



## CHAPTER 10

# Swing I

Shin-Jie Lee (李信杰)

Assistant Professor

Computer and Network Center

Department of Computer Science and Information Engineering

National Cheng Kung University



成功大學  
National Cheng Kung University



# Introduction to Swing

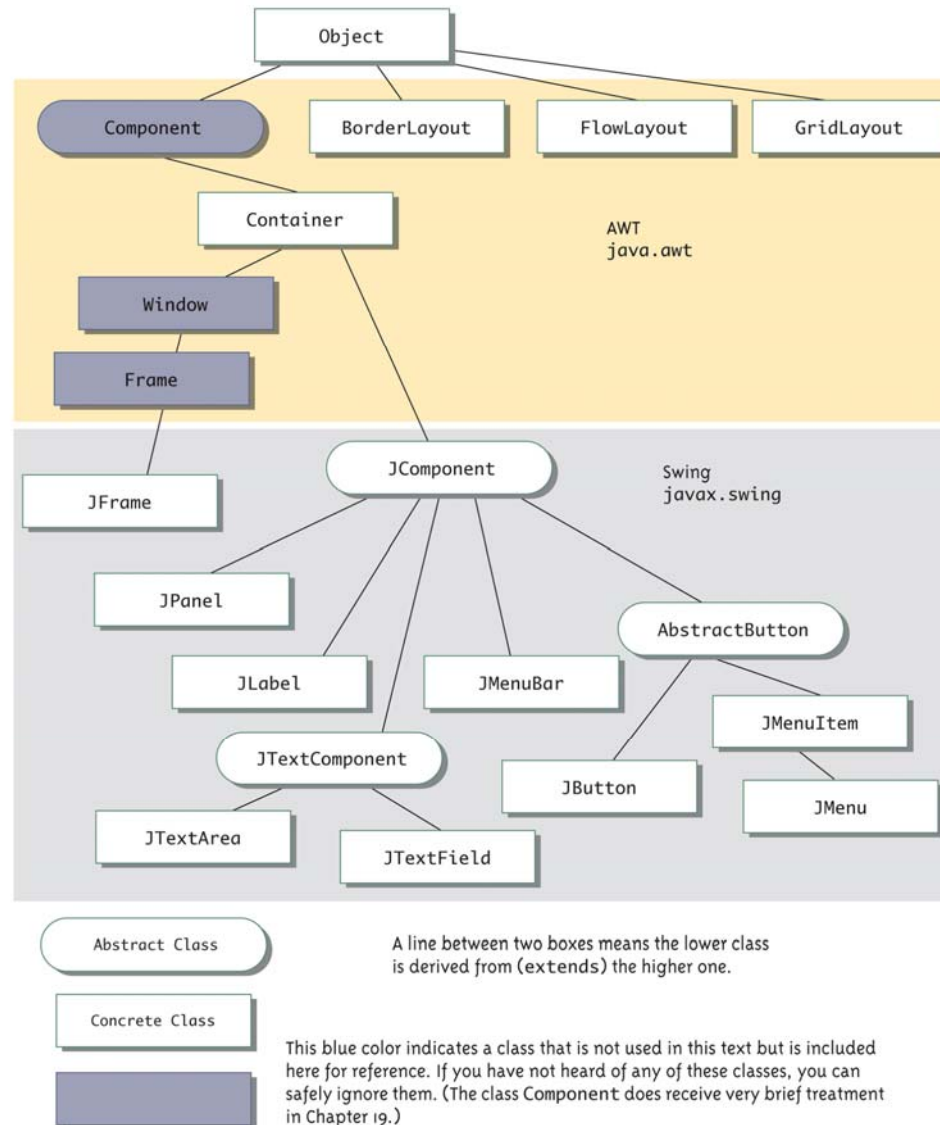
---

- ❑ A *GUI* (*graphical user interface*) is a windowing system that interacts with the user
- ❑ The Java *AWT* (*Abstract Window Toolkit*) package is the original Java package for creating *GUIs*
- ❑ The Swing package is an improved version of the AWT
  - However, it does not completely replace the AWT
  - Some AWT classes are replaced by Swing classes, but other AWT classes are needed when using Swing
- ❑ Swing GUIs are designed using a form of object-oriented programming known as *event-driven programming*



# Hierarchy of Swing and AWT Classes

Display 17.12 Hierarchy of Swing and AWT Classes





# A Simple Window

---

- ❑ A simple window can consist of an object of the **JFrame** class
  - A **JFrame** object includes a border and the usual three buttons for minimizing, changing the size of, and closing the window
  - The **JFrame** class is found in the **javax.swing** package

```
JFrame firstWindow = new JFrame();
```
- ❑ A **JFrame** can have components added to it, such as buttons, menus, and text labels
  - These components can be programmed for action

```
firstWindow.add(endButton);
```
  - It can be made visible using the **setVisible** method

```
firstWindow.setVisible(true);
```



# Lab

---

```
import javax.swing.JFrame;

public class MyFrame {

    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(800, 600);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```



# Events

---

- ❑ *Event-driven programming* is a programming style that uses a signal-and-response approach to programming
- ❑ An *event* is an object that acts as a signal to another object known as a *listener*
- ❑ The sending of an event is called *firing the event*
  - The object that fires the event is often a GUI component, such as a button that has been clicked



# Listeners

---

- ❑ A listener object performs some action in response to the event
  - A given component may have any number of listeners
  - Each listener may respond to a different kind of event, or multiple listeners may respond to the same events



# Exception Objects

---

- ❑ An exception object is an event
  - The throwing of an exception is an example of firing an event
- ❑ The listener for an exception object is the **catch** block that catches the event

```
try
{ . . . }
catch(ExceptionClass1 e)
{
    CodeToBeExecutedInAllCases
}
```





# Event Handlers

---

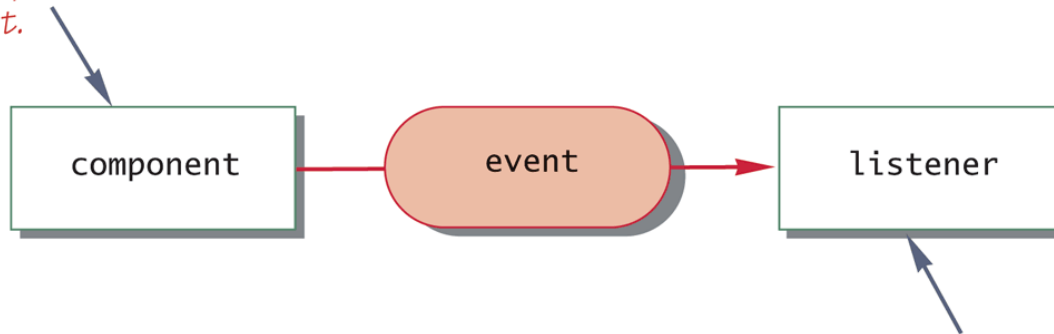
- ❑ A listener object has methods that specify what will happen when events of various kinds are received by it
  - These methods are called *event handlers*
- ❑ The programmer using the listener object will define or redefine these event-handler methods



# Event Firing and an Event Listener

Display 17.1 Event Firing and an Event Listener

*The component (for example, a button) fires an event.*



*This listener object invokes an event handler method with the **event** as an argument.*



# Event-Driven Programming

---

- ❑ In particular, *methods are defined that will never be explicitly invoked in any program*
  - Instead, methods are invoked automatically when an event signals that the method needs to be called



# Buttons

---

- ❑ A *button* object is created from the class **JButton** and can be added to a **JFrame**
  - The argument to the **JButton** constructor is the string that appears on the button when it is displayed

```
JButton endButton = new  
    JButton("Click to end program.");  
firstWindow.add(endButton);
```



# Action Listeners and Action Events

---

- ❑ Clicking a button fires an event
- ❑ The event object is "sent" to another object called a listener
  - This means that a method in the listener object is invoked automatically
  - Furthermore, it is invoked with the event object as its argument
- ❑ In order to set up this relationship, a GUI program must do two things
  1. It must specify, for each button, what objects are its listeners, i.e., it must register the listeners
  2. It must define the methods that will be invoked automatically when the event is sent to the listener



# Action Listeners and Action Events

---

```
EndingListener buttonEar = new  
    EndingListener();  
endButton.addActionListener(buttonEar);
```

- Above, a listener object named **buttonEar** is created and registered as a listener for the button named **endButton**
  - Note that a button fires events known as *action events*, which are handled by listeners known as *action listeners*



# Action Listeners and Action Events

---

- ❑ Different kinds of components require different kinds of listener classes to handle the events they fire
- ❑ An action listener is an object whose class implements the **ActionListener** interface
  - The **ActionListener** interface has one method heading that must be implemented  
`public void actionPerformed(ActionEvent e)`



# Action Listeners and Action Events

---

```
public void actionPerformed(ActionEvent e)
{
    System.exit(0);
}
```

- ❑ The **EndingListener** class defines its **actionPerformed** method as above
  - When the user clicks the **endButton**, an action event is sent to the action listener for that button
  - The **EndingListener** object **buttonEar** is the action listener for **endButton**
  - The action listener **buttonEar** receives the action event as the parameter **e** to its **actionPerformed** method, which is automatically invoked
  - Note that **e** must be received, even if it is not used





# Lab

---

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyButtonListener implements ActionListener{

    public void actionPerformed(ActionEvent e){
        System.out.println(e.getActionCommand());
        System.out.println(e.getSource());
    }

}
```



# Lab

---

```
import javax.swing.JButton;
import javax.swing.JFrame;

public class MyFrame {

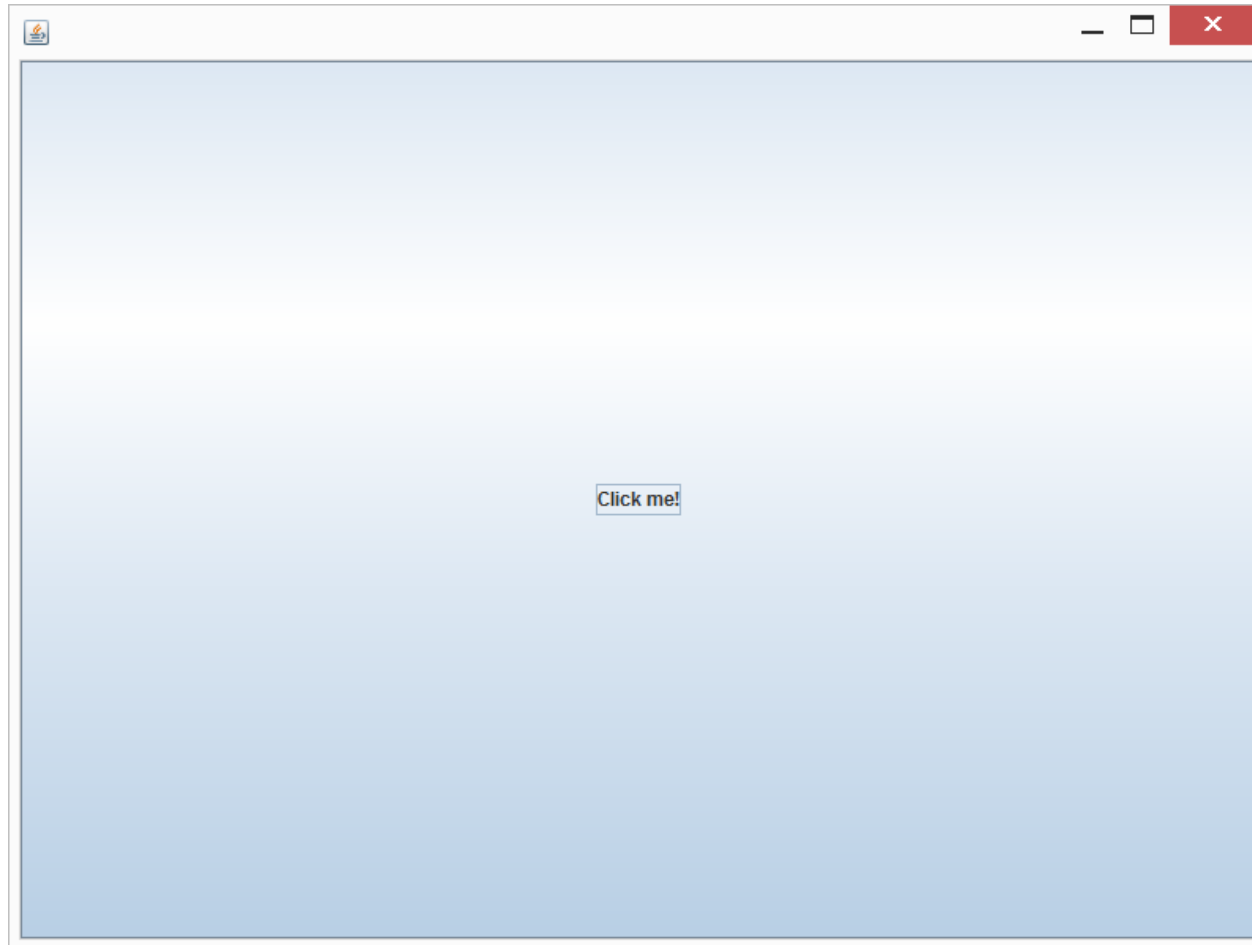
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(800, 600);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton btn = new JButton("Click me!");
        MyButtonListener mblistener = new MyButtonListener();
        btn.addActionListener(mblistener);
        frame.add(btn);

        frame.setVisible(true);
    }
}
```



# Lab





# Some Methods in the Class JFrame

## (Part 1 of 3)

---

### Display 17.3 Some Methods in the Class JFrame

---

The class JFrame is in the `javax.swing` package.

```
public JFrame()
```

Constructor that creates an object of the class JFrame.

```
public JFrame(String title)
```

Constructor that creates an object of the class JFrame with the title given as the argument.

(continued)



# Some Methods in the Class `JFrame`

## (Part 2 of 3)

### Display 17.3 Some Methods in the Class `JFrame`

```
public void setDefaultCloseOperation(int operation)
```

Sets the action that will happen by default when the user clicks the close-window button. The argument should be one of the following defined constants:

`JFrame.DO_NOTHING_ON_CLOSE`: Do nothing. The `JFrame` does nothing, but if there are any registered window listeners, they are invoked. (Window listeners are explained in Chapter 19.)

`JFrame.HIDE_ON_CLOSE`: Hide the frame after invoking any registered `WindowListener` objects.

`JFrame.DISPOSE_ON_CLOSE`: Hide and *dispose* the frame after invoking any registered window listeners. When a window is **disposed** it is eliminated but the program does not end. To end the program, you use the next constant as an argument to `setDefaultCloseOperation`.

`JFrame.EXIT_ON_CLOSE`: Exit the application using the `System.exit` method. (Do not use this for frames in applets. Applets are discussed in Chapter 18.)

If no action is specified using the method `setDefaultCloseOperation`, then the default action taken is `JFrame.HIDE_ON_CLOSE`.

Throws an `IllegalArgumentException` if the argument is not one of the values listed above.<sup>2</sup>

Throws a `SecurityException` if the argument is `JFrame.EXIT_ON_CLOSE` and the Security Manager will not allow the caller to invoke `System.exit`. (You are not likely to encounter this case.)

```
public void setSize(int width, int height)
```

Sets the size of the calling frame so that it has the width and height specified. Pixels are the units of length used.

(continued)



# Some Methods in the Class JFrame

## (Part 3 of 3)

---

### Display 17.3 Some Methods in the Class JFrame

---

```
public void setTitle(String title)
```

Sets the title for this frame to the argument string.

```
public void add(Component componentAdded)
```

Adds a component to the JFrame.

```
public void setLayout(LayoutManager manager)
```

Sets the layout manager. Layout managers are discussed later in this chapter.

```
public void setJMenuBar(JMenuBar menubar)
```

Sets the menubar for the calling frame. (Menus and menu bars are discussed later in this chapter.)

```
public void dispose()
```

Eliminates the calling frame and all its subcomponents. Any memory they use is released for reuse. If there are items left (items other than the calling frame and its subcomponents), then this does not end the program. (The method `dispose` is discussed in Chapter 19.)



# Containers and Layout Managers

---

- ❑ Multiple components can be added to the content pane of a **JFrame** using the **add** method
  - However, the **add** method does not specify how these components are to be arranged
- ❑ To describe how multiple components are to be arranged, a *layout manager* is used
  - There are a number of layout manager classes such as **BorderLayout**, **FlowLayout**, and **GridLayout**
  - If a layout manager is not specified, a default layout manager is used



# Border Layout Managers

---

- ❑ A **BorderLayout** manager places the components that are added to a **JFrame** object into five regions
  - These regions are: **BorderLayout.NORTH**, **BorderLayout.SOUTH**, **BorderLayout.EAST**, **BorderLayout.WEST**, and **BorderLayout.Center**
- ❑ A **BorderLayout** manager is added to a **JFrame** using the **setLayout** method
  - For example:  
`setLayout(new BorderLayout());`





# Border Layout Managers

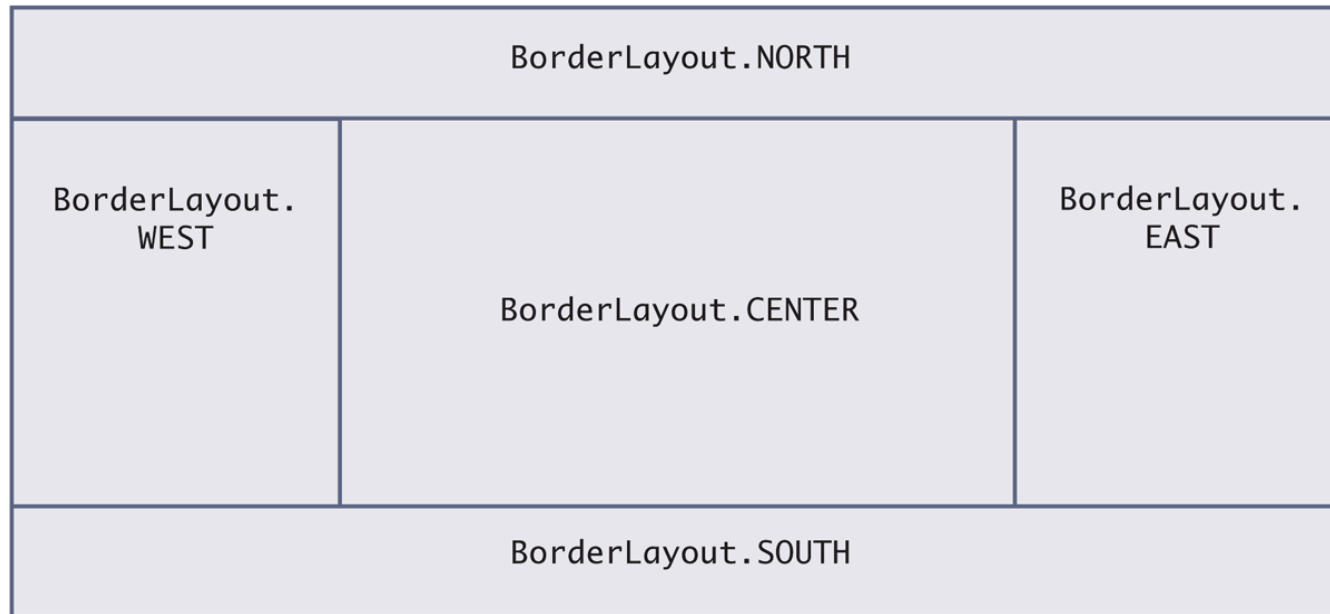
---

- ❑ The previous diagram shows the arrangement of the five border layout regions
  - Note: None of the lines in the diagram are normally visible
- ❑ When using a **BorderLayout** manager, the location of the component being added is given as a second argument to the **add** method
  - `add(label1, BorderLayout.NORTH);`**
  - Components can be added in any order since their location is specified



# BorderLayout Regions

Display 17.8 BorderLayout Regions





# Lab

```
import java.awt.BorderLayout;

import javax.swing.JButton;
import javax.swing.JFrame;

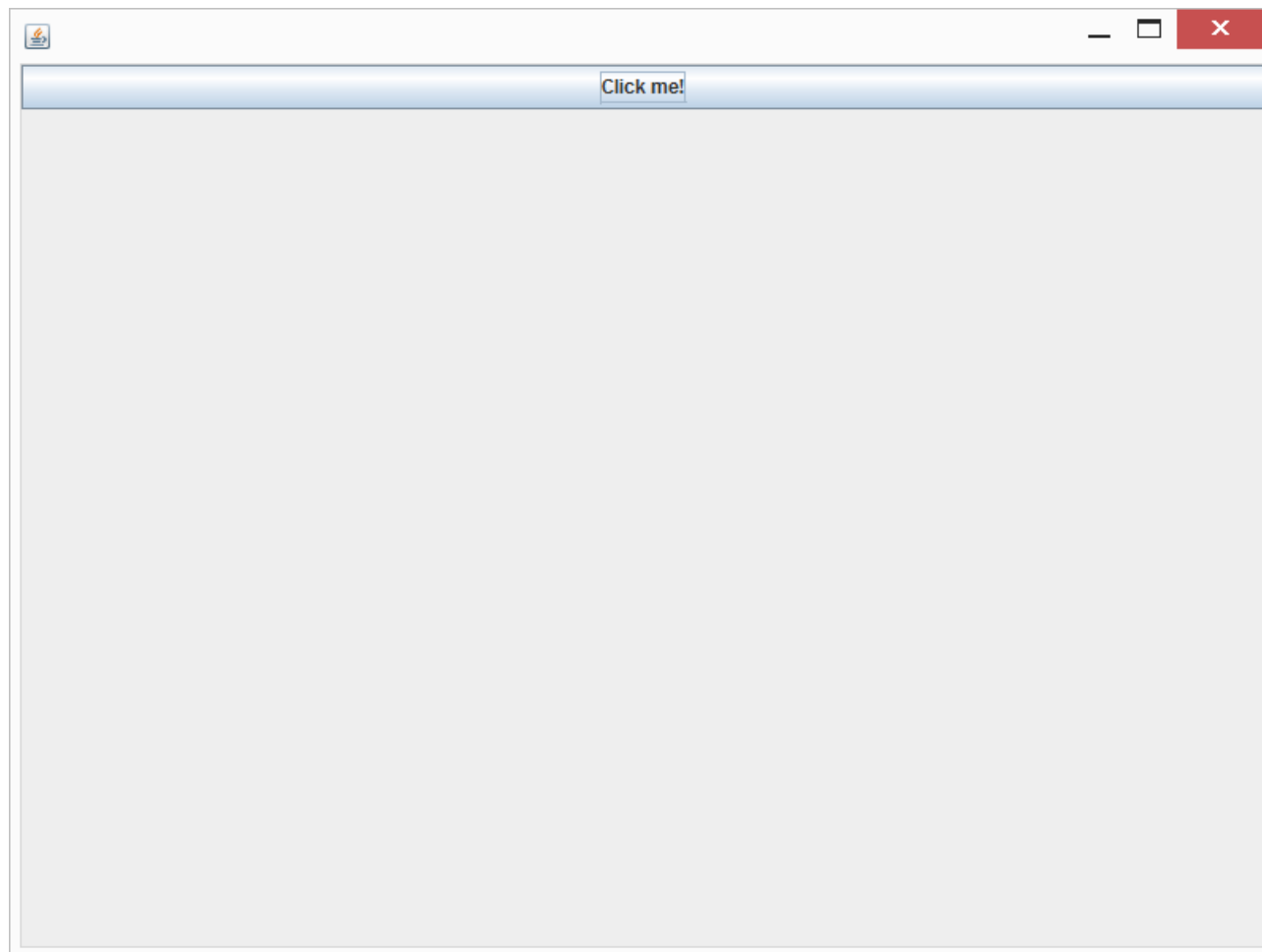
public class MyFrame {

    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(800, 600);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton btn = new JButton("Click me!");
        MyButtonListener mblistener = new MyButtonListener();
        btn.addActionListener(mblistener);

        frame.setLayout(new BorderLayout());
        frame.add(btn, BorderLayout.NORTH);

        frame.setVisible(true);
    }
}
```





# Some Layout Managers

**Display 17.10**    **Some Layout Managers**

LAYOUT MANAGER	DESCRIPTION
These layout manager classes are in the <code>java.awt</code> package.	
FlowLayout	Displays components from left to right in the order in which they are added to the container.
BorderLayout	Displays the components in five areas: north, south, east, west, and center. You specify the area a component goes into in a second argument of the <code>add</code> method.
GridLayout	Lays out components in a grid, with each component stretched to fill its box in the grid.



# Lab (No Layout)

```
import javax.swing.JButton;
import javax.swing.JFrame;

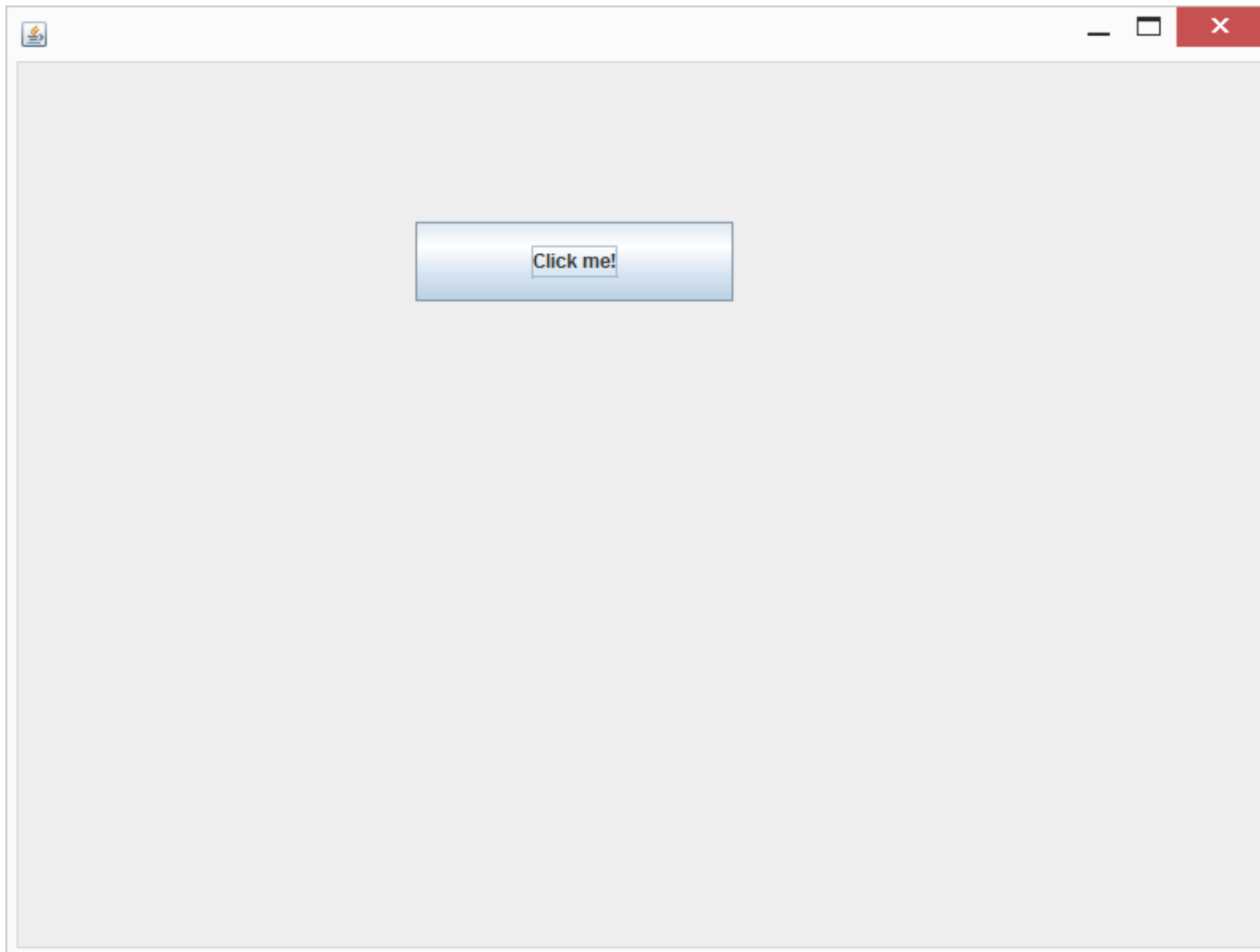
public class MyFrame {

    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(800, 600);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton btn = new JButton("Click me!");
        MyButtonListener mblistener = new MyButtonListener();
        btn.addActionListener(mblistener);
        btn.setLocation(250, 100);
        btn.setSize(200, 50);

        frame.setLayout(null);
        frame.add(btn);

        frame.setVisible(true);
    }
}
```





# Labels

---

- ❑ A *label* is an object of the class **JLabel**
  - Text can be added to a **JFrame** using a label
  - The text for the label is given as an argument when the **JLabel** is created
  - The label can then be added to a **JFrame**

```
JLabel greeting = new JLabel("Hello");  
add(greeting);
```





# Lab

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MyFrame {

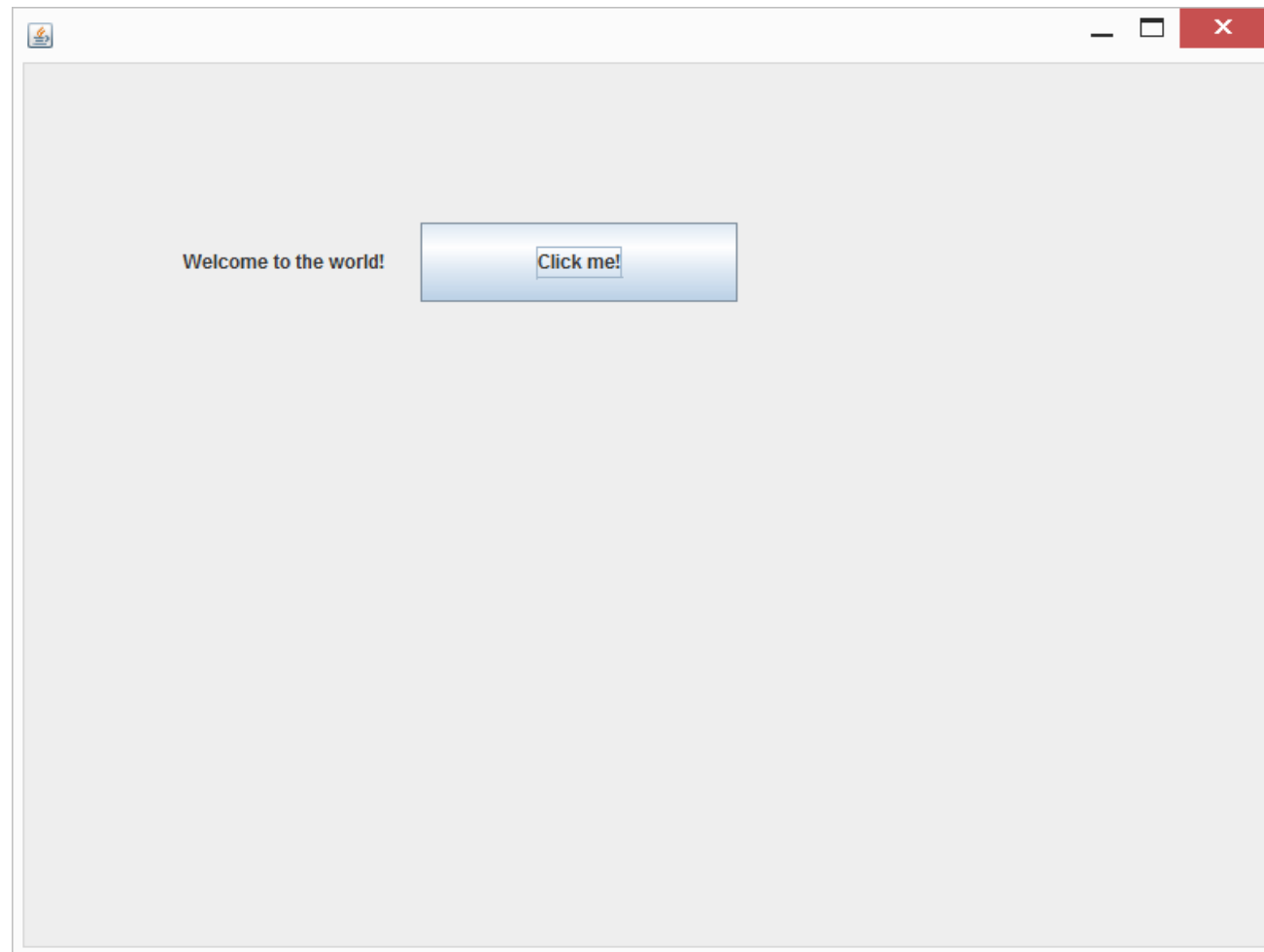
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(800, 600);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton btn = new JButton("Click me!");
        MyButtonListener mblister = new MyButtonListener();
        btn.addActionListener(mblister);
        btn.setLocation(250, 100);
        btn.setSize(200, 50);

        JLabel lb = new JLabel("Welcome to the world!");
        lb.setLocation(100,100);
        lb.setSize(200,50);

        frame.setLayout(null);
        frame.add(btn);
        frame.add(lb);

        frame.setVisible(true);
    }
}
```





# Color

---

- ❑ In Java, a *color* is an object of the class **Color**
  - The class **Color** is found in the **java.awt** package
  - There are constants in the **Color** class that represent a number of basic colors
- ❑ A **JFrame** can not be colored directly
  - Instead, a program must color something called the *content pane* of the **JFrame**
  - Since the content pane is the "inside" of a **JFrame**, coloring the content pane has the effect of coloring the inside of the **JFrame**
  - Therefore, the background color of a **JFrame** can be set using the following code:

```
getContentPane().setBackground(Color);
```



# The Color Constants

---

## Display 17.5 The Color Constants

---

<code>Color.BLACK</code>	<code>Color.MAGENTA</code>
<code>Color.BLUE</code>	<code>Color.ORANGE</code>
<code>Color.CYAN</code>	<code>Color.PINK</code>
<code>Color.DARK_GRAY</code>	<code>Color.RED</code>
<code>Color.GRAY</code>	<code>Color.WHITE</code>
<code>Color.GREEN</code>	<code>Color.YELLOW</code>
<code>Color.LIGHT_GRAY</code>	

The class `Color` is in the `java.awt` package.



# Lab

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MyFrame {

    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(800, 600);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton btn = new JButton("Click me!");
        MyButtonListener mblister = new MyButtonListener();
        btn.addActionListener(mblister);
        btn.setLocation(250, 100);
        btn.setSize(200, 50);

        JLabel lb = new JLabel("Welcome to the world!");
        lb.setLocation(100,100);
        lb.setSize(200,50);

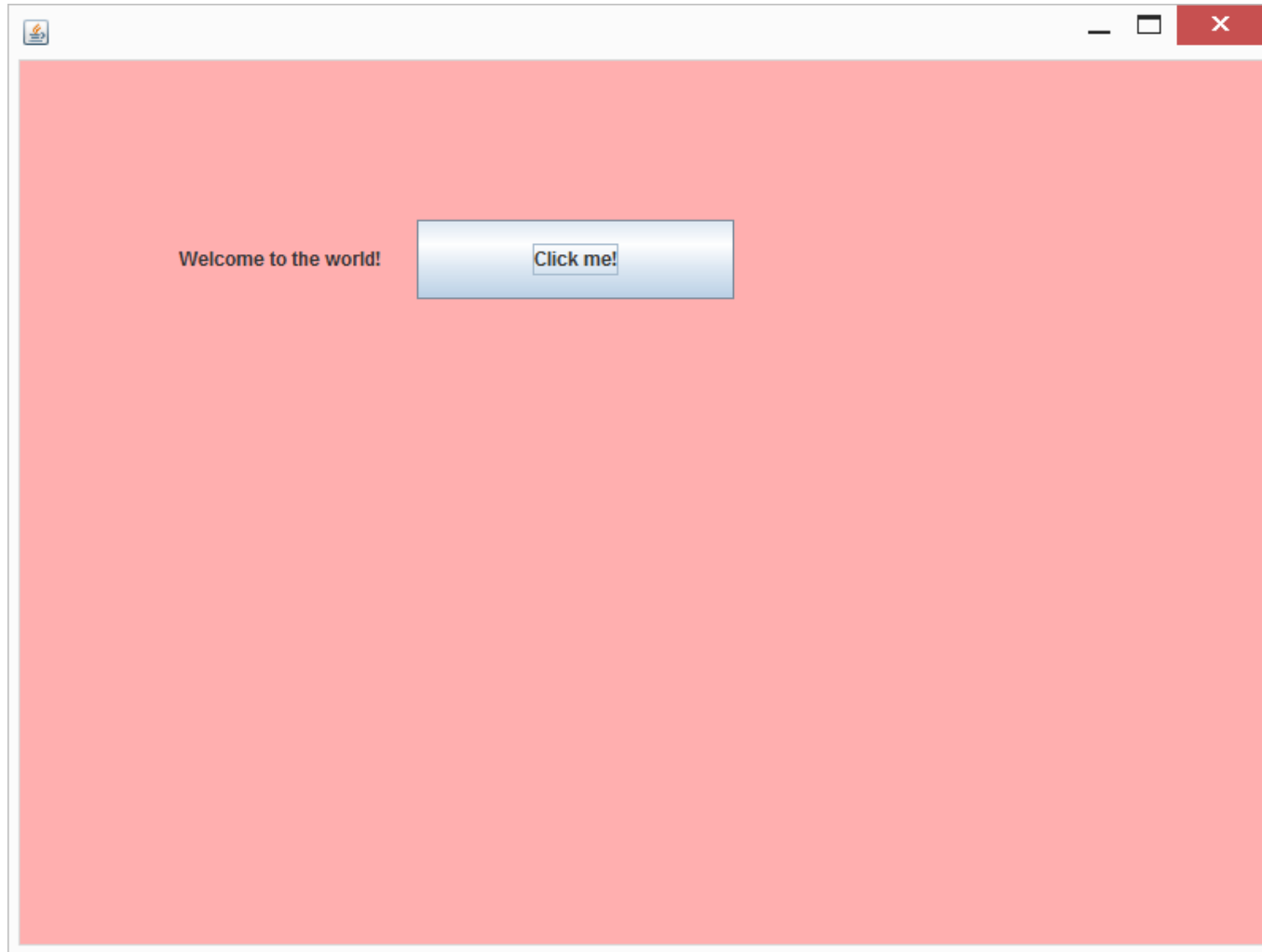
        frame.setLayout(null);
        frame.add(btn);
        frame.add(lb);

        frame.getContentPane().setBackground(Color.PINK);

        frame.setVisible(true);
    }
}
```



# Lab





# Panels

---

- ❑ A GUI is often organized in a hierarchical fashion, with containers called *panels* inside other containers
- ❑ A panel is an object of the **JPanel** class that serves as a simple container
  - It is used to group smaller objects into a larger component (the panel)
  - One of the main functions of a **JPanel** object is to subdivide a **JFrame** or other container



# Lab

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

public class PanelTest {

    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(800, 600);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton btn1 = new JButton("Click me!");
        JButton btn2 = new JButton("Click me!");

        JPanel panel = new JPanel();
        panel.setSize(200,200);
        panel.setLayout(new GridLayout());
        panel.add(btn1);
        panel.add(btn2);

        frame.setLayout(null);
        frame.add(panel);

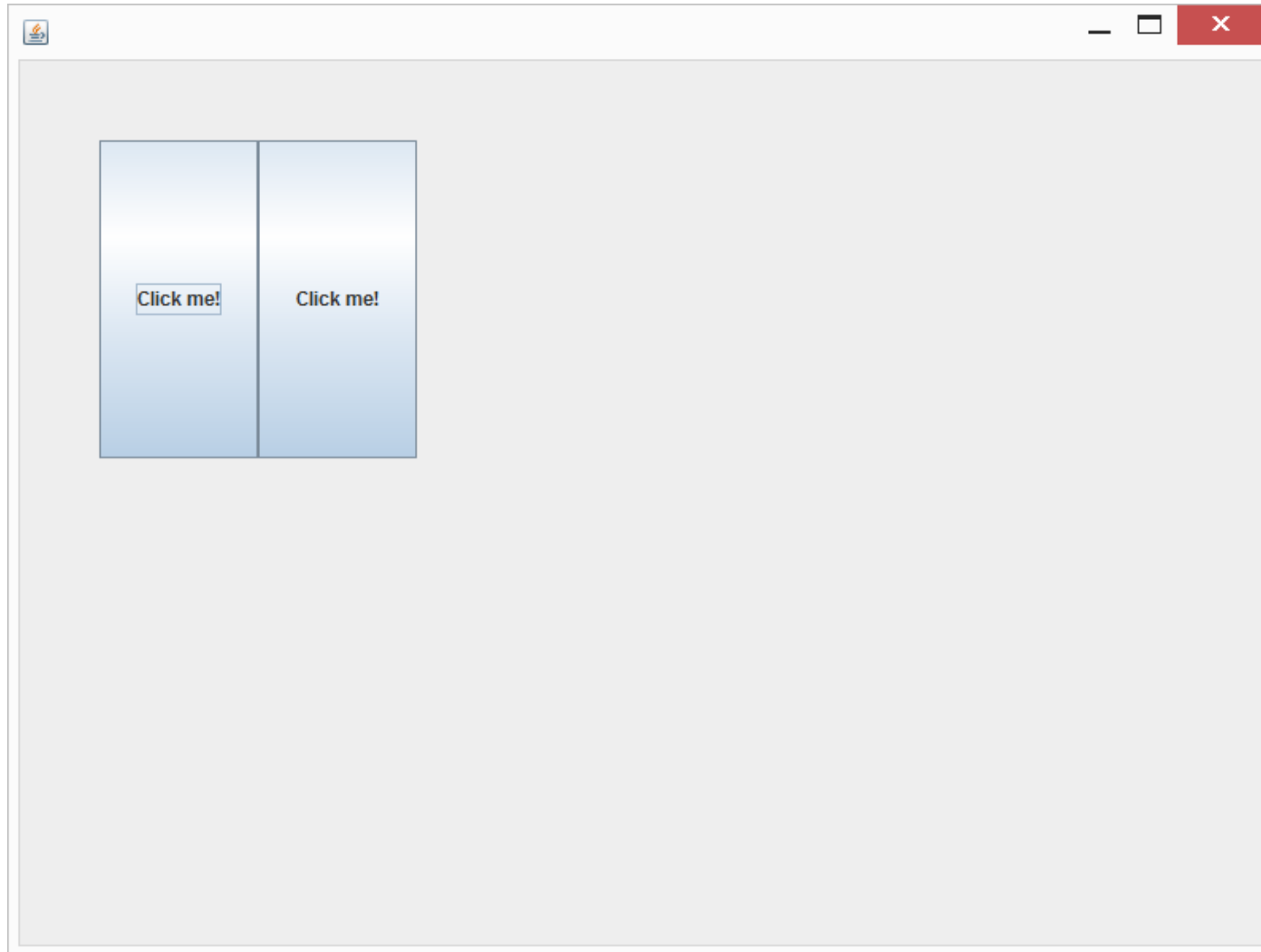
        frame.setVisible(true);

        for(int i=0;i<300;i++){
            panel.setLocation(i, i);
            try{
                Thread.sleep(100);
            }catch(Exception e){
                e.printStackTrace();
            }
        }
    }
}
```





# Lab





## Menu Bars, Menus, and Menu Items

---

- ❑ A *menu* is an object of the class **JMenu**
- ❑ A choice on a menu is called a *menu item*, and is an object of the class **JMenuItem**
  - A menu can contain any number of menu items
  - A menu item is identified by the string that labels it, and is displayed in the order to which it was added to the menu
- ❑ The **add** method is used to add a menu item to a menu in the same way that a component is added to a container object



## Menu Bars, Menus, and Menu Items

---

- ❑ The following creates a new menu, and then adds a menu item to it

```
JMenu diner = new  
    JMenu("Daily Specials");  
JMenuItem lunch = new  
    JMenuItem("Lunch Specials");  
lunch.addActionListener(this);  
diner.add(lunch);
```

- Note that the **this** parameter has been registered as an action listener for the menu item



# Nested Menus

---

- ❑ The class **JMenu** is a descendent of the **JMenuItem** class
  - Every **JMenu** can be a menu item in another menu
  - Therefore, menus can be nested
- ❑ Menus can be added to other menus in the same way as menu items



# Menu Bars and JFrame

---

- ❑ A *menu bar* is a container for menus, typically placed near the top of a windowing interface
- ❑ The **add** method is used to add a menu to a menu bar in the same way that menu items are added to a menu

```
JMenuBar bar = new JMenuBar();  
bar.add(diner);
```
- ❑ The menu bar can be added to a **JFrame** in two different ways
  1. Using the **setJMenuBar** method

```
setJMenuBar(bar);
```
  2. Using the **add** method – which can be used to add a menu bar to a **JFrame** or any other container



# Lab

```
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;

public class MenuTest {

    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(800, 600);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JMenu diner = new JMenu("Menu");
        JMenuItem item1 = new JMenuItem("MenuItem1");
        JMenuItem item2 = new JMenuItem("MenuItem2");
        item1.addActionListener(new MyButtonListener());
        item2.addActionListener(new MyButtonListener());
        diner.add(item1);
        diner.add(item2);
        JMenuBar bar = new JMenuBar();
        bar.add(diner);

        frame.setJMenuBar (bar);
        frame.setVisible(true);
    }
}
```



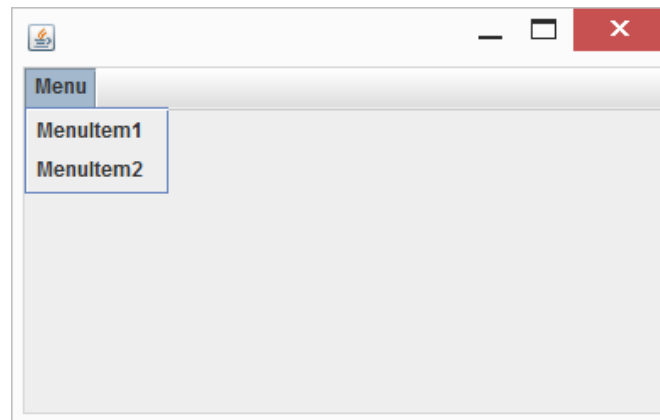
# Lab

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyButtonListener implements ActionListener{

    public void actionPerformed(ActionEvent e){

        String command = e.getActionCommand();
        if(command.equals("MenuItem1")){
            System.out.println("You pressed menuitem1");
        }else if(command.equals("MenuItem2")){
            System.out.println("You pressed menuitem2");
        }
    }
}
```





# Text Fields

---

❑ A *text field* is an object of the class **JTextField**

- It is displayed as a field that allows the user to enter a single line of text

```
private JTextField name;
```

```
. . .
```

```
name = new JTextField(NUMBER_OF_CHAR);
```

- In the text field above, at least **NUMBER\_OF\_CHAR** characters can be visible





# Text Fields

---

- ❑ There is also a constructor with one additional **String** parameter for displaying an initial **String** in the text field

```
JTextField name = new JTextField(  
    "Enter name here.", 30);
```

- ❑ A Swing GUI can read the text in a text field using the **getText** method

```
String inputString = name.getText();
```

- ❑ The method **setText** can be used to display a new text string in a text field

```
name.setText("This is some output");
```



# Lab

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JTextField;

public class MyNewFrame extends JFrame implements ActionListener{


    public static void main(String[] args) {
        MyNewFrame frame = new MyNewFrame();
        frame.setVisible(true);
    }

    public MyNewFrame(){
        setSize(800, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(null);

        JTextField input = new JTextField(50);
        input.setLocation(100,100);
        input.setSize(input.getPreferredSize());
        input.setText("<Input your name here>");
        add(input);
    }

    public void actionPerformed(ActionEvent e){
    }
}
```



—□✕



# Text Areas

---

- ❑ A *text area* is an object of the class **JTextArea**
  - It is the same as a text field, except that it allows multiple lines
  - Two parameters to the **JTextArea** constructor specify the minimum number of lines, and the minimum number of characters per line that are guaranteed to be visible

```
JTextArea theText = new JTextArea(5,20);
```

- Another constructor has one addition **String** parameter for the string initially displayed in the text area

```
JTextArea theText = new JTextArea(  
    "Enter\ntext here." 5, 20);
```



# Text Areas

---

- ❑ The line-wrapping policy for a **JTextArea** can be set using the method **setLineWrap**
    - The method takes one **boolean** type argument
    - If the argument is **true**, then any additional characters at the end of a line will appear on the following line of the text area
    - If the argument is **false**, the extra characters will remain on the same line and not be visible
- theText.setLineWrap(true);**



# Lab


```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class MyNewFrame extends JFrame implements ActionListener{
    public static void main(String[] args) {
        MyNewFrame frame = new MyNewFrame();
        frame.setVisible(true);
    }
    public MyNewFrame(){
        setSize(800, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(null);

        JTextField input = new JTextField(50);
        input.setLocation(100,100);
        input.setSize(input.getPreferredSize());
        input.setText("<Input your name here>");
        add(input);

        JTextArea inputlines = new JTextArea(10,50);
        inputlines.setLocation(100,200);
        inputlines.setSize(inputlines.getPreferredSize());
        inputlines.setText("<Input your message here>");
        add(inputlines);
    }
    public void actionPerformed(ActionEvent e){}
}
```



— □ ×



# Some Methods in the Class JTextComponent (Part 1 of 2)

## Display 17.18 Some Methods in the Class JTextComponent

All these methods are inherited by the classes JTextField and JTextArea. The abstract class JTextComponent is in the package `javax.swing.text`. The classes JTextField and JTextArea are in the package `javax.swing`.

```
public String getText()
```

Returns the text that is displayed by this text component.

```
public boolean isEditable()
```

Returns `true` if the user can write in this text component. Returns `false` if the user is not allowed to write in this text component.

(continued)





## Some Methods in the Class JTextComponent (Part 2 of 2)

### Display 17.18 Some Methods in the Class JTextComponent

```
public void setBackground(Color theColor)
```

Sets the background color of this text component.

```
public void setEditable(boolean argument)
```

If argument is true, then the user is allowed to write in the text component. If argument is false, then the user is not allowed to write in the text component.

```
public void setText(String text)
```

Sets the text that is displayed by this text component to be the specified text.



# Lab

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextArea;

public class MyDemoFrame extends JFrame implements ActionListener{
    private JTextArea inputlines;

    public static void main(String[] args) {
        MyDemoFrame frame = new MyDemoFrame();
        frame.setVisible(true);
    }
    public MyDemoFrame(){
        setSize(800, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(null);

        JButton btn = new JButton("Ok");
        btn.setLocation(100,400);
        btn.setSize(btn.getPreferredSize());
        btn.addActionListener(this);
        add(btn);

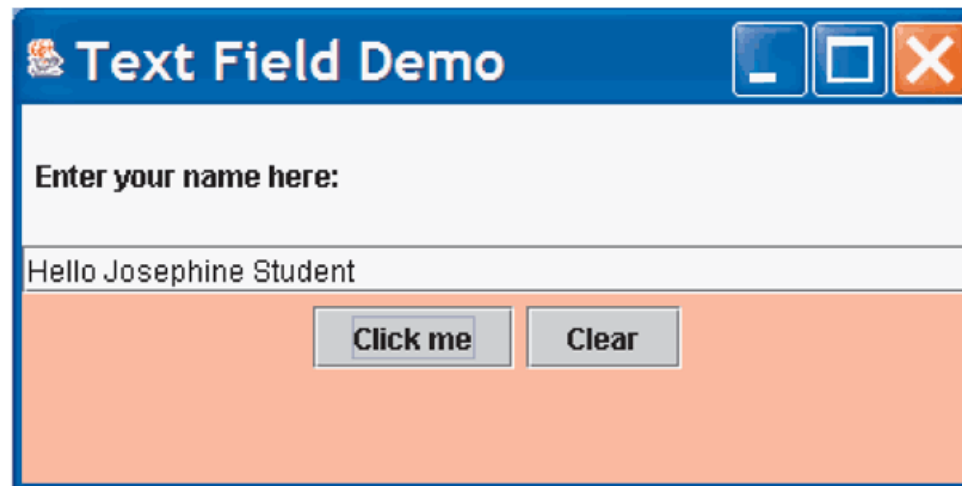
        inputlines = new JTextArea(10,50);
        inputlines.setLocation(100,200);
        inputlines.setSize(inputlines.getPreferredSize());
        inputlines.setText("<Input your message here>");
        add(inputlines);
    }
    public void actionPerformed(ActionEvent e){
        if(e.getActionCommand().equals("Ok")){
            System.out.println(inputlines.getText());
        }
    }
}
```



# Lab

---

Write a program to present a frame like the following GUI

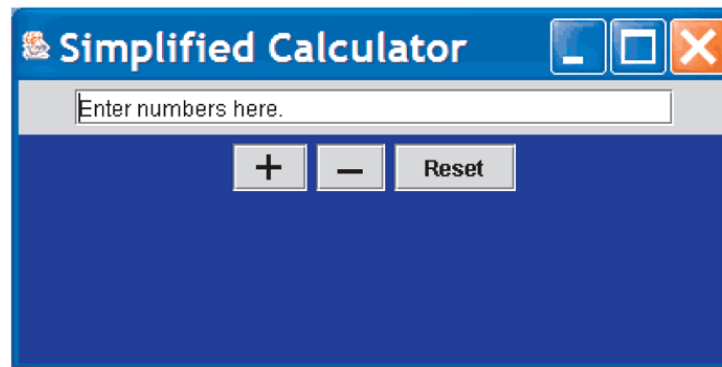




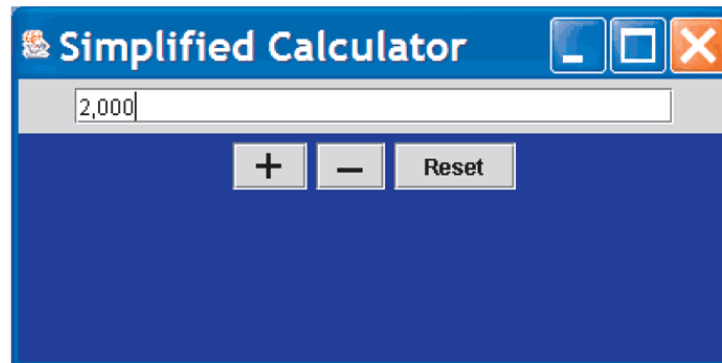
# Lab

Write a program to present a frame like the following GUI

**RESULTING GUI** (When started)



**RESULTING GUI** (After entering 2,000)





## Reference

---

- ❑ “Absolute Java”. Walter Savitch and Kenrick Mock. Addison-Wesley; 5 edition. 2012
- ❑ “Java How to Program”. Paul Deitel and Harvey Deitel. Prentice Hall; 9 edition. 2011.
- ❑ “A Programmers Guide To Java SCJP Certification: A Comprehensive Primer 3rd Edition”. Khalid Mughal, Rolf Rasmussen. Addison-Wesley Professional. 2008