The Java LayoutManagers (Outsource: 12-31 – 12-36)

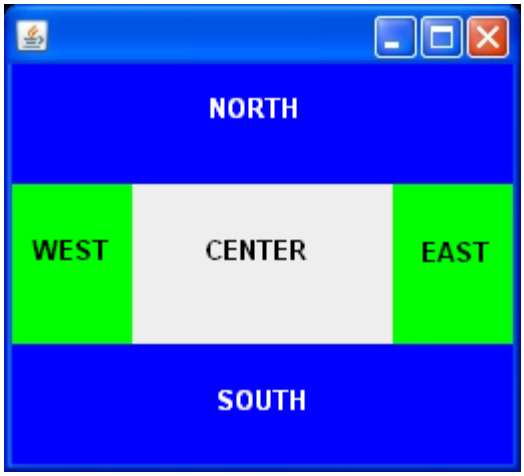
The Java LayoutManagers are a collection of classes that know how to lay out `Containers`. There are quite a number of these classes. We will look at a few of the more common LayoutManagers.

BorderLayout (Outsource: 12-33 – 12-34)

A border layout lays out a container, arranging and resizing its components to fit in five regions: north, south, east, west, and center. Each region may contain no more than one component, and is identified by a corresponding constant: NORTH, SOUTH, EAST, WEST, and CENTER. When adding a component to a container with a border layout, you must use an add( ) method that takes the object to be added as the first argument, and one of these five constants as a second argument. For example:

```
JPanel northPanel = new JPanel();
add(northPanel, BorderLayout.NORTH);
```

The following example shows a JFrame containing five JPanel's, one in each area.



For every placement but CENTER, the element that you add is compressed to fit in the smallest amount of space along one dimension while it is stretched to the maximum along the other dimension. CENTER, however, spreads out along both dimensions to occupy the middle.

BorderLayout is the default LayoutManager for Frames and Dialogs.

FlowLayout (Outsource: 12-32 – 12-33)

A flow layout arranges components in a directional flow, much like lines of text in a paragraph. The flow direction is determined by the container's componentOrientation property and may be one of two values:

- ComponentOrientation.LEFT_TO_RIGHT
- ComponentOrientation.RIGHT_TO_LEFT

Flow layouts are typically used to arrange buttons in a panel. It arranges buttons horizontally until no more buttons fit on the same line. The line alignment is determined by the `align` property. The possible values are:

- LEFT
- RIGHT
- CENTER
- LEADING
- TRAILING

The following example shows a JFrame containing a JPanel. Three JButtons have been added to the JPanel using the flow layout manager (its default layout manager) to position the three buttons.

FlowLayout is the default LayoutManager for the JPanel class.

GridLayout (Outsource: 12-34 – 12-35)

The GridLayout class is a layout manager that lays out a container's components in a rectangular grid. The container is divided into equal-sized rectangles, and one component is placed in each rectangle. The following example shows a JFrame using a GridLayout to position 9 JButtons into three rows and two columns:

There are numerous other LayoutManagers. Read the LayoutManager help file to find out about the others.

Component (Outsource: 12-37)

A *component* is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of components are the buttons, textfields, checkboxes, and scrollbars of a typical graphical user interface.

Method Summary	
Color	**getBackground**() Gets the background color of this component.
Graphics	**getGraphics**() Returns this component's graphics context, which lets you draw on a component.
int	**getHeight**() Returns the current height of this component.
String	**getText()** Returns the text contained in this TextComponent.
int	**getWidth**() Returns the current width of this component.
void	**setBackground**(Color bg) Sets the background color of this component.
void	**setForeground**(Color fg) Sets the foreground color of this component.
void	**setBorder**(Border border) Sets the border of this component.
void	**setEnabled**(boolean enabled) Sets whether or not this component is enabled.
void	**setFont**(Font f) Sets the font for this component.
void	**setPreferredSize**(Dimension preferredSize) Sets the preferred size of this component.
void	**setText**(String t) Sets the text of this TextComponent to the specified text.
void	**setVisible**(boolean aFlag) Makes the component visible or invisible.

JButton (Outsource: 12-37)

An implementation of a "push" button. Buttons can be configured, and to some degree controlled, by Actions. To configure a button to respond to user input (i.e. mouse clicks) you must add an `ActionListener` to each button.

Constructor Summary

`JButton``()` Creates a button with no set text or icon.
`JButton``(Action a)` Creates a button where properties are taken from the `Action` supplied.
`JButton``(Icon icon)` Creates a button with an icon.
`JButton``(String text)` Creates a button with text.
`JButton``(String text, Icon icon)` Creates a button with initial text and an icon.

Method Summary

`void`	**`addActionListener`**`(ActionListener l)` Adds an `ActionListener` to the button.
Icon.	**`getIcon`**`()` Returns the default icon.
void	**`setIcon`**`(Icon defaultIcon)` Sets the button's default icon.

JTextField (Outsource: 12-39)

`JTextField` is a lightweight component that allows the editing of a single line of text.

Constructor Summary

`JTextField``()` Constructs a new `TextField`.
`JTextField``(Document doc, String text, int columns)` Constructs a new `JTextField` that uses the given text storage model and the given number of columns.
`JTextField``(int columns)` Constructs a new empty `TextField` with the specified number of columns.

JTextField(String text) Constructs a new TextField initialized with the specified text.
JTextField(String text, int columns) Constructs a new TextField initialized with the specified text and columns.

Method Summary

String	**getSelectedText()** Returns the selected text contained in this TextComponent.
void	**setEditable**(boolean b) Sets the specified boolean to indicate whether or not this TextComponent should be editable.
void	**setHorizontalAlignment**(int alignment) Sets the horizontal alignment of the text. Valid keys are: • JTextField.LEFT • JTextField.CENTER • JTextField.RIGHT

JLabel (Outsource: 12-38)

A JLabel object can display either text, an image, or both. You can specify where in the label's display area the label's contents are aligned by setting the vertical and horizontal alignment. By default, labels are vertically centered in their display area. Text-only labels are leading edge aligned, by default; image-only labels are horizontally centered, by default.

Constructor Summary

JLabel() Creates a JLabel instance with no image and with an empty string for the title.
JLabel(Icon image) Creates a JLabel instance with the specified image.
JLabel(Icon image, int horizontalAlignment) Creates a JLabel instance with the specified image and horizontal alignment.
JLabel(String text) Creates a JLabel instance with the specified text.
JLabel(String text, Icon icon, int horizontalAlignment) Creates a JLabel instance with the specified text, image, and horizontal alignment.
JLabel(String text, int horizontalAlignment) Creates a JLabel instance with the specified text and horizontal alignment.

Method Summary	
Icon.	**getIcon**() Returns the default icon.
void	**setHorizontalAlignment**(int alignment) Sets the horizontal alignment of the text. Valid keys are: • SwingConstants.LEFT • SwingConstants.CENTER • SwingConstants.RIGHT
void	**setIcon**(Icon defaultIcon) Sets the button's default icon.

Menus

There are three steps involved in adding menus to a `JFrame` object.

1. Instantiate an instance of a `JMenuBar`
2. Instantiate an instance of a `JMenu` and add it to the `JMenuBar`
3. Instantiate an instance of a `JMenuItem` and add it to the `JMenu`. (Every instance of a `JMenuItem` must be assigned an `ActionListener`)

The following example demonstrates the addition of a menu to a `JFrame` object:

```
JMenuBar bar = new JMenuBar();              // Instantiate a JMenuBar
JMenu file = new JMenu("File");             // Instantiate a JMenu called "File"
JMenuItem open = new JMenuItem("Open"); // Instantiate a JMenuItem called "Open"
open.addActionListener(new ActionListener() {  // Add an ActionListener to open
   public void actionPerformed(ActionEvent e) {
      // code defining the action to be performed goes here
   } });
file.add(open);      // Add open to the file menu
bar.add(file);       // Add file to the JMenuBar
setJMenuBar(bar);    // Sets the JMenuBar for this JFrame
```

What is an Interface (Outsource: 9-16)

Within the Java programming language, an interface is a device that unrelated objects use to interact with each other. An interface is probably most analogous to a protocol (an agreed on behavior). An interface defines a set of methods but does not implement them. A class that implements the interface agrees to implement all the methods defined in the interface, thereby agreeing to certain behavior. In other words, an *interface* is a named collection of method definitions (without implementations). An interface can also declare constants.

Implementing an Interface (Outsource: 11-40 – 11-42)

To use an interface, you write a class that *implements* the interface. When a class claims to implement an interface, the class is claiming that it provides a method implementation for all of the methods declared within the interface (and its superinterfaces).

When a class implements an interface, it is essentially signing a contract. Either the class must implement all the methods declared in the interface and its superinterfaces, or the class must be declared `abstract`. The method signature--the name and the number and type of arguments in the class--must match the method signature as it appears in the interface.

It is possible to simultaneously extend another class and implement an interface. It is possible for a particular class to only extend a single class, however it can implement as many interfaces as desired.

Implementing an Interface is an example of `Polymorphism`. Polymorphism is the property of being able to have methods with the same name, while having possibly different implementations. Therefore, interfaces are polymorphic in nature.

The ActionListener Interface (Outsource: 12-67 – 12-68)

The Java standard class library contains a number of important interfaces. `ActionListener` is the interface for receiving action events. Any class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's `addActionListener` method. When the action event occurs, that object's `actionPerformed` method is invoked.

The `ActionListener` interface contains only one method, `actionPerformed`, which receives an `ActionEvent` object as an argument.

Method Summary

`void`	**`actionPerformed`**`(ActionEvent e)` Invoked when an action occurs.

When the user clicks a `button`, chooses a `menu` item or presses `ENTER` in a text field, an action event occurs. The result is that an `actionPerformed` message is sent to all action listeners that are registered on the relevant component.

Here is the action event handling code from an application named Beeper:

```
public class Beeper
{
   private JButton button = new JButton("OK");

   public Beeper() {
       //adding an ActionListener to the JButton:
       button.addActionListener(new ActionListener() {
           public void actionPerformed(ActionEvent e) {
               // code defining the action to be performed goes here
               Toolkit.getDefaultToolkit().beep();
           }
   });
}
```

Lab 21 - ASSIGNMENT

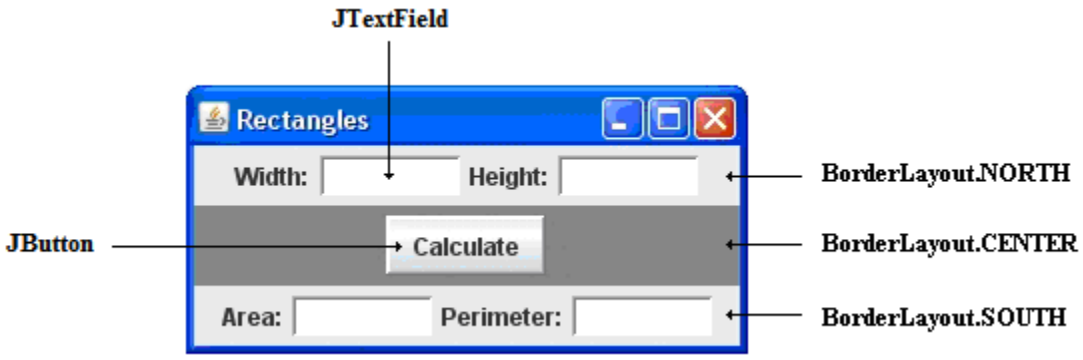
Lab 21A - 60 points

PROGRAM OBJECTIVE

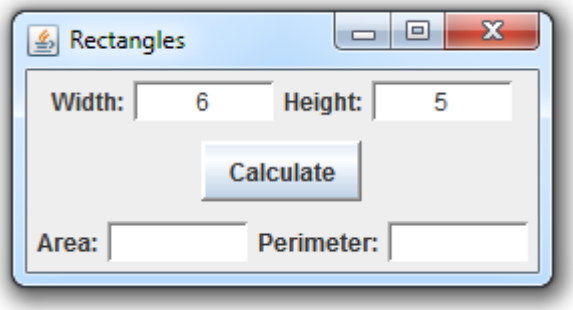
WAP using a `GUI` interface that allows the user to enter the width and height of a rectangle and calculates the resulting area and perimeter when the user clicks the `Calculate` button.

LAYOUT SUMMARY

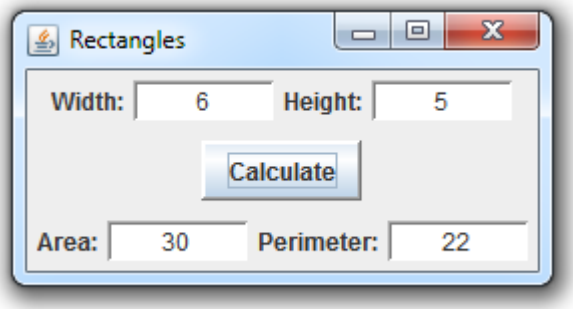
Place `JPanel` objects in the NORTH, CENTER, and SOUTH quadrants of the `JFrame`. Add JTextFields, JButtons, and JLabels to the appropriate panels.



SAMPLE OUTPUT



`after entering the width and height`



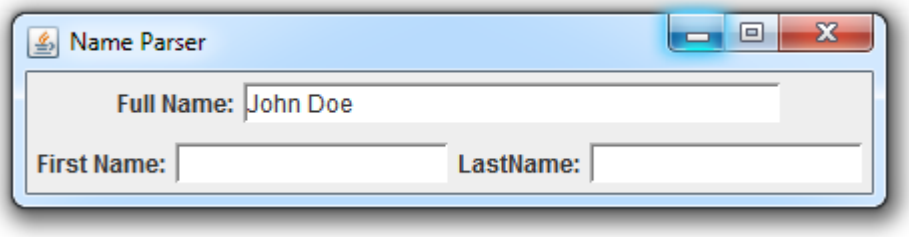
after clicking on ***Calculate***

Lab 21B - 70 points

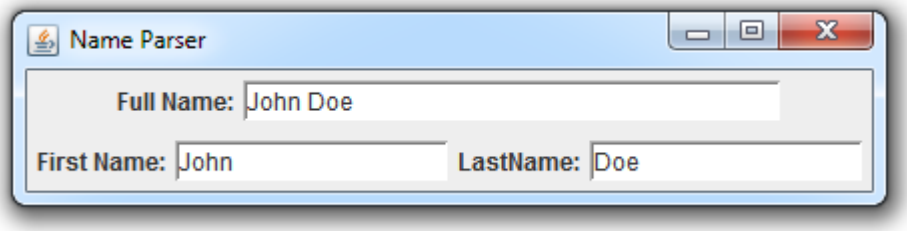
PROGRAM OBJECTIVE

WAP using a `GUI` interface that allows the user to enter his/her full name (first and last name) into a text field. When the user presses **ENTER** the program should parse the full name into two parts – first name and last name. Display the first name and the last name in two separate text fields.

SAMPLE OUTPUT



before pressing the **ENTER** key



after pressing the **ENTER** key

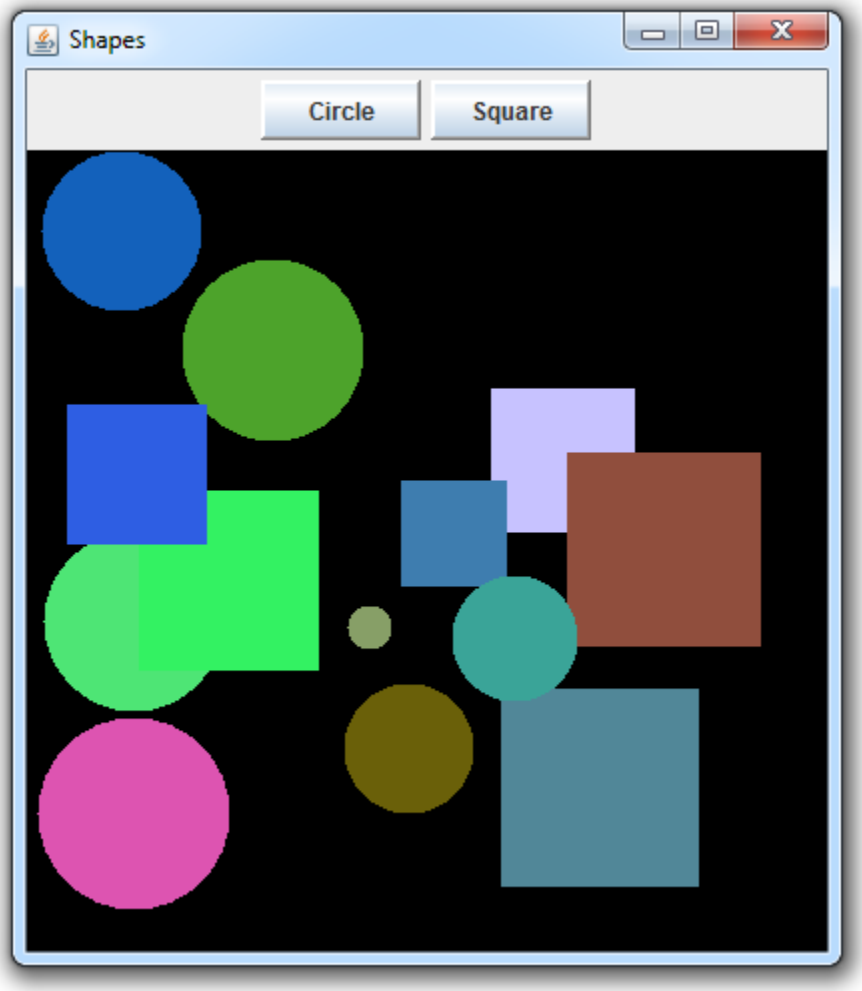
Lab 21C - 80 points

PROGRAM OBJECTIVE

WAP using a GUI interface that allows the user to draw filled circles (fillOval) and squares (fillRect) at random locations on the screen.

Circle – draw a circle (using a random color) at a random screen location. Use Math.random to generate random x and y values for the location. You can also use Math.random to generate separate red, green, and blue (RGB) colors for a random color.
Square – draw a square (using a random color) at a random screen location. Use Math.random to generate random x and y values for the location. You can also use Math.random to generate separate red, green, and blue (RGB) colors for a random color.

SAMPLE OUTPUT

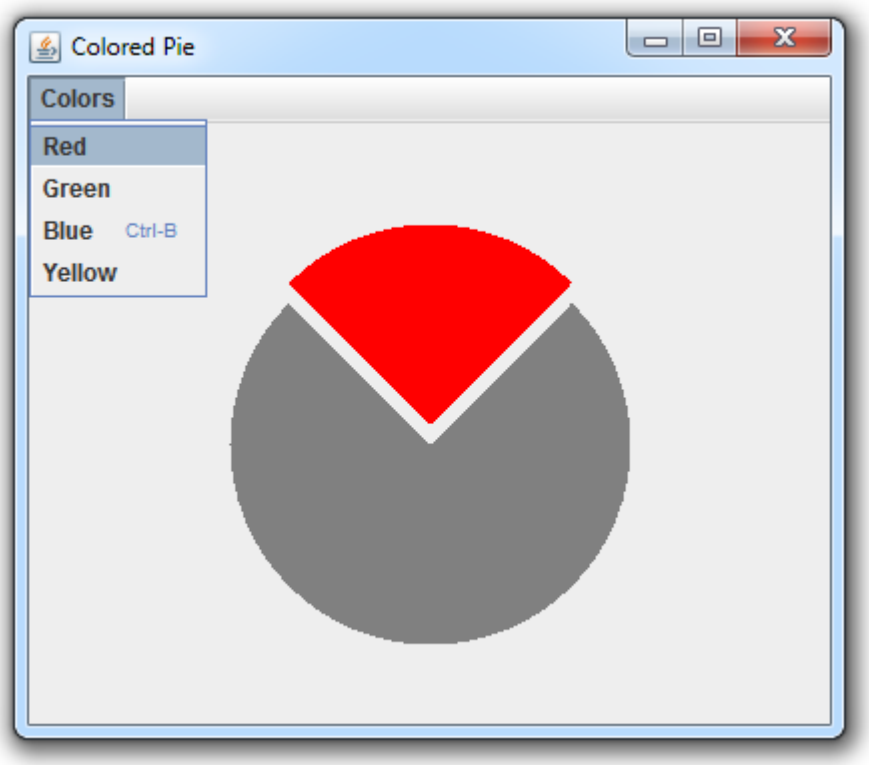


Lab 21D - 90 points

PROGRAM OBJECTIVE

WAP using a `GUI` interface that allows the user to change the color of a slice out of a pie. When the user selects a color from the menu (Red, Green, Blue, and Yellow) the slice of the pie should change to the selected color. You have been provided with a `PiePanel` class that draws the pie. Set the content pane of your JFrame to be a new instance of a `PiePanel`. The `PiePanel` class has a `setSliceColor` method that receives a `Color` argument. You will only need to implement the main class.

SAMPLE OUTPUT



Lab 21E - 100 points

PROGRAM OBJECTIVE

WAP using a `GUI` interface that allows the user to change the direction a cannon is facing and/or fire the cannon using a JToolbar.

To use a **`Tool Bar,`** the **`Layout`** for the frame needs to be a **`BorderLayout`** (which is the default layout). Instantiate a JToolbar object, populate it with three JButtons (Left, Right, and Fire). You have been provided with images for each button. Each JButton will require an ActionListener. Add the JToolbar to the JFrame in the NORTH quadrant. Add an instance of the CannonPanel to the CENTER quadrant.

The `CannonPanel` class contains three methods you can call to affect the behavior of the cannon.

Method Summary - CannonPanel	
void	**fire()** Fires the cannon.
void	**turnLeft()** Makes the cannon face left.
void	**turnRight()** Makes the cannon face right.

SAMPLE OUTPUT