

Clase 24

**Caso práctico:
Creación de una calculadora
simple con notación postfijo**

Objetivos

- **Construir una calculadora que sirva para demostrar cómo dividir la funcionalidad en clases significativas**
- **Explicar dos *modelos* de software útiles:**
 - **Modelo/Vista/Controlador (MVC)**
 - **Máquina de estados finitos (MEF)**

Notación infijo y notación postfijo

Las primeras calculadoras científicas utilizaban, en su mayoría, una notación de expresiones conocida como *postfijo* en vez de la notación *infijo*, a la que estamos más habituados

Infijo: $(13.5 - 1.5) / (3.4 + 0.6)$

Postfijo: 13.5 1.5 - 3.4 0.6 + /

Notación infijo y notación postfijo (II)

- La notación postfijo prescinde de los paréntesis que en la notación infijo son necesarios para indicar la precedencia y el orden de evaluación
- La notación postfijo se lee de izquierda a derecha, y evalúa los operadores según vayan apareciendo
- Si eliminamos los paréntesis del modelo infijo, obtenemos una expresión ambigua:

$$13.5 - 1.5 / 3.4 + 0.6$$

¿Sería

$$(13.5 - 1.5) / (3.4 + 0.6) \text{ ó } 13.5 - (1.5 / 3.4) + 0.6?$$

La notación postfijo en pilas

- La mejor manera de interpretar la notación postfijo es utilizándola con pilas
- Cuando el siguiente elemento es un número, se añade a la pila
- Cuando el siguiente elemento es un operador:
 - el número de operandos correspondientes al operador salen de la pila,
 - se aplica el operador, y
 - se inserta en la pila el resultado obtenido

Evaluación de la notación postfijo

Infijo: $(13.5 - 1.5)/(3.4 + 0.6)$

Postfijo: 13.5 1.5 - 3.4 0.6 + /

Elemento	Pila
13.5	13.5
1.5	1.5 13.5
-	12.0
3.4	3.4 12.0
0.6	0.6 3.4 12.0
+	4.0 12.0
/	3.0

Modelos de software

- **Son diseños de soluciones a problemas recurrentes de software**
- **Estos modelos son independientes del lenguaje de implementación**
- **Son más generales y flexibles que los algoritmos y las estructuras de datos**

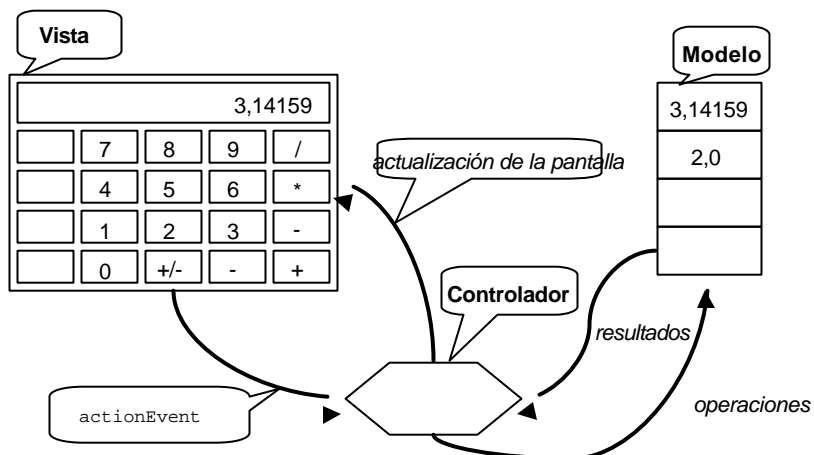
Modelo/Vista/Controlador (MVC)

- **Se trata de un modelo muy útil para el diseño de programas interactivos con interfaz gráfica de usuario (GUI)**
- **Su uso se remonta a la aparición de Smalltalk, uno de los primeros lenguajes orientados a objetos**
- **MVC ha influido decisivamente en el modelo de eventos de Java:**
 - **Las fuentes de eventos forman parte de la vista**
 - **Los oyentes de eventos son parte del controlador**

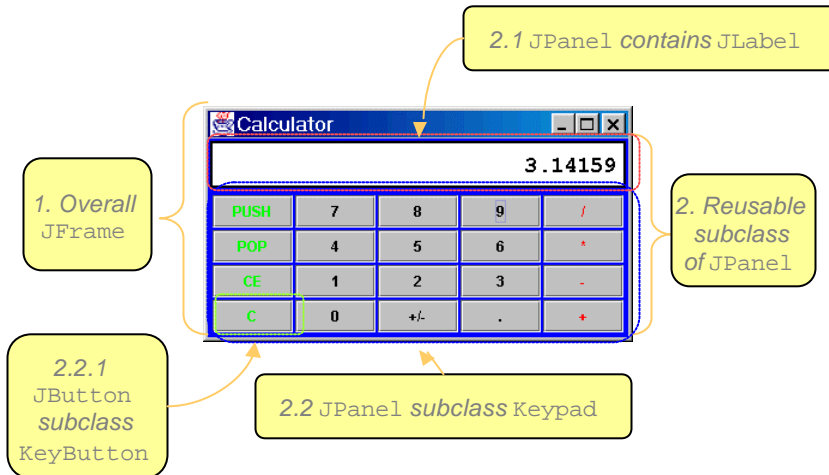
MVC aplicado a nuestra calculadora

- La *vista* es la clase (Calculator) que contiene la interfaz de usuario, el teclado y el monitor
- El *controlador* es la clase (Calculator Controller) que interactúa con el usuario, que procesa los eventos y actualiza la calculadora sin preocuparse por los detalles del monitor
- El *modelo* es la clase (CalculatorModel) que, por así decirlo, gestiona los cálculos, la CPU y la memoria de la calculadora
- La aplicación (CalculatorApp), como siempre, reúne estos elementos mediante un `main()`.

MVC en la calculadora



GUI de la calculadora



Calculadora, 1

```
public class Calculator extends javax.swing.JPanel
    implements ActionListener
{
    class Keypad extends JPanel { . . . }
    class KeypButton extends JButton { . . . }

    static final String I_CE = "CE";
    static final String C_CE = "-1";
    static final int    I_CE = -1;
    . . .
    public static boolean isDigit( int eT )
    { return( eT >= 0 && eT <= 9 ); }

    public static boolean isOp( int eT )
    { return( eT <= I_PUSH && eT >= I_ADD ); }
```

Calculadora, clase JButton

```
class JButton extends JButton {
    JButton( String l, ActionListener a ) {
        super( l );
        setBackground( Color.lightGray );
        addActionListener( a );
    }
    JButton( String l, Color f, ActionListener a ) {
        this( l, a );
        setForeground( f );
    }
    JButton( String l, String c, Color f,
            ActionListener a ) {
        this( l, a );
        setForeground( f );
        setActionCommand( c );
    }
}
```

Calculadora, clase Keypad

```
class Keypad extends JPanel {
    Keypad() {
        setLayout( new GridLayout( 0, 5, 2, 2 ));
        . . .
        add( new JButton( L_CE, C_CE, Color.green,
                        Calculator.this ));
        add( new JButton( "1", Color.black,
                        Calculator.this ));
        . . .
    }
}
```

Calculadora, 2

```
private JLabel display = null;

public void setDisplay( String s )
{ display.setText( s ); }

public void clearDisplay()
{ display.setText( EMPTYSTR ); }

public void actionPerformed((ActionEvent e) {
    if ( controller != null )
        controller.actionPerformed( e );
}
```

CalculatorController: *el controlador*

- La tarea de esta clase (CalculatorController) es la más difícil, por ser la más compleja
- Su función consiste en interpretar las órdenes que da el usuario al pulsar los botones, actualizar la pantalla y usar la clase *modelo* para efectuar todos los cálculos
- Es la clase que conecta las otras partes del modelo. Conoce las otras dos clases, *vista* y *modelo*, mientras que la vista únicamente conoce el controlador y el modelo, aunque es utilizado por el controlador, nunca llama a éste

Controlador y Modelo

- ¿Lo que muestra la calculadora es una copia de la parte superior de la pila?
- ¿Qué hay en la pila? `Doubles`?
- Al introducir una cifra en la calculadora, ¿nos interesa considerar los valores intermedios como dobles o como caracteres?
- Para evitar errores de redondeo, trataremos la visualización como un `String` (en realidad, un `StringBuffer`) que debe insertarse en la parte superior de la pila como un `Double`.

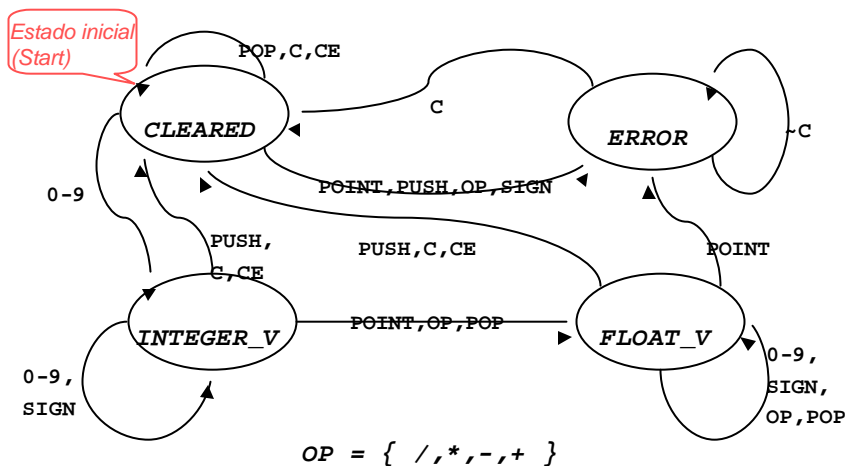
Controlador y Estado (*State*)

- Otro inconveniente del controlador es que su comportamiento depende de las acciones ya realizadas
- Por ejemplo:
 - si el usuario, al introducir un nuevo número, comienza marcando el 3, el controlador asumirá que el usuario va a introducir un número entero.
 - Pero si a continuación el usuario marca ., el controlador deberá interpretarlo como un número de coma flotante.
 - Y si el usuario marca otra vez ., dará error, puesto que el controlador ha detectado ya un punto, que indica decimal
- El comportamiento del controlador depende del *estado previo*

Máquinas de estado finito (FSM)

- El mejor modo de tratar este tipo de situaciones suele ser utilizando otro patrón de diseño conocido como *máquina de estado finito* o *autómata de estado finito*.
- Una FSM es un objeto capaz de ocupar cualquier estado dentro de un número finito de ellos.
- Uno de estos estados, llamado estado inicial (*start*) indica el estado de una instancia recién creada de la FSM
- La FSM acepta *tokens* de entrada o eventos, uno cada vez
- Especifica una transición a un nuevo estado para cada posible estado y para cada *token* de entrada
- Esta transición, a su vez, suele especificar una acción, que es lo que hace útil a la FSM

La FSM CalculatorController



FSM: funcionamiento

Vamos a seguir el ejemplo:

3.14 PUSH 2 *

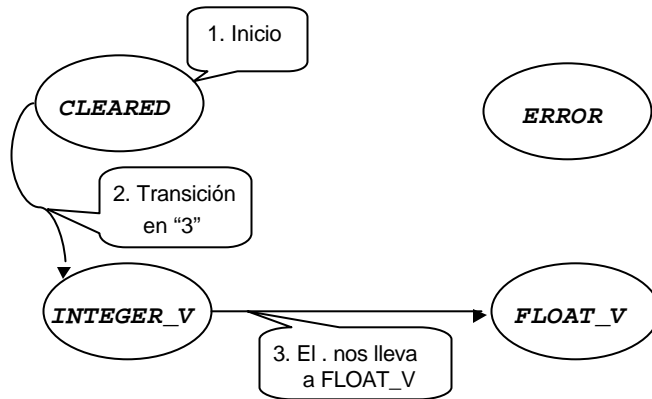
(o, en notación infijo, $3.14 * 2$), en la FSM:

1. Partimos del estado CLEARED (estado de inicio)
2. Al marcar 3 pasamos al estado INTEGER_V
3. . nos lleva al estado FLOAT_V
4. 1 y 4 se añaden al número en el acumulador, permaneciendo en estado FLOAT_V
5. PUSH nos lleva a CLEARED y añade el valor 3.14 a la parte superior de la pila del modelo

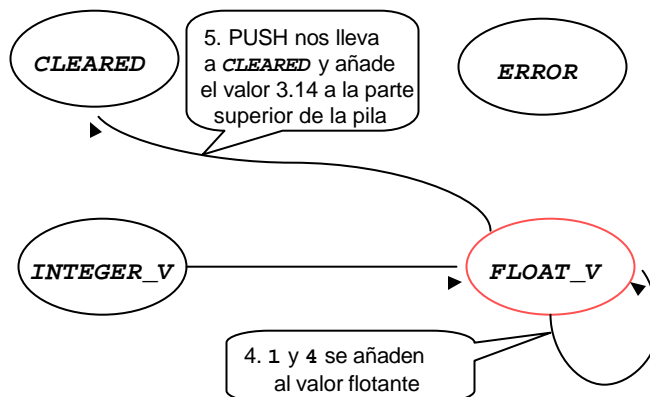
FSM: funcionamiento, 2

6. 2 nos lleva a INTEGER_V e inicia un nuevo número en el acumulador
7. * es una operación, que implícitamente añade 2 a la pila del modelo. También nos lleva momentáneamente a CLEARED y hace que el controlador invoque el método mul() del modelo, obteniendo como resultado 6.28. Este resultado se envía al acumulador y nos devuelve a FLOAT_V.

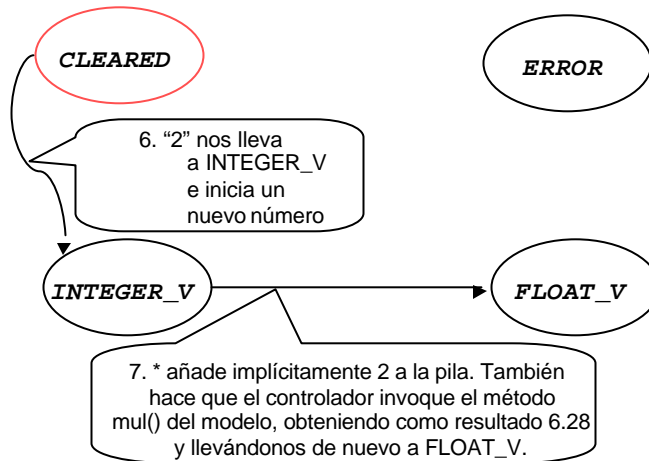
FSM: funcionamiento, 3



FSM: funcionamiento, 4



FSM: funcionamiento, 5



CalculatorController, 1

```
public class CalculatorController implements
    ActionListener
{
    private Calculator calculator;
    private CalculatorModel model;
    private StringBuffer acc;
    private int state = CLEARED;
    private boolean neg = false;

    private final int CLEARED = 0;
    private final int INTEGER_V = 1;
    private final int FLOAT_V = 2;
    private final int ERROR = 3;
```

CalculatorController, 2

```
public void actionPerformed((ActionEvent e) {
    String eStr = e.getActionCommand();
    int eType = Integer.parseInt( eStr );

    if ( state == ERROR ) {
        // La única forma de salvar el error es un CLEAR general
        if ( eType == Calculator.I_C ) {
            reset();
        } else
            return;
    }
}
```

CalculatorController, 3

```
if ( eType == Calculator.I_CE ) clearEntry();
else if ( eType == Calculator.I_C ) reset();
else {
    switch ( state ) {
        case CLEARED:
            if ( Calculator.isDigit( eType ) ) {
                state = INTEGER_V;
                acc.append( eStr );
            } else if ( Calculator.isOp( eType ) )
                doOp( eType );
            else if ( eType == Calculator.I_POINT ) {
                state = FLOAT_V;
                acc.append ( "0." );
            } else
                error();
            break;
        // . . . casos INTEGER_V, FLOAT_V
    }
}
```

La sentencia switch

```
switch ( int expresión ) {  
    case int constante:  
        sentencias;  
        break;  
    ...  
    case int constante:  
        sentencias;  
        // omisión de break: peligrosa  
default:  
    sentencias;  
}
```

CalculatorController, doOp()

```
private void doOp( int eT ) {  
    . . .  
    try {  
        if ( state != CLEARED )  
            doPush();  
        switch( eT ) {  
            case Calculator.I_DIV:  
                setAcc( model.div() ); break;  
            case Calculator.I_MUL: . . .  
            case Calculator.I_SUB: . . .  
            case Calculator.I_ADD: . . .  
        }  
    } catch ( EmptyStackException e )  
    { error(); }
```

CalculatorModel: ideas generales

Comparada con CalculatorController

la clase `CalculatorModel` parece sencilla, pese a que es en la que se efectúan los cálculos

- Cada método de operación matemática, `div()`, `mul()`, `sub()`, y `add()`, elige cuidadosamente el orden correcto de los argumentos que se retiran de la pila
- Tanto estos métodos como `CalculatorModel.pop()` pueden lanzar una excepción `EmptyStackException`, ya que cada uno de ellos invoca `Stack.pop()`.
- Los métodos no captan la excepción, por lo que ésta es devuelta a la rutina que realizó la llamada, del modo que veremos en próximas clases.

Código CalculatorModel

```
public class CalculatorModel {
    private Stack stack = new ArrayStack();

    public void push( double d )
    { stack.push( new Double( d ) ); }

    public double pop() throws EmptyStackException
    { return ( ( Double ) stack.pop()).doubleValue(); }

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