

# GUI & Event Driven Programming using Java

- *Introduction*

- Graphical user interface (GUI)
  - Presents a user-friendly mechanism for interacting with an application
  - Often contains title bar, menu bar containing menus, buttons and combo boxes
  - Built from GUI components

# Graphical Components

button

menus

title bar

menu bar

combo box



scroll bars

- Dialog boxes
  - Used by applications to interact with the user
  - Provided by Java's JOptionPane class
    - Contains input dialogs and message dialogs

```
1 // Fig. 11.2: Addition.java
2 // Addition program that uses JOptionPane for input and output.
3 import javax.swing.JOptionPane; // program uses JOptionPane
4
5 public class Addition
6 {
7     public static void main( String args[] )
8     {
9         // obtain user input from JOptionPane input dialogs
10        String firstNumber =
11            JOptionPane.showInputDialog( "Enter first integer" );
12        String secondNumber =
13            JOptionPane.showInputDialog( "Enter second integer" );
14
15        // convert String inputs to int values for use in a calculation
16        int number1 = Integer.parseInt( firstNumber );
17        int number2 = Integer.parseInt( secondNumber );
18
19        int sum = number1 + number2; // add numbers
20
21        // display result in a JOptionPane message dialog
22        JOptionPane.showMessageDialog( null, "The sum is " + sum,
23            "Sum of Two Integers", JOptionPane.PLAIN_MESSAGE );
24    } // end method main
25 } // end class Addition
```

Show input dialog to receive first integer

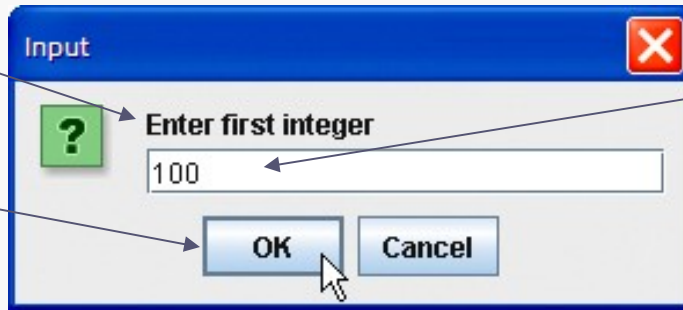
Show input dialog to receive second integer

Show message dialog to output sum to user

- Addition.java (2 of 2)

Input dialog displayed by lines 10–11

Prompt to the user

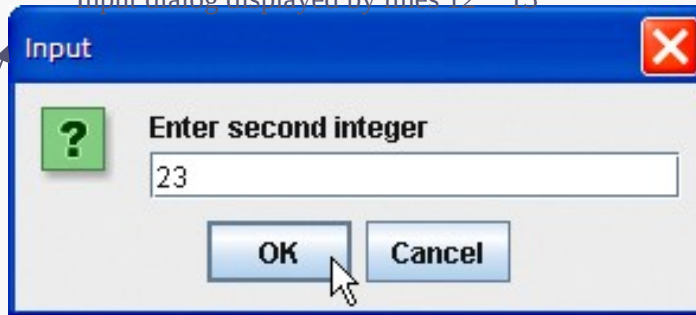


Text field in which the user types a value

When the user clicks **OK**, `showInputDialog` returns to the program the **100** typed by the user as a **String**. The program must convert the **String** to an **int**

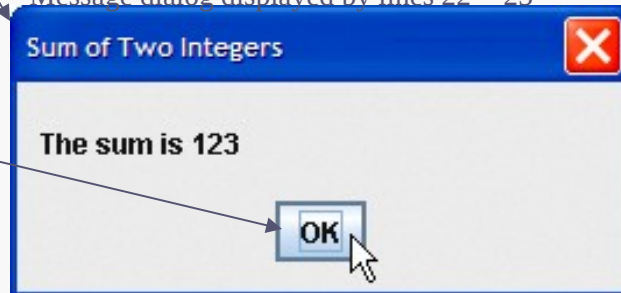
Input dialog displayed by lines 12–13

title bar



Message dialog displayed by lines 22–23

When the user clicks **OK**, the message dialog is dismissed (removed from the screen)







Message dialog type	Icon	Description
<code>ERROR_MESSAGE</code>		A dialog that indicates an error to the user.
<code>INFORMATION_MESSAGE</code>		A dialog with an informational message to the user.
<code>WARNING_MESSAGE</code>		A dialog warning the user of a potential problem.
<code>QUESTION_MESSAGE</code>		A dialog that poses a question to the user. This dialog normally requires a response, such as clicking a Yes or a No button.
<code>PLAIN_MESSAGE</code>	no icon	A dialog that contains a message, but no icon.

Fig. 11.3 | `JOptionPane` static constants for message dialogs.

# 11.3 Overview of Swing Components

- Swing GUI components
  - Declared in package `javax.swing`
  - Most are pure Java components
  - Part of the Java Foundation Classes (JFC)



<b>Component</b>	<b>Description</b>
<b>JLabel</b>	<b>Displays uneditable text or icons.</b>
<b>TextField</b>	<b>Enables user to enter data from the keyboard. Can also be used to display editable or uneditable text.</b>
<b>Button</b>	<b>Triggers an event when clicked with the mouse.</b>
<b>CheckBox</b>	<b>Specifies an option that can be selected or not selected.</b>
<b>ComboBox</b>	<b>Provides a drop-down list of items from which the user can make a selection by clicking an item or possibly by typing into the box.</b>
<b>List</b>	<b>Provides a list of items from which the user can make a selection by clicking on any item in the list. Multiple elements can be selected.</b>
<b>Panel</b>	<b>Provides an area in which components can be placed and organized. Can also be used as a drawing area for graphics.</b>

# Swing vs. AWT

- Abstract Window Toolkit (AWT)
  - Precursor to Swing
  - Declared in package `java.awt`
  - Does not provide consistent, cross-platform look-and-feel

## Portability Tip 11.1

- Swing components are implemented in Java, so they are more portable and flexible than the original Java GUI components from package `java.awt`, which were based on the GUI components of the underlying platform. For this reason, Swing GUI components are generally preferred.

# Lightweight vs. Heavyweight GUI Components

- Lightweight components
  - Not tied directly to GUI components supported by underlying platform
- Heavyweight components
  - Tied directly to the local platform
  - AWT components
  - Some Swing components

## Superclasses of Swing's Lightweight GUI Components

- Class `Component` (package `java.awt`)
  - Subclass of `Object`
  - Declares many behaviors and attributes common to GUI components
- Class `Container` (package `java.awt`)
  - Subclass of `Component`
  - Organizes Components
- Class `JComponent` (package `javax.swing`)
  - Subclass of `Container`

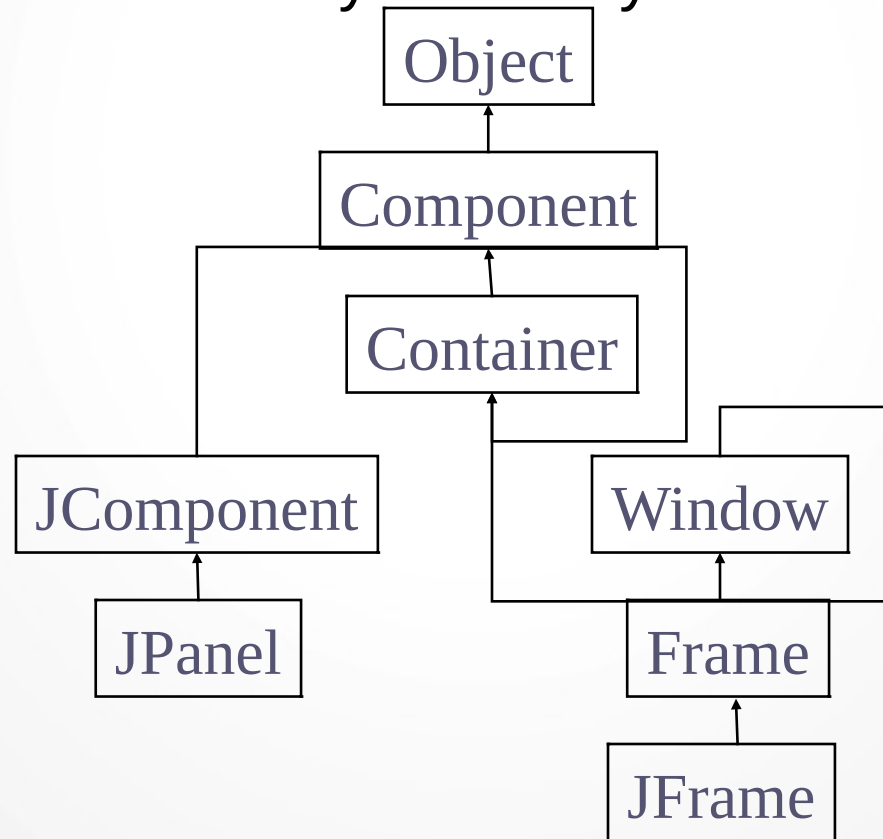
- Common lightweight component features
  - Pluggable look-and-feel to customize the appearance of components
  - Shortcut keys (called mnemonics)
  - Common event-handling capabilities
  - Brief description of component's purpose (called tool tips)
  - Support for localization

# Swing Components

- Swing is a collection of libraries that contains primitive *widgets* or *controls* used for designing *Graphical User Interfaces* (GUIs).
- Commonly used classes in javax.swing package:
  - JButton, JTextBox, JTextArea, JPanel, JFrame, JMenu, JSlider, JLabel, JIcon, ...
  - There are many, many such classes to do anything imaginable with GUIs
  - Here we only study the basic architecture and do simple examples

# Swing components, cont.

- Each component is a Java class with a fairly extensive inheritance hierarchy:





# Using Swing Components

- Very simple, just create object from appropriate class – examples:
  - `JButton but = new JButton();`
  - `JTextField text = new JTextField();`
  - `JTextArea text = new JTextArea();`
  - `JLabel lab = new JLabel();`
- Many more classes. Don't need to know every one to get started.
- See ch. 9 Hortsman

# Adding components

- Once a component is created, it can be added to a container by calling the container's **add** method:

```
Container cp = getContentPane(); ← This is required
```

```
cp.add(new JButton("cancel"));
```

```
cp.add(new JButton("go"));
```

How these are laid out is determined by the layout manager.

# Laying out components

- Not so difficult but takes a little practice
- Do not use absolute positioning – not very portable, does not resize well, etc.

# Laying out components

- Use layout managers – basically tells form how to align components when they're added.
- Each Container has a layout manager associated with it.
- A JPanel is a Container – to have different layout managers associated with different parts of a form, tile with JPanels and set the desired layout manager for each JPanel, then add components directly to panels.

# Layout Managers

- Java comes with 7 or 8. Most common and easiest to use are
  - FlowLayout
  - BorderLayout
  - GridLayout
- Using just these three it is possible to attain fairly precise layout for most simple applications.

# Setting layout managers

- Very easy to associate a layout manager with a component. Simply call the **setLayout** method on the Container:

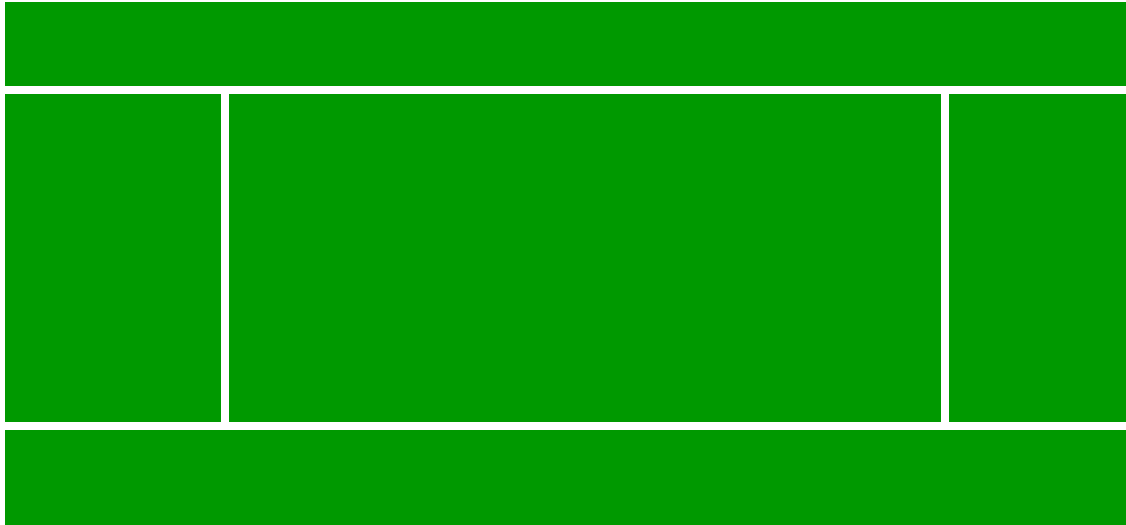
```
JPanel p1 = new JPanel();  
p1.setLayout(new FlowLayout(FlowLayout.LEFT));
```

```
JPanel p2 = new JPanel();  
p2.setLayout(new BorderLayout());
```

As Components are added to the container, the layout manager determines their size and positioning.

# Layouts

## BorderLayout



Position must be specified, e.g. `add ("North", myComponent)`

# Layouts

## FlowLayout

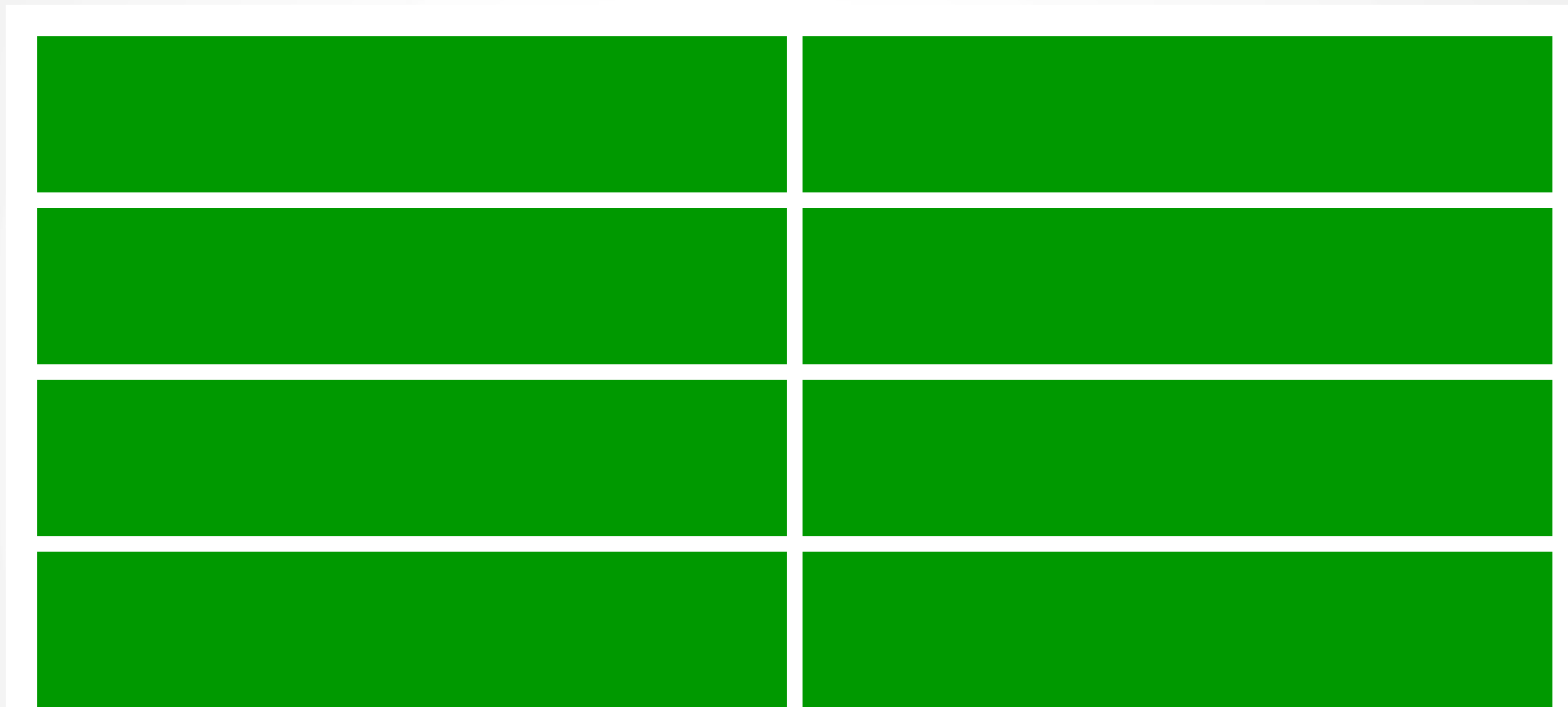


FlowLayout is the default layout manager for every JPanel. It simply lays out components from left to right, starting new rows if necessary



# Layouts

## GridLayout



GridLayout simply makes a bunch of components equal in size and displays them in the requested number of rows and columns .

# Event handling

# What are events?

- All components can listen for one or more *events*.
- Typical examples are:
  - Mouse movements
  - Mouse clicks
  - Hitting any key
  - Hitting return key
  - etc.
- Telling the GUI what to do when a particular event occurs is the role of the event handler.

# ActionEvent

- In Java, most components have a special event called an *ActionEvent*.
- This is loosely speaking the most common or canonical event for that component.
- A good example is a click for a button.
- To have any component listen for ActionEvents, you must register the component with an ActionListener.

## Delegation, cont.

- This is referred to as the Delegation Model.
- When you register an ActionListener with a component, you must pass it the class which will handle the event – that is, do the work when the event is triggered.
- For an ActionEvent, this class must implement the ActionListener interface

# actionPerformed

- The actionPerformed method has the following signature:  
void actionPerformed(ActionEvent)
- The object of type(ActionEvent) passed to the event handler is used to query information about the event.
- Some common methods are:
  - getSource()
    - object reference to component generating event
  - getActionCommand()
    - some text associated with event (text on button, etc).

## actionPerformed, cont.

- These methods are particularly useful when using one eventhandler for multiple components.

# Simplest GUI

```
import javax.swing.JFrame;
class SimpleGUI extends JFrame{
    SimpleGUI(){
        setSize(400,400); //set frames size in pixels
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        show();
    }

    public static void main(String[] args){
        SimpleGUI gui = new SimpleGUI();
        System.out.println("main thread coninues");
    }
}
```



# Another Simple GUI

```
import javax.swing.*;
class SimpleGUI extends JFrame{
    SimpleGUI(){
        setSize(400,400); //set frames size in pixels
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        JButton but1 = new JButton("Click me");
        Container cp = getContentPane();//must do this
        cp.add(but1);
        show();
    }

    public static void main(String[] args){
        SimpleGUI gui = new SimpleGUI();
        System.out.println("main thread coninues");
    }
}
```

# Add Layout Manager

```
import javax.swing.*; import java.awt.*;
class SimpleGUI extends JFrame{
    SimpleGUI(){
        setSize(400,400); //set frames size in pixels
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        JButton but1 = new JButton("Click me");
        Container cp = getContentPane();//must do this
        cp.setLayout(new FlowLayout(FlowLayout.CENTER));
        cp.add(but1);
        show();
    }

    public static void main(String[] args){
        SimpleGUI gui = new SimpleGUI();
        System.out.println("main thread coninues");
    }
}
```

# Add call to event handler

```
import javax.swing.*; import java.awt.*;
class SimpleGUI extends JFrame{
    SimpleGUI(){
        setSize(400,400); //set frames size in pixels
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        JButton but1 = new JButton("Click me");
        Container cp = getContentPane();//must do this
        cp.setLayout(new FlowLayout(FlowLayout.CENTER));
        but1.addActionListener(new MyActionListener());
        cp.add(but1);
        show();
    }
    public static void main(String[] args){
        SimpleGUI gui = new SimpleGUI();
        System.out.println("main thread coninues");
    }
}
```

# Event Handler Code

```
class MyActionListener implements ActionListener{  
    public void actionPerformed(ActionEvent ae){  
        JOptionPane.showMessageDialog(“I got clicked”, null);  
    }  
  
}
```

# Add second button/event

```
class SimpleGUI extends JFrame{
    SimpleGUI(){
        /* .... */
        JButton but1 = new JButton("Click me");
        JButton but2 = new JButton("exit");
        MyActionListener al = new MyActionListener();
        but1.addActionListener(al);
        but2.addActionListener(al);
        cp.add(but1);
        cp.add(but2);
        show();
    }
}
```

# How to distinguish events –Less good way

```
class MyActionListener implements ActionListener{
    public void actionPerformed(ActionEvent ae){
        if (ae.getActionCommand().equals("Exit")){
            System.exit(1);
        }
        else if (ae.getActionCommand().equals("Click me")){
            JOptionPane.showMessageDialog(null, "I'm clicked");
        }
    }
}
```

# Good way

```
class MyActionListener implements ActionListener{
    public void actionPerformed(ActionEvent ae){
        if (ae.getSource() == but2){
            System.exit(1);
        }
        else if (ae.getSource() == but1){
            JOptionPane.showMessageDialog(null, "I'm clicked")
        }
    }
}
```

Question: How are but1, but2 brought into scope to do this?

Question: Why is this better?