Building a GUI in Java with Swing

CITS1001 extension notes

Rachel Cardell-Oliver

Lecture Outline

1. Swing components
2. Building a GUI
3. Animating the GUI

Swing

- A collection of classes of GUI components
- Contains classes to implement windows, buttons, menus etc.
- Uses “event listener” model to attach handlers to the UI components
- Huge and complex collection of packages
- Augments/replaces older AWT

Basic Workflow

- Create a window to hold your entire GUI
- Arrange UI components inside the window
  - Add individual components, or
  - Group components into containers and add *them* to the window
- Attach handler(s) to each UI component to handle the events they generate

The main window

- The class `javax.swing.JFrame` is used as the outermost window in a Swing GUI

↳ java.lang.Object
↳ java.awt.Component
↳ java.awt.Container
↳ java.awt.Window
↳ java.awt.Frame
↳ javax.swing.JFrame

Extend `JFrame`

```
import javax.swing.*;
import java.awt.*;

public class MyGUI extends Jframe {
  // Declare UI components

  public MyGUI(String title) {
    super(title);
      // Add UI components
    pack();
    setVisible(true);
  }
}
```

Constructor

- First, call the superclass constructor
- Then add any UI components (none yet)
- Then “pack” the window - this lays out the UI components
- Make the window visible
  - Currently the window appears, but does nothing at all!

Getting rid of the window

- What should happen if the user closes the window?
- For the main application, closing should terminate the program

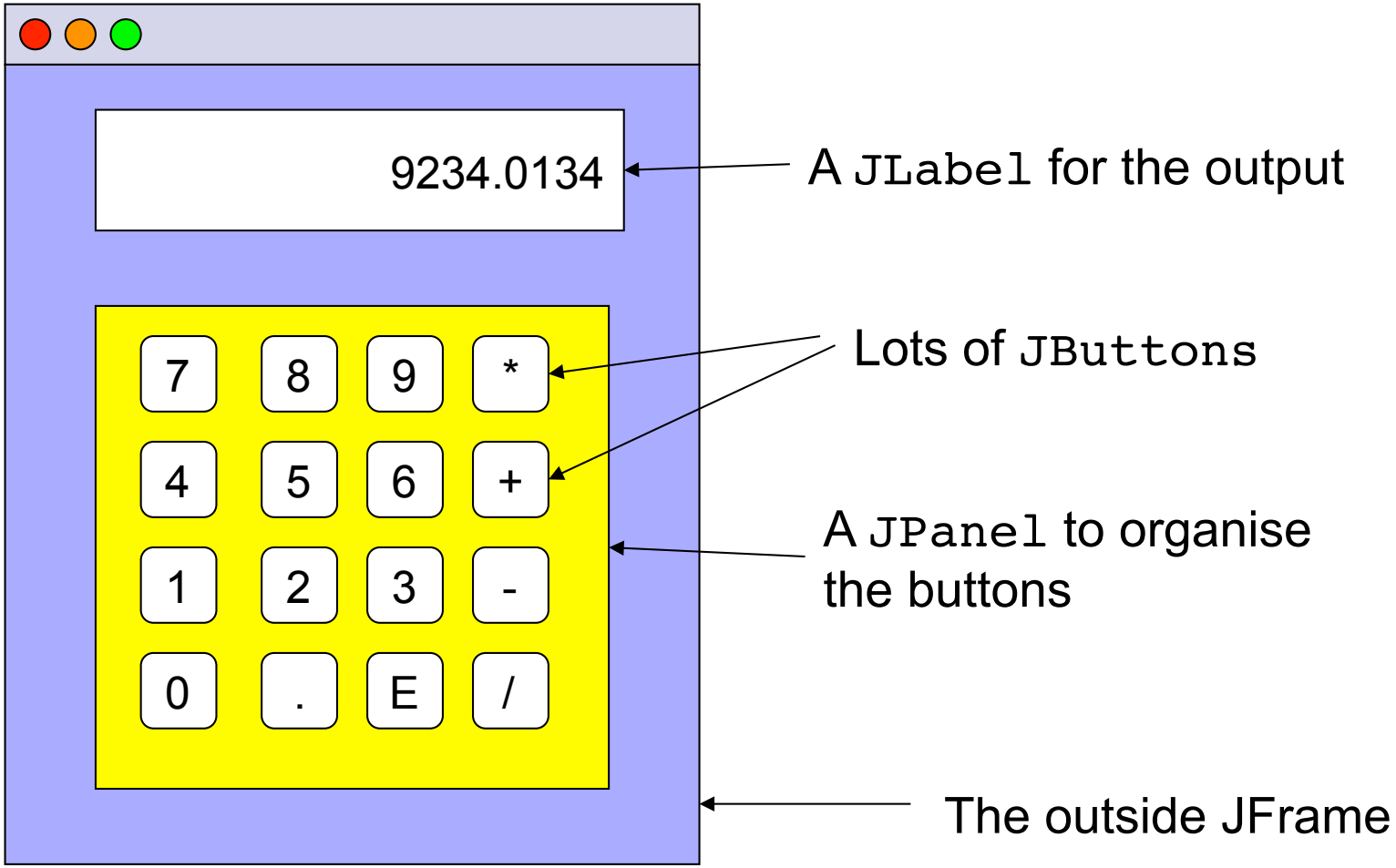
```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

- Other options are
  - `DO_NOTHING_ON_CLOSE, HIDE_ON_CLOSE, DISPOSE_ON_CLOSE`

UI Components

- Many UI components, including
  - `JLabel` - information labels
  - `JButton` - clickable buttons
  - `JTextField` - text entry fields
- Organize these with
  - A layout manager, and
  - `JPanel` to group components

An RPN Calculator



Layout Managers

- The contents of any container are laid out according to a specified “layout manager”
  - `java.awt.FlowLayout` - free flowing
  - `java.awt.BorderLayout` - five different areas (center, north, south, east, west)
  - `java.awt.GridLayout` - rectangular grid
  - `java.awt.GridBagLayout` - complex grid

Which layouts?

- The overall `JFrame` will have a `BorderLayout` with a `JLabel` in the "North" position and the button panel in the "Center" position
- In the constructor for `MyGUI` we add

```
setLayout(new BorderLayout());
```

Add the display label

- In the instance variables

```
private JLabel display;
```

- In the constructor

```
display = new JLabel
  (“0”,SwingConstants.RIGHT);
add(display,BorderLayout.NORTH);
```

Declare the buttons

- Extra instance variables

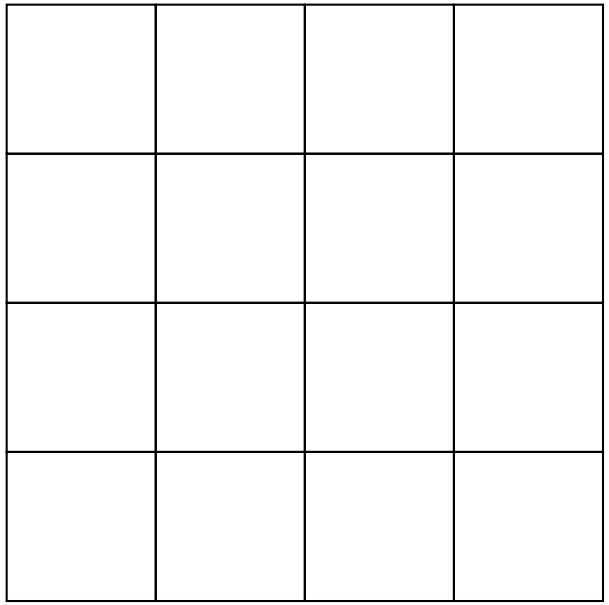
```
private JButton[] numbers = new
JButton[10];

private JButton add;
private JButton sub;
private JButton mul;
private JButton div;

private JButton decimal;
private JButton enter;
```

Create the buttons

```
for (int i=0; i<10; i++)
numbers[i] = new JButton(String.valueOf(i));

decimal = new JButton(".");

add = new JButton("+");
sub = new JButton("-");
div = new JButton("/");
mul = new JButton("*");

enter = new JButton("Enter");
```

Create the button panel

```
JPanel buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(4,4));
```



Add buttons to the panel..

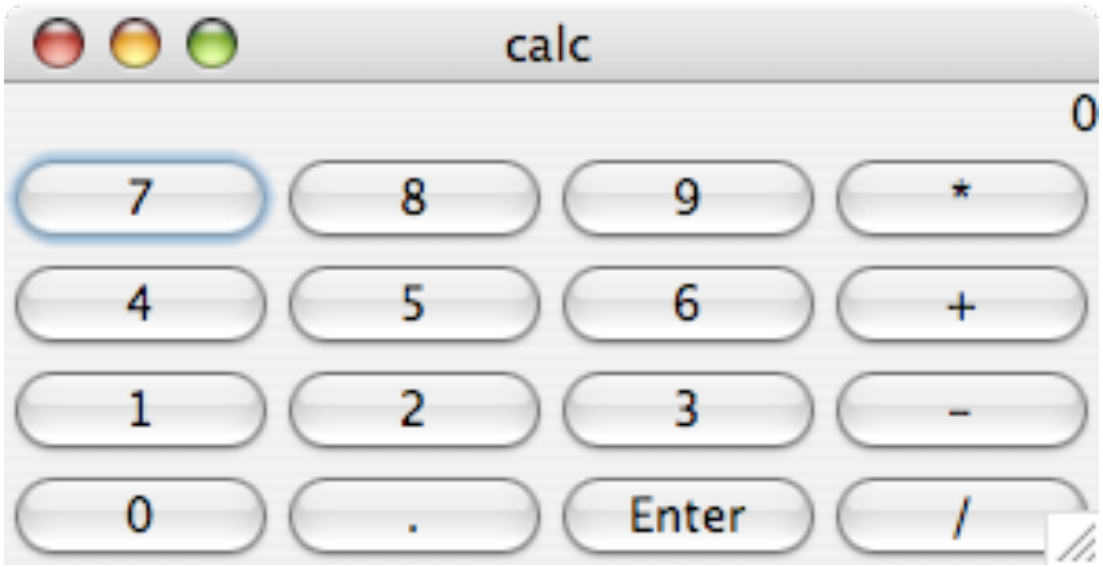
```
buttonPanel.add(numbers[7]);
buttonPanel.add(numbers[8]);
buttonPanel.add(numbers[9]);
buttonPanel.add(mul);

...
```

- Each new component occupies the next empty grid position working row by row..

```
...
buttonPanel.add(numbers[0]);
buttonPanel.add(decimal);
buttonPanel.add(enter);

buttonPanel.add(div);
```

Add panel to main `JFrame`

```
add(buttonPanel,BorderLayout.CENTER);
```



Ugly GUI

- This is pretty ugly at the moment - it can be beautified with borders, background colours, images etc.
- However this is beyond our immediate scope!

Adding logic

- Now we need to add the logic - to actually respond to button clicks etc.
- The GUI will be responsible for the entry and display of the numbers, while an existing class will manage the arithmetic

Make the GUI an ActionListener

- Declare that the class implements the `java.awt.event.ActionListener` interface

```
import java.awt.event.*;
public class MyGUI extends JFrame
  implements ActionListener {

   // previous code goes here

}
```

Register with all the buttons

- While the GUI constructs the buttons, it should register itself with them

```
for (int i=0; i<10; i++) {
  numbers[i] = new JButton(String.valueOf
  (i));
  numbers[i].addActionListener(this);
}
decimal = new JButton(".");
decimal.addActionListener(this);
```

A required method

- To claim to be an `ActionListener`, the GUI object must implement

```
void actionPerformed(ActionEvent e)
```

- Whenever an action occurs on a UI component (e.g. a `JButton`) the button calls *this method* of all of its registered listeners

What should it do?

- Every time the user clicks a button, the button will call the `actionPerformed` method of the GUI object
- The GUI object therefore needs to
  - Work out which button was pushed
  - Do the appropriate thing

Which button was pushed?

- The `ActionEvent` object contains this information and it can be obtained using

  `e.getActionCommand()`
- Each button has an “action command”
  - The default “action command” is simply the label on the button, or
  - (Better) the programmer can set the action command on construction of the button

What should our GUI do?

```
public void actionPerformed(ActionEvent e) {
      String s = e.getActionCommand();
      switch (s.charAt(0)) {
          case '+':
          case '-':
          case '*':
          case '/': doOperation(s);
                    break;
          case 'E': doEnter(s);
                    break;
          default: processDigitOrDecimal(s);
      }
  }
}
```

Processing digits

- Extract the currently displayed string from the `JLabel` that is the display

```
String curr = display.getText();
```

- Perform some processing to decide how the display should be updated, and then set some new text

```
display.setText(update);
```

Processing logic

- Logic is surprisingly involved!
  - If display at maximum length, do nothing
  - If button pressed was “.” and display already contains a “.” do nothing
  - If display currently “0”
    - If button pushed was a digit then replace the 0
    - If button pushed was “.” then append it to the 0
  - Otherwise
    - Append the string to the current display

Regular Expressions

- A regular expression, often called a **pattern**, is an expression that describes a set of strings.
- The IEEE POSIX Basic Regular Expressions (BRE) standard can be used to denote regular expressions

RE examples

- [ab6] matches a *or* b *or* 6
- [a-z] matches any lower case letter from a to z
- [a-z]* matches any *sequence* of zero or more lower case letters
- [a]*b[0-9]* matches aaab04324324, b, ab999 etc
- a-[c]* matches a followed by - followed by zero or more c characters

String.matches

- public boolean **matches**(String regex)
  - Tells whether or not this string matches the given regular expression.
  - Regular expressions in Java use the Posix standard

A simple test program for experimenting with REs

```
String s = "aaabbb";

if (s.matches("[a]*[b]*")) {
   System.out.println("matches");
} else {
   System.out.println(“does not match");
}
```