. Si no coincide se compara con la excepción del segundo *catch*, y así sucesivamente. Cuando se encuentra un *catch* cuya excepción coincide con la que se ha lanzado, se ejecuta el bloque de sentencias de ese *catch*.

Si ningún bloque *catch* coincide con la excepción lanzada, dicha excepción se lanza fuera de la estructura *try-catch-finally*. Si no se captura en el método, se debe lanzar delegándola. Para ello hay que declararlo en la cabecera del método.

El bloque *finally* se ejecuta tanto si *try* terminó normalmente como si se capturó una excepción en algún bloque *catch*, es decir, se ejecuta siempre. Se suele utilizar para liberar recursos como cerrar ficheros, liberar conexiones de base de datos o de red, …).

También es posible, agrupar varias excepciones en el mismo *catch* con el operador "|".

7.3. EXCEPCIONES EN JAVA.

7.3.5. ESTRUCTURA DE CONTROL DE EXCEPCIONES.

Se produce una delegación de excepciones cuando un método utiliza una sentencia que puede generar una excepción, pero la excepción que se puede generar no la captura y la trata, sino que la delega a quien le llamó. Para ello, simplemente, se declara en la cabecera del método que la excepción que se puede producir durante la ejecución del método, la puede generar dicho método.

Ejemplo:

private void metodo() throws Exception { … }

Java permite crear "artificialmente" excepciones, es decir, provocar nosotros mismos una excepción. De esta manera, el flujo del programa se dirigirá a la instrucción *try-catch* más cercana y/o "superior".

Ejemplo:

throw new Exception();

7.3. EXCEPCIONES EN JAVA.

7.3.5. ESTRUCTURA DE CONTROL DE EXCEPCIONES.

```
public class Test04_excepciones {

    public static void main(String[] args) {

        System.out.println("");

        int a=10, b=0;
        double c=0;
        try {
            c=divide(a,b);
        } catch (ArithmeticException e) {
            System.out.println("Error: "+e.toString());
        } catch (Exception e) {
            System.out.println("Error: "+e.toString());
        }
        System.out.println("Resultado: "+c);

        System.out.println("");
    }

    private static double divide(int a, int b) throws Exception {
        return a/b;
    }

}
```

7.3. EXCEPCIONES EN JAVA.

7.3.5. ESTRUCTURA DE CONTROL DE EXCEPCIONES.

```
public class Test05_excepciones {

    public static void main(String[] args) {

        System.out.println("");
        try {
            metodo1();
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("");
    }

    private static void metodo1() throws Exception {
        try {
            metodo2();
        } catch (Exception e) {
            throw new Exception("Error en metodo1!",e);
        }
    }

    private static void metodo2() throws Exception {
        try {
            metodo3();
        } catch (Exception e) {
            throw new Exception("Error en metodo2!",e);
        }
    }

    private static void metodo3() throws Exception {
        throw new Exception("Error en metodo3!");
    }

}
```

7.3. EXCEPCIONES EN JAVA.

7.3.6. CLASE EXCEPTION.

La clase *Exception* es la superclase de todos los tipos de excepciones. Todas las clases de excepciones heredan, ya sea de forma directa o indirecta, de la clase *Exception*, formando una jerarquía de clases.

La clase *Exception* dispone de varios métodos que nos dan información sobre la excepción producida:

String getMessage()	Muestra una indicación específica del error ocurrido.
String toString()	Muestra la clase de excepción y el texto de *getMessage()*.
void printStackTrace()	Muestra el mismo mensaje que cuando no se controla el error.

7.3. EXCEPCIONES EN JAVA.

7.3.6. CLASE EXCEPTION.

```java
public class Test06_excepciones {

    public static void main(String[] args) {

        System.out.println("");

        Integer a=10;
        Integer b=0;    // Con b=0 daría un tipo de error y b=null da otro diferente!!
        Double c=0.0;
        try {
            c=(double)(a/b);
        } catch (ArithmeticException e) {                   // Excepción más específica
            c=0.0;
            System.out.println("Error: "+e.toString());
        } catch (NullPointerException e) {                  // Excepción más específica
            c=1.0;
            System.out.println("Error: "+e.getMessage());
        } catch (Exception e) {                             // Excepción menos específica (más general)
            e.printStackTrace();
        } finally {
            System.out.println("Resultado: "+c);
        }

        System.out.println("");
    }

}
```

FIN