

# Introduction to Computation and Problem Solving

## Class 24: Case Study: Building a Simple Postfix Calculator

Prof. Steven R. Lerman  
and  
Dr. V. Judson Harward

1

## Goals

- To build a calculator to show how to divide functionality up into meaningful classes
- To introduce two useful software *patterns*:
  - Model/View/Controller
  - Finite State Machine

2

## Infix vs Postfix Notation

Most early scientific calculators used an expression notation called *postfix* rather than the *infix* notation that we are used to.

**Infix:**  $(13.5 - 1.5) / (3.4 + 0.6)$

**Postfix:** 13.5 1.5 - 3.4 0.6 + /

3

## Infix vs Postfix, 2

- Postfix notation gets rid of the parentheses that are necessary to determine precedence and evaluation order in infix notation.
- Postfix is scanned left to right. Operators are evaluated as soon as they are encountered.
- If we remove the parentheses from our infix version, we get the ambiguous

$13.5 - 1.5 / 3.4 + 0.6$

Is that

$(13.5 - 1.5) / (3.4 + 0.6)$  or  
 $13.5 - (1.5 / 3.4) + 0.6?$

4

## Postfix and Stacks

- The best way to interpret postfix notation is with a stack.
- Whenever the next item is a number, we push it on the stack.
- When the next item is an operator,
  - the appropriate number of operands for the operator are popped off the stack,
  - the operator is applied, and
  - the result is pushed back on the stack.

5

## Evaluating Postfix

**Infix:**  $(13.5 - 1.5) / (3.4 + 0.6)$

**Postfix:** 13.5 1.5 - 3.4 0.6 + /

Postfix Item	Stack
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

6

## Software Patterns

- A software pattern is a design of a solution to a recurring software problem.
- Patterns are independent of implementation language.
- They are more general and flexible than algorithms and data structures.

7

## Model/View/Controller (MVC)

- Model/View/Controller is a useful pattern for the design of interactive programs with a GUI
- MVC has a long history going back to one of the earliest OO languages, Smalltalk.
- MVC has strongly affected the Java event model.
  - Event sources are part of the view.
  - Event listeners are part of the controller.

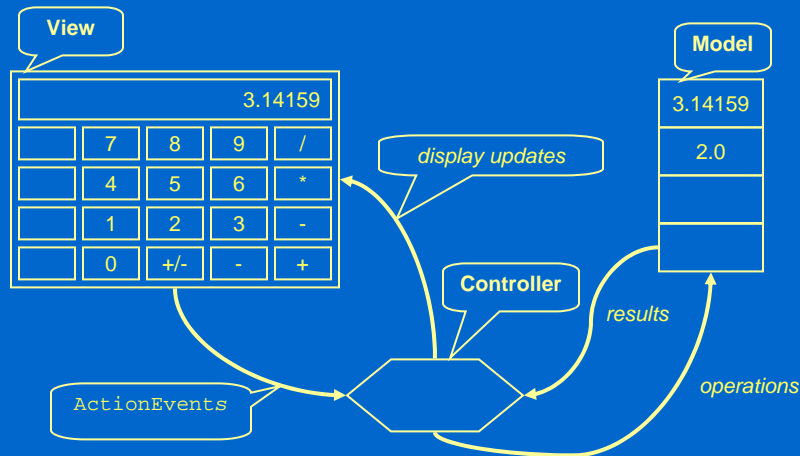
8

## MVC and our Calculator

- The *view* is the class (`Calculator`) that presents the user interface, the keypad and display.
- The *controller* is the class (`CalculatorController`) that interacts with the user, that processes events and updates the calculator without worrying about the details of the display.
- The *model* is the class (`CalculatorModel`) that handles computation, the CPU and memory of the calculator, if you want to think of it that way.
- The application (`CalculatorApp`), of course, pulls these pieces together with a `main()`.

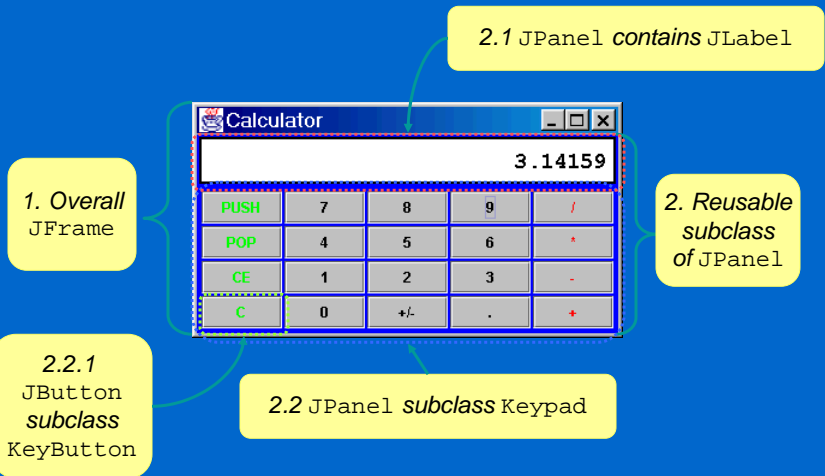
9

## MVC in the Calculator



10

## Calculator GUI



11

## Calculator, 1

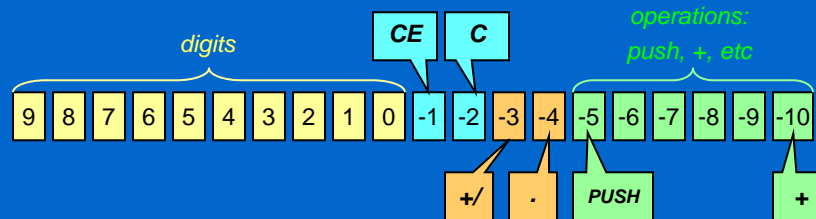
```
public class Calculator extends javax.swing.JPanel
    implements ActionListener
{
    class Keypad extends JPanel { . . . }
    class KeyButton extends JButton { . . . }

    static final String L_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 ); }
}
```

12

## Calculator Key Codes



13

## Calculator, 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 );
    }
}
```

14

## What Is an Event's `ActionCommand`

- By default it is the label on the component (e.g., `JButton`) that sent the `ActionEvent`.
- You can set it and use it to tell you which button sent the event.

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

15

## Calculator, 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 ));
        . . .
    }
}
```

16

## Calculator, 2

```
private JLabel display;

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 );
}
```

17

## CalculatorController: the *Controller*

- This class's task is the hardest because it is the messiest.
- Its job is to interpret the user's button presses, update the display, and use the *model* to perform all the calculations.
- The `CalculatorController` is the class that connects the other parts of the pattern. It knows about both the view and the model, while the view knows only about the controller, and the model is used by the controller, but never uses (calls) it back.

18

## Controller and Model

- Is the calculator display a copy of what is on the top of the stack?
- What goes on the stack? `Doubles`?
- When we enter a number on the calculator do we want to think of the intermediate values as doubles or as characters?
- To avoid round off error, we treat the display as a `String` (actually a `StringBuffer`) that has to be pushed onto the stack as a `Double`.

19

## The Controller and *State*

- The other difficulty for the controller is that its behavior depends on what has already happened.
- For instance,
  - if the user starts entering a new number by clicking 3 the controller should assume that the user is entering an integer.
  - But if the user then hits ., the controller must now assume a floating point number.
  - If the user then hits a second ., that's an error, because the controller has already seen a decimal point.
- The behavior of the controller depends on *previous state*.

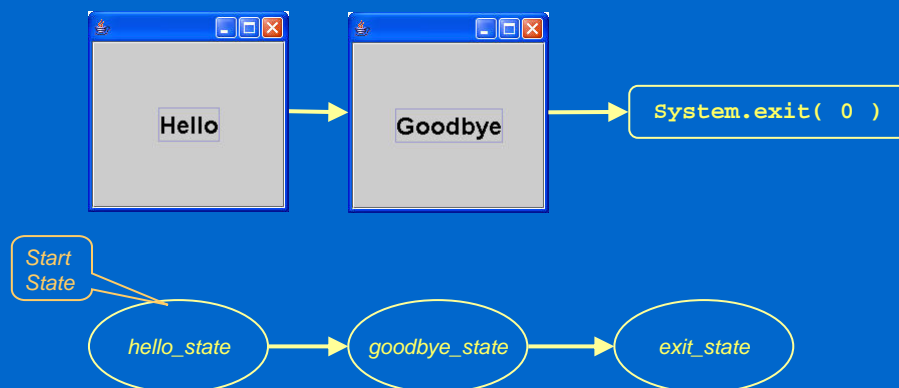
20

## Finite State Machines

- This kind of situation is frequently best handled by another design pattern called a *finite state machine* or *finite state automaton*.
- A FSM is an object that can occupy one of a finite number of states.
- One of these states, called the *start state*, specifies the state of a freshly created instance of the FSM.
- The FSM accepts *input tokens* or events, one at a time.
- The FSM specifies a *transition* to a new state for each possible state and input token.
- The transition often specifies an action, which is what makes the FSM useful.

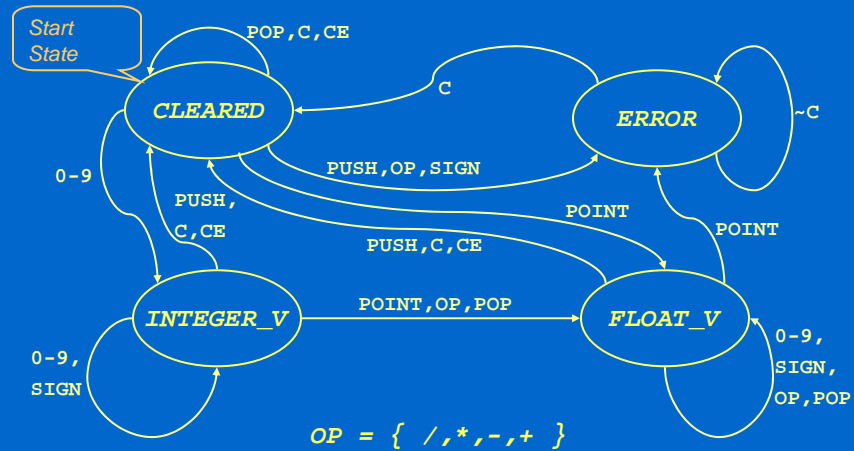
21

## Hello Application as FSM



22

## The CalculatorController FSM



23

## FSM in Operation

Let's follow the example,

3.14 PUSH 2 \*

(or, in infix, 3.14 \* 2), through the FSM:

1. We start in the CLEARED state (the **start** state)
2. 3 puts us into the INTEGER\_V state
3. . transitions us to the FLOAT\_V state
4. 1 and 4 just add to the number in the accumulator while remaining in FLOAT\_V
5. PUSH transitions us to CLEARED and pushes the value 3.14 onto the stack in the model

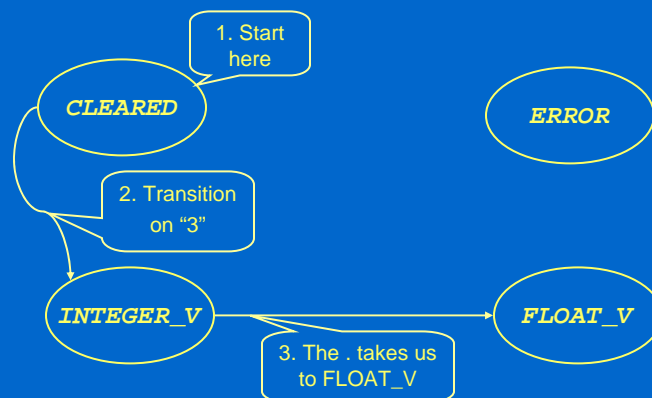
24

## FSM in Operation, 2

- 2 transitions us to `INTEGER_V` and starts a new number in the accumulator
- \* is an operation, and implicitly pushes 2 onto the model stack. It also transitions us momentarily to `CLEARED`, then causes the controller to invoke the `mul()` method on the model returning the result 6.28. The returned result is sent to the accumulator putting us back in `FLOAT_V`.

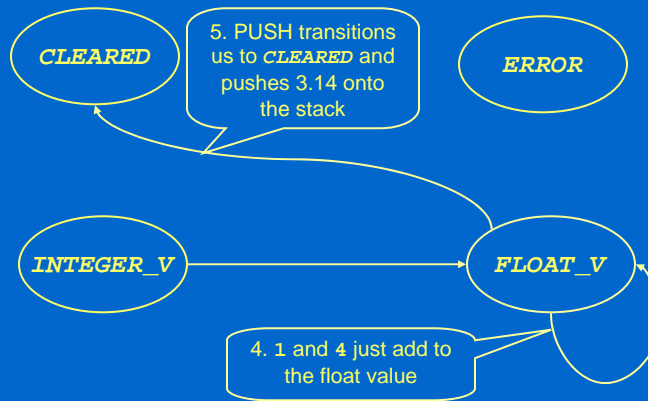
25

## FSM in Operation, 3



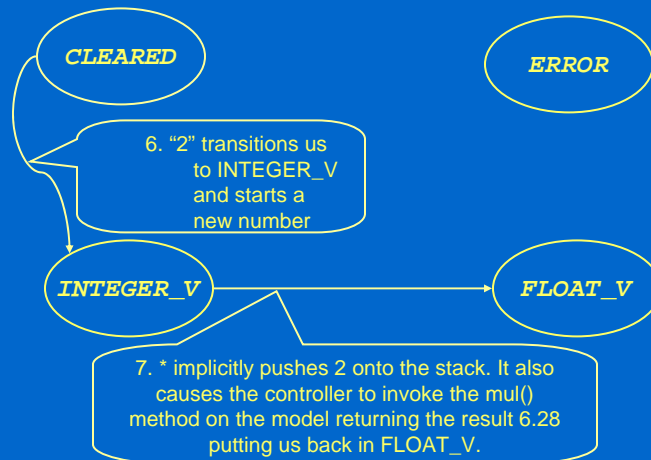
26

## FSM in Operation, 4



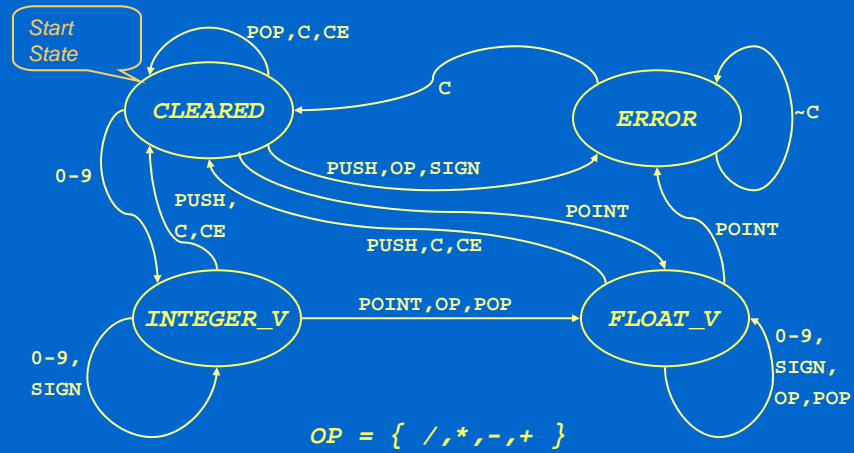
27

## FSM in Operation, 5



28

## The CalculatorController FSM



29

## 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;
  
```

30

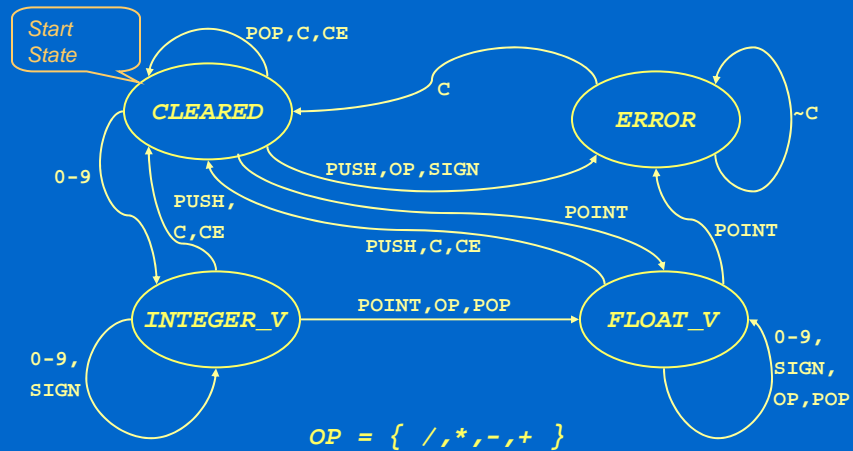
## CalculatorController, 2

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

    if ( state == ERROR ) {
        // The only way to exit error is a general CLEAR
        if ( eType == Calculator.I_C ) {
            reset();
        } else
            return;
    }
}
```

31

## The CalculatorController FSM



32

## 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;
        // . . . cases INTEGER_V, FLOAT_V
    }
}
```

33

## 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(); }
```

34

## CalculatorModel Overview

After working with the controller, the model seems simple although it is where the computation occurs.

- Each operation method, `div()`, `mul()`, `sub()`, and `add()`, is careful to get the order of the arguments popped off the stack correct.
- All these methods and `CalculatorModel.pop()` can throw an `EmptyStackException` because they each invoke `Stack.pop()`.
- The methods don't catch the exception so it is passed back to the calling routine in a way that we will examine in a later lecture.

35

## CalculatorModel Code

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

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

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

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

• • •  
36

## CalculatorApp

```
public class CalculatorApp {
    public static void main( String[] args ) {
        JFrame top = new JFrame( "Calculator" );
        top.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        Calculator calc = new Calculator();
        CalculatorModel model = new CalculatorModel( );
        CalculatorController contrl =
            new CalculatorController( calc, model );
        calc.setController( contrl );
    }
}
```