

Clase adicional 10

Temas

- Excepción
 - Tipos de excepciones
 - Detectar una excepción
 - Detectar varias excepciones
 - Crear una excepción propia
 - Arrojar una excepción
- Flujo
 - Entrada, salida y error tradicionales
 - Flujos de entrada y salida
 - Ejemplo
- Búsqueda binaria
 - Búsqueda lineal
 - Búsqueda binaria
 - Árbol binario
- Problemas de la clase adicional
- Problema de diseño

Excepción

El término excepción es una forma abreviada de "evento excepcional" que se produce durante la ejecución de un programa que interrumpe el flujo normal de instrucciones. Cuando JVM se encuentra con una excepción:

- detiene el procesado del código en el que está trabajando
- crea un tipo concreto de objeto de excepción
- ejecuta la rutina que gestiona dicha excepción

Tipos de excepciones

Existen dos tipos de excepciones en Java: comprobadas y sin comprobar.

Las excepciones comprobadas se producen por algún error en el entorno en el que se desarrolla el código. Por ejemplo, si el código intenta leer información de un archivo que no existe, el programa arrojará una excepción `IOException`. Una excepción comprobada se escapa del control del programador: puede producirse incluso si el código no contiene ni un solo error. Por lo tanto, debe estar preparado para obtener excepciones comprobadas en el código: podrá detectarlas o arrojarlas.

Las excepciones no comprobadas suelen producirse por errores de programación. En este caso, lo más indicado es corregir el programa. Por este motivo, no

se da por hecho que se gestionarán todas en el código (se deben evitar en primer lugar). Por ejemplo, si el código intenta leer el décimo elemento de un array de tamaño 9, el programa arrojará una excepción `ArrayIndexOutOfBoundsException`.

Detectar una excepción

Si cree que determinadas partes del código podrían generar una excepción, encierre dicho código dentro de un bloque `try...catch`. Éste es el formato

```
try
{ código }
catch (XYException e)
{ gestor de la excepción XYException }
finally { ejecutar siempre }
```

- o Si cualquier parte del código dentro del bloque `try` crea una excepción, el programa saltará el resto del código del bloque `try` y ejecutará el código del gestor dentro del bloque `catch`.
- o Si no se produce ninguna excepción, el programa recorrerá todo el código del bloque `try` y saltará el bloque `catch`.
- o Independientemente de lo que ocurra dentro del bloque `try`, el código dentro del bloque `finally` siempre se ejecutará.

¿Qué debe hacer si detecta una excepción? El método más sencillo es imprimirla. Hay tres formas de hacerlo

`getMessage()` devuelve una cadena que describe la excepción que se ha producido

`toString()` devuelve una cadena compuesta por el nombre de la clase de la excepción concreta y el mensaje de error

`printStackTrace()` imprime la secuencia de llamadas al método que han producido la excepción en el flujo de error tradicional

A continuación, mostramos un ejemplo

```
public class T10Exception {
    int addInputs(String[] args) {
        int sum = 0;
        try {
            for (int i=0; i<args.length; i++)
                sum += Integer.parseInt(args[i]);
        }
    }
}
```

```

catch (NumberFormatException e) {

    System.out.println("\nResultado de getMessage() " + e.getMessage());
    System.out.println("Resultado de toString() " + e.toString());
    System.out.println("Resultado de printStackTrace() " + e.printStackTrace());
}
return sum;
}

public static void main(String[] args) {
    T10Exception self = new T10Exception();
    String[] test = {"1", "2", "X"};
    self.addInputs(test);
}
}

```

Éstos son los datos de entrada y salida

```

Salida de getMessage(): X
Salida de toString(): java.lang.NumberFormatException: X
Salida de printStackTrace():
java.lang.NumberFormatException: X
en java.lang.Integer.parseInt(Integer.java:414)
en java.lang.Integer.parseInt(Integer.java:463)
en T10Exception.addInputs(T10Exception.java:6)
en T10Exception.main(T10Exception.java:19)

```

En el ejemplo anterior, cuando el usuario introduce un valor no *integer*, el método `parseInt` arroja una excepción `NumberFormatException` detectada por la cláusula `catch` y el programa imprime el contenido de la excepción en distintos formatos. Observe que `printStackTrace()` no devuelve una cadena. Imprime directamente el mensaje en `stderr`.

Tal como vimos en el material de clase, las excepciones pueden heredar información de unas a otras. Por ejemplo, `FileNotFoundException` amplía `IOException` de tal modo que una cláusula `catch (IOException e)` también detectará la excepción `FileNotFoundException`. Si no está seguro de qué excepción generará el segmento de código, simplemente detecte (`Exception e`) y se detectará cualquier otra excepción.

Detectar varias excepciones

Es posible detectar varios tipos de excepciones en un bloque `try...catch`: simplemente utilice una cláusula `catch` independiente para cada tipo, tal como se muestra a continuación:

```

try { código }

catch (IOException e1)

{ e1.printStackTrace(); }

catch (NumberFormatException e2)

```

```
{ e2.printStackTrace(); }
```

Crear sus propias excepciones

Al igual que ocurre con otras clases de Java, puede ampliar las clases de excepciones existentes y crear sus propios tipos. A continuación puede ver un ejemplo

```
Class MyException extends NumberFormatException {  
    public MyException(String msg) {  
        super("Formato no válido " + msg);  
    }  
}
```

Arrojar una excepción

En el ejemplo anterior, hemos detectado la excepción `NumberFormatException` y la hemos gestionado in situ. Es posible que en algunas ocasiones este comportamiento no sea el más indicado. Por ejemplo, un programador está escribiendo un método para que el resto de las clases lo utilicen y que podría desencadenar una excepción. Tal vez quiera que el usuario de la clase decida cómo gestionar la excepción. En este caso, necesitará "arrojar" la excepción, esto es, delegar la responsabilidad de la gestión en el llamante. Se hace del siguiente modo.

- o Decidir qué excepción causará el método
- o Declarar la cláusula "throws" en el encabezado del método
- o Gestionar la excepción en el llamante (try/catch)

Ésta es la modificación del ejemplo anterior

```
public class T10Exception {  
    int addInputs(String[] args) throws NumberFormatException {  
        int sum = 0;  
        for (int i=0; i<args.length; i++)  
            sum += Integer.parseInt(args[i]);  
        return sum;  
    }  
  
    public static void main(String[] args) {  
  
        T10Exception self = new T10Exception();  
        String[] test = {"1", "2", "X"};  
        try {  
            self.addInputs(test);  
        }  
        catch (NumberFormatException e) {  
            System.out.println("\nResultado de getMessage() " + e.getMessage());  
            System.out.println("Resultado de toString() " + e.toString());  
            System.out.println("Resultado de printStackTrace() ");  
        }  
    }  
}
```

```
        e.printStackTrace();
    }
}
```

En este caso, el método `addInputs` decide no gestionar la excepción. En su lugar, la arroja al llamante. Por tanto, el método `main` debe detectarla y gestionarla. El resultado será el mismo.

Observe que un método puede declarar excepciones no comprobadas que él mismo arroja, pero DEBE declarar las excepciones comprobadas o, de lo contrario, el compilador se quejará. `NumberFormatException` es un buen ejemplo de una excepción sin comprobar que QUEREMOS comprobar. `IOException` o `FileNotFoundException` son un buen ejemplo de excepciones comprobadas que se deben declarar.

Flujos

Hasta ahora, nuestros programas han recibido información introducida por el usuario a través del teclado, por ejemplo, `getText()` de `JTextField`. En esta sección, le mostraremos cómo utilizar los flujos para introducir datos y obtener información en otras fuentes, como un archivo o una conexión de red.

Entrada, salida y error tradicionales

En realidad, ya ha utilizado los flujos para enviar datos a la pantalla desde el principio del curso

- `System.out.println(String)` imprime una cadena en el flujo de salida tradicional que, normalmente (aunque no obligatoriamente), es la "pantalla".
- Otro objeto de flujo de salida tradicional es el objeto `System.err`. Este objeto permite que un programa emita mensajes de error. En este caso, de nuevo el resultado suele dirigirse por defecto a la pantalla.

`System.out` y `System.err`, así como `System.in` (que no hemos utilizado demasiado) se crean automáticamente al ejecutarse un programa de Java. Estos objetos podrían ser suficiente si sólo quiere escribir en la pantalla o leer desde el teclado.

Flujos

La biblioteca `java.io` permite introducir y obtener datos de otras fuentes de datos, como discos, conductos interprocesales o conexiones de red. Esto se logra gracias a los flujos. Java proporciona cuatro tipos de flujos

- `InputStream` es un objeto desde el que se lee una secuencia de datos binarios
- `OutputStream` es un objeto en el que se escribe una secuencia de datos binarios
- `Reader` es un objeto en el que se lee una secuencia de texto
- `Writer` es un objeto en el que se escribe una secuencia de texto

Cada flujo tiene un número determinado de subclases y cada una de ellas gestiona un tipo de fuente de datos. Por ejemplo:

- o InputStream
 - o FileInputStream
 - o ObjectInputStream
 - o PipedInputStream
- o OutputStream
 - o FileOutputStream
 - o ObejctOutputStream
 - o PipedOutputStream

Una vez conectado un InputStream a una fuente de datos, puede utilizar su función `read()` para leer los datos de dicha fuente. Sin embargo, la función `read()` es bastante limitada: solamente puede leer arrays de bytes. La mayor parte del tiempo, necesitará añadir un "filtro" para convertir los bytes en tipos de datos más útiles. Entre los ejemplos de flujos de entrada de filtros se encuentran `DataInputStream`, `BufferedInputStream`, etc.

En resumen, para leer datos de una fuente de datos, necesita seguir estos procedimientos:

1. Identificar la fuente de datos (¿qué es? ¿qué tipos de datos contiene? etc.)
2. Conectar un flujo de entrada adecuado a dicha fuente de datos (por ejemplo, `FileInputStream` o `ObjectInputStream`)
3. Asociar un flujo del filtro a dicho flujo de entrada (p.ej., `DataInputStream`)
4. Leer datos utilizando métodos proporcionados por el flujo de entrada del filtro (por ejemplo, `readInt()`).

Esto mismo ocurre para `OutputStreams`. A continuación, le guiaremos en un ejemplo de entrada y salida que actualiza los registros de empleados basándose en el número de horas trabajadas en el mes. Éstos son los requisitos:

Ejemplo

Tenemos dos archivos. El primero, `Employee_May.dat`, contiene 5 registros de empleados con el formato siguiente

Nombre, NSS, tarifa por hora, salario hasta la fecha

Éste es el contenido del archivo

Wen Xiao, 555-12-3456, 65, 20000

Anil Gupta, 555-22-2222, 70, 30000

Pei-Lei Fan, 555-33-4444, 60, 150000

Katie Chase, 555-44-5555, 80, 40000

El segundo archivo, Hours.dat, contiene 5 enteros, que son el número de horas que cada empleado ha trabajado ese mes. Estos enteros tienen la misma secuencia que los registros de los empleados. Éste es el contenido del archivo

```
50 60 40 50 70
```

Nuestro programa lee el número de horas trabajadas del archivo Hours.dat, calcula el salario mensual del empleado, lo actualiza e imprime los nuevos datos en un archivo llamado Employee_June.dat

Observe que, por claridad, hemos dividido el código en bloques relativamente independientes. De hecho, existen muchas más formas eficaces de crear este programa.

Importar encabezados

```
import java.io.*;
import java.util.*;
```

Leer el archivo de datos (Hours.dat)

1. Crear un objeto File que represente Hours.dat
2. Conectar el objeto File a un flujo de entrada (FileInputStream)
3. Asociar un flujo de filtro (DataInputStream) al flujo de entrada
4. Leer 5 enteros del flujo de entrada de datos
5. Cerrar el flujo de entrada

```
File f = new File("Hours.dat");
FileInputStream fin= new FileInputStream(f);
DataInputStream din = new DataInputStream(fin);
int[] hours = new int[10];
for (int i=0; i<5; i++)
    hours[i] = din.readInt();
din.close();
```

Leer el archivo de texto (Employee_May.dat)

6. Conectar Employee_May.dat a un FileReader
7. Asociar un BufferedReader al FileReader
8. Leer 5 cadenas del BufferedReader
9. Cerrar el BufferedReader

Aquí hemos introducido una nueva clase llamada `BufferedReader`. En pocas palabras, el *buffering* es una técnica que aumenta la eficacia de entrada y salida. En vez de leer o escribir inmediatamente en el disco cada vez que se solicita la operación, el objeto de flujo utilizará el *buffer* siempre que pueda. Por ejemplo, si estuviera escribiendo una serie de 256 enteros en un archivo sin *buffering*, cada vez que emitiera un comando `writeln(int)`, el sistema escribiría en el disco. Con el *buffering*, sólo se escribiría en el disco cuando el *buffer* estuviera lleno. `BufferedReader` también ofrece la función `readLine()`, que permite al usuario leer líneas en vez de caracteres.

```
FileReader fr = new FileReader("Employee_May.dat");
BufferedReader in = new BufferedReader(fr);
String[] records = new String[5];
for (int j=0; j<5; j++)
    records[j] = in.readLine();
in.close();
```

Buscar los datos

La función `readLine()` devuelve una cadena que contiene los 4 campos del registro de un empleado. Necesitamos conocer la tarifa por hora y el salario hasta la fecha.

10. Asignar la cadena a un `StringTokenizer`

11. Buscar el tercer y el cuarto *token* en la cadena

12. Calcular el salario de este mes y sumarlo al salario hasta la fecha

En este proceso, utilizamos una clase útil denominada `StringTokenizer` que divide la cadena en cuatro fragmentos (*token*) independientes basados en el delimitador. En nuestro ejemplo, el delimitador es `","`

```
StringTokenizer st;
String name, ssn;
double hourlyRate, salary;
for(int k=0; k<5; k++) {

    st = new StringTokenizer(records[k], ",");
    name = st.nextToken(); ssn = st.nextToken();
    hourlyRate = Double.parseDouble(st.nextToken());
    salary = Double.parseDouble(st.nextToken());
    salary += hourlyRate * hours[k];
    records[k] = name + ", " + ssn + ", " + hourlyRate + ", " + salary
}
}
```

Tenga en cuenta que en las llamadas a `parseDouble()`, es posible que quiera buscar la excepción `NumberFormatException`

Mostrar los datos

13. Crear un `FileWriter` con el nombre de archivo `Employee_June.dat`

14. Asociar un `PrintWriter` al `FileWriter`

15. Escribir el array de la cadena en el `PrintWriter`

16. Cerrar el `PrintWriter`

```
FileWriter fw = new FileWriter("Employee_June.dat");
PrintWriter out = new PrintWriter(fw);
for (int i=0; i<5; i++)
    out.println(records[i]);
out.close();
```

Búsqueda binaria

El objetivo de la búsqueda es encontrar un registro concreto dentro de una serie. Existen varios enfoques con distinta eficacia.

Búsqueda lineal

La búsqueda lineal es el proceso secuencial por el que se analizan los registros, comenzando por el primero hasta llegar, o bien a una coincidencia, o al final de la búsqueda sin obtener resultados satisfactorios. La búsqueda lineal es útil cuando los registros se distribuyen aleatoriamente o si los datos se almacenan en una lista enlazada lineal. Este enfoque puede resultar razonable si el tamaño de la serie es pequeño y si el contenido es altamente dinámico (por ejemplo, eliminaciones frecuentes).

Búsqueda binaria

La búsqueda binaria es un procedimiento sencillo por el cual se buscan elementos en un array ya ordenado. Se basa en la estrategia "divide y vencerás" que utilizamos en el método de la bisección para el cálculo de raíces. Su funcionamiento se basa en la división del conjunto de datos por la mitad, determinando en qué mitad se encuentra el elemento deseado y, a continuación, cortándolo por la mitad otra vez, etc. Como decíamos, la búsqueda binaria se basa en la estrategia "divide y vencerás". Es mucho más eficaz que la búsqueda lineal, ya que exige un menor número de iteraciones. Sin embargo, antes de utilizar la búsqueda binaria, debemos ordenar los datos.

Realizar búsquedas con un árbol de búsqueda binaria

Un árbol de búsqueda binaria es un árbol binario en el que la descendencia situada a la izquierda de cualquier nodo es "más pequeño" que el nodo raíz y la descendencia situada a la derecha es "más grande" que el nodo raíz. Para buscar un elemento en un árbol binario, lleve a cabo las acciones siguientes:

Comparar la clave de búsqueda con la clave raíz.

Si es igual, la búsqueda ha terminado.

Si no, determinar si es más pequeña que la clave raíz.

Si es así, debe encontrarse en el subárbol izquierdo.

Si es más grande, debe encontrarse en el subárbol derecho.

Por lo tanto, en cada ciclo aproximadamente se descarta la mitad de la serie. Al final, el procedimiento encuentra la clave en el árbol o llega a un nodo NULL y llega a la conclusión de que la clave no se encuentra en el árbol.

Problemas de la clase adicional

1. Excepción

En el boletín de problemas 7, entramos en la clase `ArrayStack` cuyo método `pop()` arrojaba una excepción.

```
public Object pop() throws EmptyStackException {
    if (isEmpty())
        throw new EmptyStackException();
    else
        return stack[top--];
}
```

¿Se trata de una excepción comprobada o sin comprobar?

¿Por qué decidimos arrojar la excepción en vez de gestionarla dentro del método `pop`?

Modifique el programa del boletín de problemas 7 para que gestione la excepción

Problema de diseño

Tal como hemos mencionado en esta clase adicional, un flujo puede ser un archivo o una conexión de red. En este ejercicio, escriba un breve programa que lea un archivo html de un sitio Web.

Sugerencia: utilice la clase `URL` y no `File`