

Clase 28

Excepciones y clases internas

Objetivos

Profundizaremos en dos funciones avanzadas de Java:

- **Excepciones ofrecidas por el mecanismo de gestión de errores de Java.**
- **Clases internas para aligerar la sobrecarga de las subclases y la implementación de interfaces. Hacen que utilizar el modelo de eventos de Java sea mucho más agradable.**

Arrojar una excepción

```
public static double average( double [] dArray )
    throws IllegalArgumentException
{
    if ( dArray.length == 0 )
        throw new IllegalArgumentException();
    else
    {
        double sum = 0.0;
        for ( int i = 0; i < dArray.length; i++ )
            sum += dArray[ i ];
        return sum / dArray.length;
    }
}
```

3

Declarar una excepción

- Si un método puede arrojar una excepción, siempre es posible declarar el tipo de la excepción en el encabezado, detrás de la palabra clave `throws`

```
public static double average( double [] dArray )
    throws IllegalArgumentException
```

- El compilador requiere la declaración la posibilidad de arrojar la excepción si la clase de la excepción no proviene de `RuntimeException` o `Error`. Éstas reciben el nombre de excepciones comprobadas (comprobadas por el compilador).
- Las excepciones que provienen de `RuntimeException` o `Error` reciben el nombre de excepciones no comprobadas y su declaración es opcional.
- `IllegalArgumentException` realmente no está comprobada.

4

Excepciones comprobadas y no comprobadas

- La diferencia entre las excepciones comprobadas y no comprobadas es bastante arbitraria.
- En principio, las excepciones comprobadas son aquellas que el programador debe ser capaz de solucionar en tiempo de ejecución, como por ejemplo una `FileNotFoundException`. A menudo, éstas son generadas en el código del sistema por el usuario y no se trata de un error del programador.
- Por su parte, se supone que las excepciones no comprobadas pueden ocurrir en cualquier lugar (por eso es difícil comprobarlas) y pueden ser el resultado de un error del programador (por lo que la mejor forma de tratarlas es arreglar el programa). Buenos ejemplos: `NullPointerException`, `ArrayIndexOutOfBoundsException`. Mal ejemplo: `NumberFormatException`, arrojada, por ejemplo, por `Integer.parseInt(String s)`.

5

Creación de instancias de excepciones

- Las excepciones son objetos. Debe crear una nueva instancia de una excepción para poder arrojlarla

```
if ( dArray.length == 0 )
    throw new IllegalArgumentException();
```
- Las excepciones pueden ser arbitrariamente complejas, pero las que se incluyen en el JDK disponen de un constructor que toma una cadena de mensajes de error única.

6

DetECCIÓN DE EXCEPCIONES

- La ejecución normal de un método cesa en el momento en que se arroja una excepción.
- En ese momento, el entorno del tiempo de ejecución busca un bloque `try` de cierre con una cláusula `catch` coincidente.
- Tras ejecutar la cláusula `catch`, el programa reanuda la acción con la primera sentencia después del bloque `catch`.

7

Patrón throw/try/catch

```
try
{
    if ( error )
        throw new MyException();
    //salta la siguiente ejecución
}
catch ( MyException e )
{
    // trata la excepción e
}
//reanuda la ejecución
```

8

Detección de excepciones, 2

- Si no hay ningún bloque `try` de cierre en el método actual, o si hay uno con una cláusula `catch` correctamente insertada, la máquina virtual de Java busca un par `try/catch` coincidente en la pila de llamada.
- Si la JVM no puede encontrar dicho par en la pila de llamada, el comportamiento predeterminado es imprimir un mensaje de error y detener el hilo (y, a menudo, todo el programa).

9

Detectar excepciones en la pila de llamada

```
double [] myDoubles = {... };
double a = 0.0;
try {
    a = average( myDoubles );
}
catch ( IllegalArgumentException e )
{
    // hacer algo con ello
}
System.out.println( "Media = " + a );
```

10

No detectar excepciones en la pila de llamada

```
import javax.swing.*;

public class BadArgument {
    public static void main( String [] args ) {
        while ( true ) {
            String answer = JOptionPane.showInputDialog(
                "Escriba un entero" );
            int intAnswer = Integer.parseInt( answer );
            // ¿Qué ocurre si el usuario escribe %!Z$?
            if ( intAnswer == 42 )
                break;
        }
        System.exit( 0 );
    }
}
```

Escribir sus propias clases de excepciones

- Escribir su propia clase de excepción es sencillo.
- Las nuevas clases de excepciones le permiten tratar un nuevo tipo de error por separado.
- Las clases de excepciones amplían `java.lang.Exception`.

```
public class DataFormatException
    extends java.lang.Exception {
    public DataFormatException()
        { super(); }
    public DataFormatException(String s)
        { super( s ); }
}
```

Excepciones y tratamiento de errores

- La filosofía del tratamiento de errores comienza con asumir que el lugar en que se descubre un error casi nunca coincide con el lugar en que se puede arreglar.
- Lenguajes más antiguos, como C, utilizan el horrible sistema de códigos de error.
- C++ introdujo las excepciones en la familia C.
- Pero son muchos los programadores que, sencillamente, no realizan comprobaciones para buscar errores.

13

La estrategia de las excepciones

El objetivo de utilizar excepciones es separar el problema de **detectar** errores y el problema de **tratarlos**.

14

Excepciones en Calculator

En CalculatorController:doOp()

```
try { ...
    switch( eT )
    {
        case Calculator.I_DIV:
            setAcc( model.div() );
            break;
        ...
    }
catch ( EmptyStackException e )
{ error(); }
```

15

Excepciones en Calculator, 2

¿Dónde se arroja la excepción? No en CalculatorModel.

```
public double div()
    throws EmptyStackException
{
    double bot = pop();
    double top = pop();
    return top / bot;
}
```

16

Excepciones en Calculator, 3

```
CalculatorController:  
public double doOp()  
ejecuta y coloca FSM en  
estado de ERROR
```

```
CalculatorModel:  
public double div()  
¡no lo detecta!
```

```
ArrayStack:  
public Object pop()  
arroja  
EmptyStackException
```

17

Excepciones en Calculator, 4

- A pesar de que `EmptyStackException` es una excepción sin comprobar, está causada aquí por un error del usuario (no ha introducido suficientes operandos), no del programador.
- `ArrayStack` no sabe cómo arreglarlo, por lo que lo arroja.
- `CalculatorModel` no sabe cómo arreglarlo, por lo que lo arroja.
- Finalmente, `CalculatorController` puede hacer algo al respecto, por lo que detecta la excepción y pone todo el método `Calculator` en estado de `Error` hasta que el usuario lo borre.

18

Excepciones y herencia

- Dado que las excepciones son instancias de clases, las clases pueden utilizar la herencia. Una excepción `FileNotFoundException` es una clase proveniente de `IOException`.
- Cuando se detecta un error, debe crear y arrojar una nueva instancia de un tipo adecuado de excepción.
- Se buscará en la pila la sentencia de detección más cercana que coincida con la clase de la excepción o con una de sus **superclases**.

19

Clases anidadas estáticas

Ahora pasemos a las clases *internas* (y *anidadas*).
Puede definir una *clase anidada estática*
(`static`) dentro de otra clase:

```
public abstract class java.awt.geom.Line2D
{
    public static class Double { ... }
    public static class Float { ... }
}
```

20

Clases anidadas estáticas, 2

- Se comportan como cualquier otra clase de primer nivel, salvo en que su nombre verdadero es el nombre de la clase externa concatenada con el nombre de la clase interna: p.ej., `Line2D.Double`
- Una clase anidada se considera parte de la clase que la encierra:
 - Conviértala en `public` si quiere que métodos de otras clases la utilicen
 - Conviértala en `private` si sólo la va a utilizar en la clase que la encierra

21

Clase anidada `private static`

```
public class SLinkedList implements List {
    private int length = 0;
    private SLink first = null;
    private SLink last = null;

    private static class SLink {
        Object item; SLink next;

        SLink( Object o, SLink n )
        { item = o; next = n; }

        SLink( Object o )
        { this( o, null ); }
    }
}
```

22

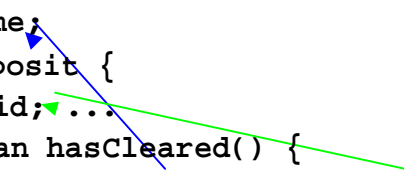
Clases internas

- Si una clase anidada no es estática, podemos llamarla *clase interna*.
- Las instancias de las clases internas se suelen crear utilizando `new` en un método de instancia de la clase que lo encierra.
- Los métodos de las clases internas tienen acceso a las variables y métodos de instancia de la instancia de la clase que los encierra.
- Las clases internas pueden tener constructores con o sin argumentos.

23

Ejemplo de clase interna

```
public class BankAccount { ...
    private Bank home;
    public class Deposit {
        private int id; ...
        public boolean hasCleared() {
            Transaction t = home.getTrans(id);
            return t.cleared();
        }
    }
}
```

A blue arrow points from the `home` field in the `Deposit` class back to the `home` field in the `BankAccount` class. A green arrow points from the `id` field in the `Deposit` class to the `id` parameter in the `hasCleared()` method.

24

Clases internas anónimas

- **Son clases internas “asequibles” en el sentido de que son fáciles de definir**
 - No tienen nombre, por lo que puede crear una única instancia en el lugar que la defina.
 - Deben ampliar una clase o implementar una interfaz.
 - No pueden tener un constructor definido y, por tanto, siempre utilizan uno predeterminado.
- **Se suelen utilizar en código Swing para implementar interfaces de escucha o adaptadores.**

25

Ejemplo de interfaz de escucha

```
public class AnonExample extends JFrame
{
    private JLabel countLabel;
    private int count = 0;

    public AnonExample()
    {
        JPanel myPanel = new JPanel();
        JButton myB = new JButton( "Incremento" );
        myPanel.add( myB );
        countLabel = new JLabel( "0" );
        myPanel.add( countLabel );
    }
}
```

26

Ejemplo de interfaz de escucha, 2

```
myB.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    { countLabel.setText(
        String.valueOf( ++count ));
    }
}
);
getContentPane().add( myPanel,
    BorderLayout.CENTER );
}
```

Variables de la clase que encierra

27

Ejemplo de interfaz de escucha sin clase anónima

```
public class MyAction implements ActionListener
{
    private int count;
    private JLabel myL;
    public MyAction( JButton b, JLabel l )
    {
        count = 0; myL = l;
        b.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    { l.setText( String.valueOf( count++ )); }
}
```

28

Adaptadores

- Algunas interfaces de escucha tienen muchos métodos y es posible que tal vez sólo esté interesado en utilizar uno. Para la mayoría de interfaces `Listener` con varios métodos, existe un adaptador (`Adapter`) correspondiente: una clase con implementaciones nulas de los métodos de escucha.
- Patrón: crear una clase interna anónima que amplíe el adaptador y anule el único método que le interesa.

29

Ejemplo de adaptador

```
package java.awt.event;
public interface MouseMotionListener extends
    ActionListener {
    public void mouseDragged(MouseEvent e);
    public void mouseMoved(MouseEvent e);
}

public abstract class MouseMotionAdapter implements
    MouseMotionListener {
    public void mouseDragged(MouseEvent e) {}
    public void mouseMoved(MouseEvent e) {}
}
```

30

Ejemplo de adaptador, 2

```
import java.awt.event.*; import javax.swing.*;
public class MouseMotion extends JFrame {
    public MouseMotion() {
        setSize( 600, 400 );
        setDefaultCloseOperation( EXIT_ON_CLOSE );
        getContentPane().addMouseListener(
            new MouseMotionAdapter() {
                public void mouseDragged( MouseEvent e ) {
                    System.err.println( "Arrastrado: " + e.getX() +
                        ", " + e.getY() );
                }
            }
        );
    }
}
```

31