

Graphical User Interfaces

Michael C. Hackett
Computer Science Department

Community
College
of Philadelphia

Contents

- Introduction
- Dialog Boxes
- Creating Windows
 - Frames
 - Panels
- Labels
- Buttons
 - Action Events and Listeners
 - Action Commands
- Text Fields
- Radio Buttons
 - Button Groups
- Check Boxes
- Sliders
- Text Areas
- Layout Managers
 - FlowLayout
 - BorderLayout
 - GridLayout

Contents

- Menu Systems
 - Menu Bars
 - Menus and Menu Items
 - Submenus
- Mnemonics
- Fonts
- 2-D Graphics
 - Canvas and Graphics objects
 - Drawing Lines
 - Drawing Rectangles
 - Drawing Ovals
 - Drawing Arcs
 - Drawing Polygons
 - Drawing Text

Colors/Fonts

• Local Variable Names	—	Brown
• Primitive data types	—	Fuchsia
• Literals	—	Blue
• Keywords	—	Orange
• Object names	—	Green
• Operators/Punctuation	—	Black
• Field Names	—	Lt Blue
• Method Names	—	Purple
• Parameter Names	—	Gold
• Comments	—	Gray
• Package Names	—	Pink

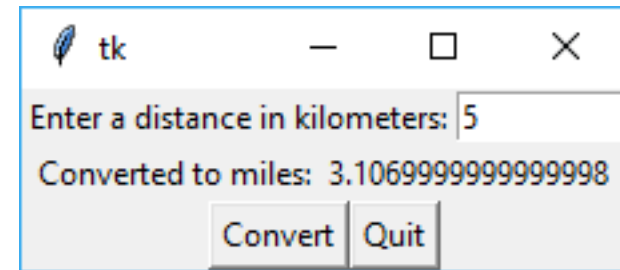
Source Code — **Consolas**
Output — Courier New

Graphical User Interfaces

- A **graphical user interface** (*GUI* or “*gooey*”) allows a user to interact with a program using pictures, icons and other visual components.
 - As opposed to programs using a *command line interface* (*CLI*).

```
===== RESTART: C:/testing/kiloprogram.py :  
Enter a distance in kilometers: 5  
Converted to miles: 3.1069999999999998  
>>> |
```

Command Line Interface



Graphical User Interface

Graphical User Interfaces

- Prior to GUIs, all work on a computer was done using a CLI.
 - This made it very difficult for new computers users.
 - Commands needed to be memorized and entered using a keyboard.

```

IBM Personal Computer DOS-Version 3.00

A:>dir /w

   Diskette/Platte, Laufwerk A:, hat keinen Namen
Verzeichnis von A:\

COMMAND  COM      CONFIG  SYS      AUTOEXEC  BAT      ANSI      SYS      SORT      EXE
SHARE    EXE      FIND     EXE      ATTRIB    EXE      MORE      COM      ASSIGN    COM
PRINT    COM      SYS      COM      CHKDSK    COM      FORMAT    COM      VDISK     SYS
BASIC    COM      BASICA   COM      FDISK     COM      COMP      COM      TREE      COM
BACKUP    COM      RESTORE  COM      LABEL     COM      DISKCOPY   COM      DISKCOMP  COM
KEYBSP    COM      KEYBIT   COM      KEYBGR    COM      KEYBUK     COM      KEYBFR    COM
MODE      COM      SELECT   COM      GRAPHICS  COM      RECOVER    COM      EDLIN     COM
GRAFTABL  COM

   36 Datei(en)      100352 Byte frei

A:>chkdsk

 362496 Byte Gesamtkapazität
 37888 Byte in 2 geschützten Dateien
224256 Byte in 36 Benutzerdatei(en)
100352 Byte auf Diskette/Platte
verfügbar

524288 Byte Gesamtspeicher
435072 Byte frei

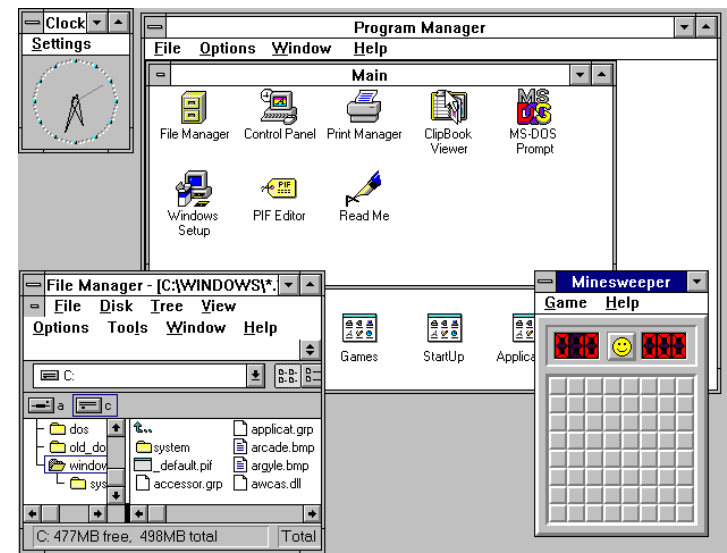
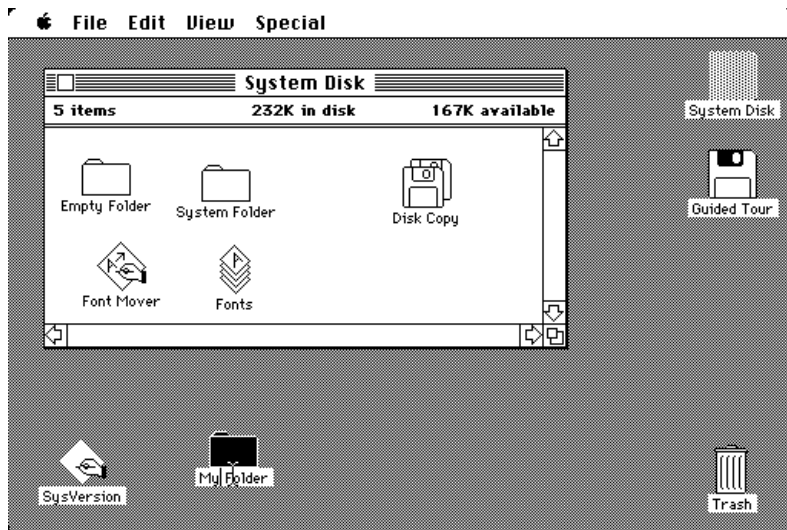
A:>_

```

```
nuucp:x:9:9:uucp Admin:/var/spool/uucppublic:/usr/lib/uucp/uucico
snmnp:x:25:25:SendMail Message Submission Program:::
listen:x:37:4:Network Admin:/usr/net/nls:
gdm:x:50:50:GDM Reserved UID:::
webservd:x:80:80:WebServer Reserved UID:::
postgres:x:90:90:PostgreSQL Reserved UID::/usr/bin/pfksd
svctag:x:95:12:Service Tag UID:::
nobody:x:60001:60001:NFS Anonymous Access User:::
noaccess:x:60002:60002:No Access User:::
nobody4:x:65534:65534:SunOS 4.x NFS Anonymous Access User:::
~
~
~
~
~
~/etc/passwd" 17 lines, 677 characters
# ^D
testimage console login: root
Password:
Mar 29 11:36:16 testimage login: ROOT LOGIN /dev/console
Last login: Sat Mar 29 11:04:43 on console
Sun Microsystems Inc. SunOS 5.10 Generic January 2005
-bash-3.00#
```

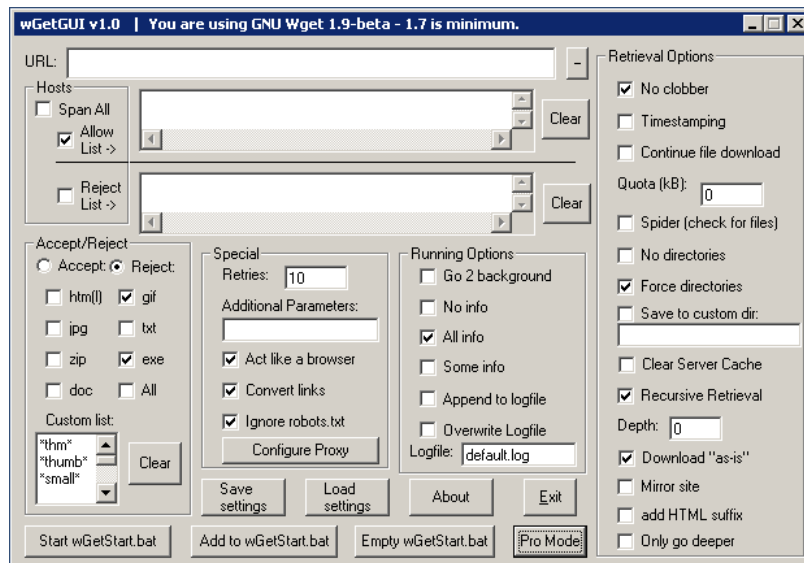
Graphical User Interfaces

- With a GUI, users did not need to remember commands to use the operating system.
 - They could use a mouse to click buttons to perform the commands and click icons to open programs.

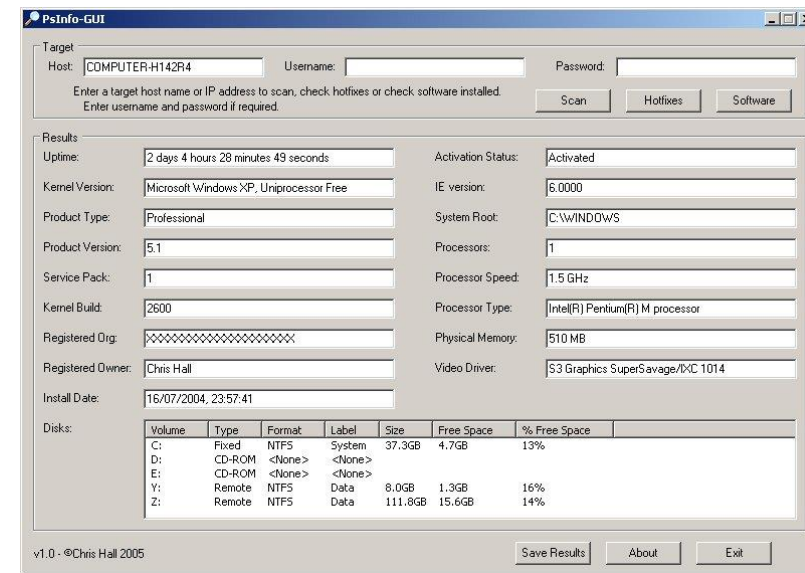


Graphical User Interfaces

- A successful GUI is one that is *user-friendly*.
 - The interface is intuitive, organized and familiar.
 - The user should not feel overwhelmed or confused.



Cluttered GUI



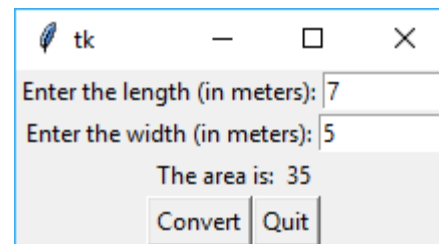
Clean GUI

Graphical User Interfaces

- Events in a command line program are predetermined.
 - The user enters a length, then enters a width and the area is printed.

```
Enter the length (in meters): 7
Enter the width (in meters): 5
The area is 35 meters.
>>>
```

- In a graphical program, the user determines the order of events.
 - The user could enter the length first, or the width first.
 - The area is not displayed until the user clicks the Convert button



Event-Driven Programming

- Unlike CLI programs, GUI programs are ***event-driven***.
 - The user ***triggers*** events in the program to happen.
 - Clicking buttons.
 - Checking check boxes.
 - Selecting an item from a menu or list.
 - A ***listener*** is an object that that executes code when a particular event has been triggered.
- A GUI program, essentially, exists in a loop.
 - The loop keeps the GUI alive/running and waits for events to occur.
 - It then executes the code associated with a particular event.

Abstract Windowing Toolkit (AWT)

- The AWT allows for the creation of Java applications with GUI components.
 - But... it doesn't actually *draw* user interface components on the screen.
- The AWT communicates with a layer of software called *peer classes*.
- Each version of Java for a particular operating system has its own set of peer classes.

Abstract Window Toolkit (AWT)

- Java programs using the AWT...
 - Look consistent with other applications on the same system.
 - Offer only components that are common to all the operating systems that support Java.
- The behavior of components across various operating systems can differ.
- There are some limitations to the AWT.
 - Programmers cannot easily extend the AWT components.
 - AWT components are commonly called *heavyweight components* as they rely heavily on peer classes.

Swing

- *Swing* is a library of classes that provide an improved alternative to AWT for creating GUI applications and applets.
 - Introduced with the release of Java 2.
- Very few Swing classes rely on peer classes, so they are referred to as *lightweight components*.
- Swing draws most of its own components.
 - Swing components have a consistent look and predictable behavior on any operating system.
 - Swing components can be easily extended.

JavaFX

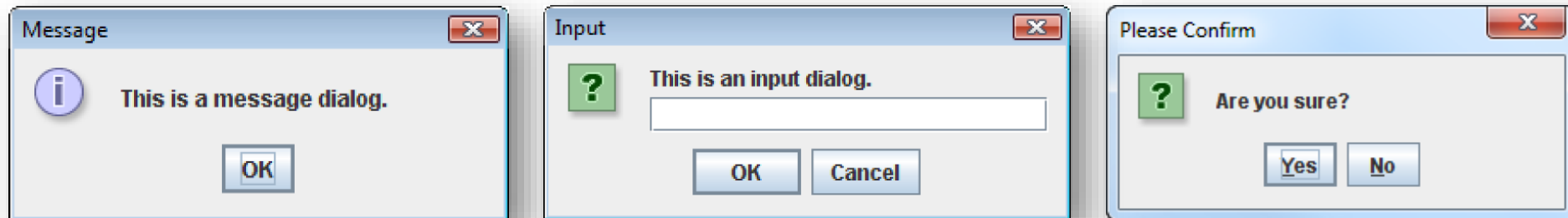
- *JavaFX* was to be the successor to the Swing libraries.
 - First released in 2008
 - Included as part of the JDK with Java 7 (2012)
 - Decoupled from the JDK since Java 11 (2018)
 - “OpenJFX” (I’m not optimistic about it)
- Swing will not be obsolete for the foreseeable future.
 - Once you have a handle on Swing, learning JavaFX isn’t very difficult.
 - It’s good for introducing GUI development concepts.

Dialog Boxes

- A dialog box is a small graphical window that displays a message to the user or requests input.
- A variety of dialog boxes can be displayed using Swing's JOptionPane class:
 - Message Dialog - A dialog box that displays a message.
 - Input Dialog - A dialog box that prompts the user for input.
 - Confirmation Dialog - A dialog box that asks the user a Yes/No question.

Dialog Boxes

- The **JOptionPane** class provides static methods to display each type of dialog box.



- Must be imported: `import javax.swing.JOptionPane;`

Message Dialogs

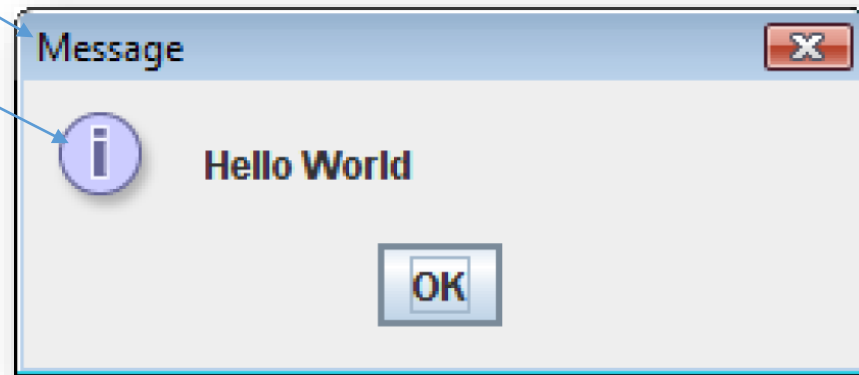
```
JOptionPane.showMessageDialog(null, "Hello World");
```

- The first argument can be a reference to a graphical component.
 - The dialog box is displayed atop the center of that component.
 - If null is passed as the first argument, it causes the dialog box to be displayed in the center of the screen.
- The second argument is the message that is to be displayed.



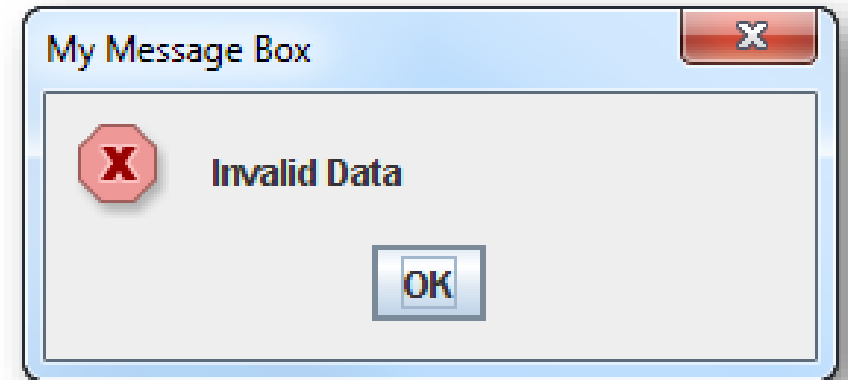
Message Dialogs

- By default a message dialog box has:
 - The string “Message” displayed in its title bar
 - An information icon.



Message Dialogs

- Two additional arguments can specify:
 - The title bar's text.
 - The icon.



```
JOptionPane.showMessageDialog(null,  
    "Invalid Data",  
    "My Message Box",  
    JOptionPane.ERROR_MESSAGE);
```

Message Dialogs

- These constants can be use to control the icon that is displayed.

`JOptionPane.ERROR_MESSAGE`

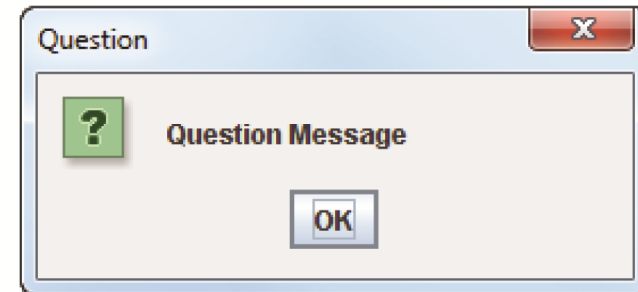
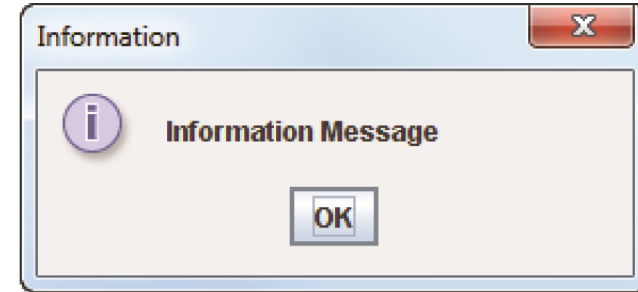
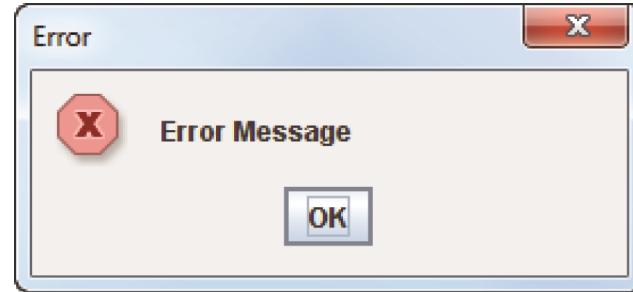
`JOptionPane.INFORMATION_MESSAGE`

`JOptionPane.WARNING_MESSAGE`

`JOptionPane.QUESTION_MESSAGE`

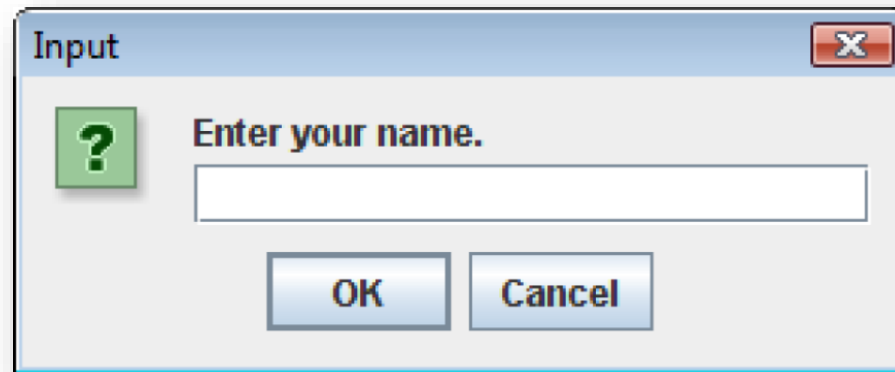
`JOptionPane.PLAIN_MESSAGE`

Message Dialogs



Input Dialogs

- An input dialog is a small window that asks the user to enter data.

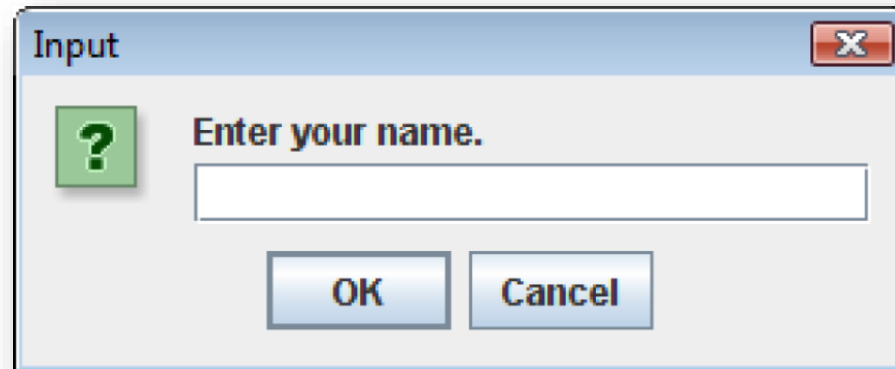


- The dialog displays a message, a text field, an OK button, and a Cancel button.
 - If OK is pressed, the dialog returns the user's input **as a String**.
 - If Cancel is pressed, the dialog returns null.

Input Dialogs

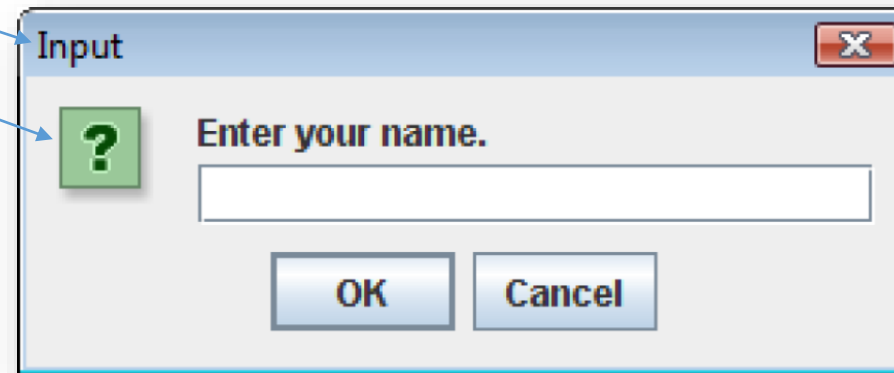
```
String name;  
name = JOptionPane.showInputDialog("Enter your name.");
```

- The argument passed to the method is the message to display.
 - If the user clicks on the OK button, name references the string entered by the user.
 - If the user clicks on the Cancel button, name references null.



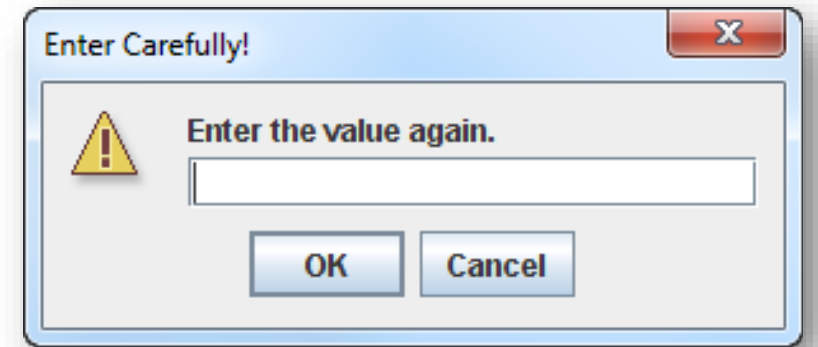
Input Dialogs

- By default an input dialog box has:
 - The string “Input” displayed in its title bar
 - A question icon.



Input Dialogs

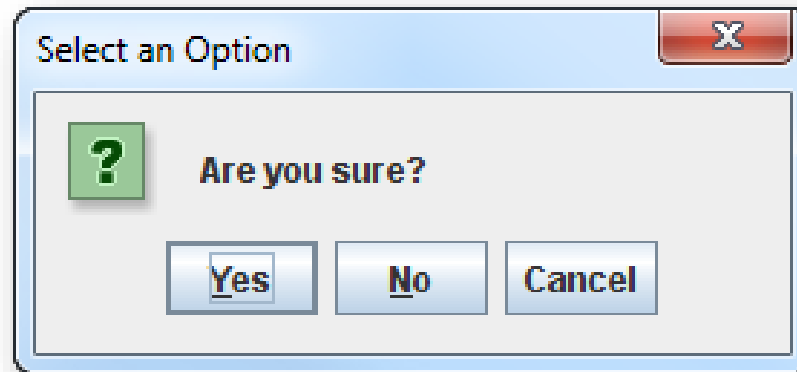
- Three more arguments specify:
 - The location where the input dialog appears.
 - The title bar's text.
 - The icon.



```
String name;  
name = JOptionPane.showInputDialog(null,  
    "Enter the value again.",  
    "Enter Carefully!",  
    JOptionPane.WARNING_MESSAGE);
```

Confirmation Dialogs

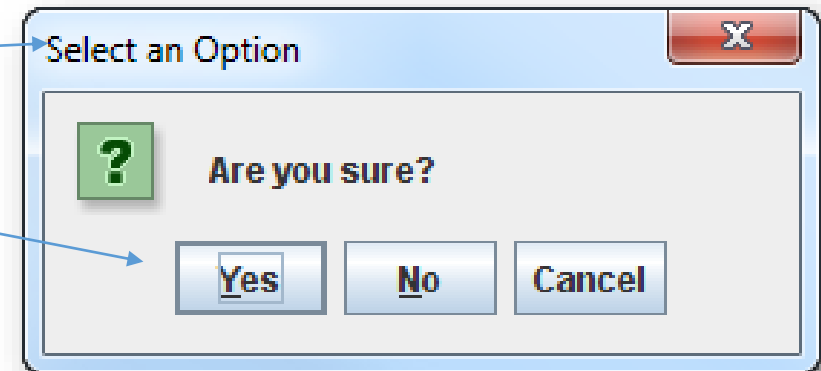
- A confirmation dialog box typically asks the user a yes or no question.
 - By default Yes, No, and Cancel buttons are displayed.



Confirmation Dialogs

```
int answer;  
answer = JOptionPane.showConfirmDialog(null, "Are you sure?");
```

- By default the confirmation dialog box displays:
 - “Select an Option” in its title bar,
 - Yes, No, and Cancel buttons.



Confirmation Dialogs

- The `showConfirmDialog` method returns an `int` that represents the button clicked by the user.
- The button that was clicked is determined by comparing the method's return value to one of the following constants:

`JOptionPane.YES_OPTION`

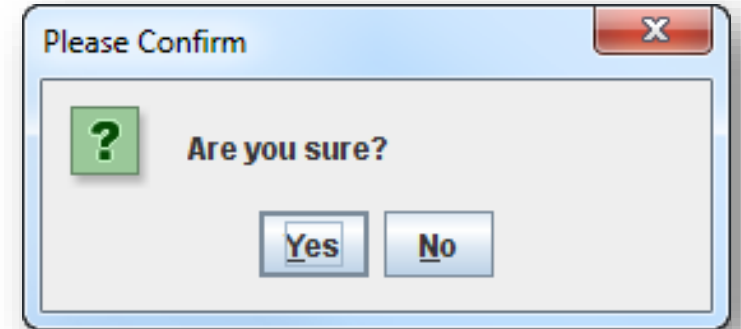
`JOptionPane.NO_OPTION`

`JOptionPane.CANCEL_OPTION`

Confirmation Dialogs

```
int answer;  
answer = JOptionPane.showConfirmDialog(null, "Are you sure?");  
if(answer == JOptionPane.YES_OPTION){  
    //If the user clicked Yes, this code is executed.  
}  
else if(answer == JOptionPane.NO_OPTION){  
    //If the user clicked no, this code is executed.  
}  
else if(answer == JOptionPane.CANCEL_OPTION){  
    //If the user clicked Cancel, this code is executed.  
}
```

Confirmation Dialogs



```
int answer;  
answer = JOptionPane.showConfirmDialog(null,  
    "Are you sure?",  
    "Please Confirm",  
    JOptionPane.YES_NO_OPTION);
```

- The third argument will be the title bar's text.
- The fourth argument is one of the following:

```
JOptionPane.YES_NO_OPTION  
JOptionPane.YES_NO_CANCEL_OPTION
```

Modal Dialog Boxes

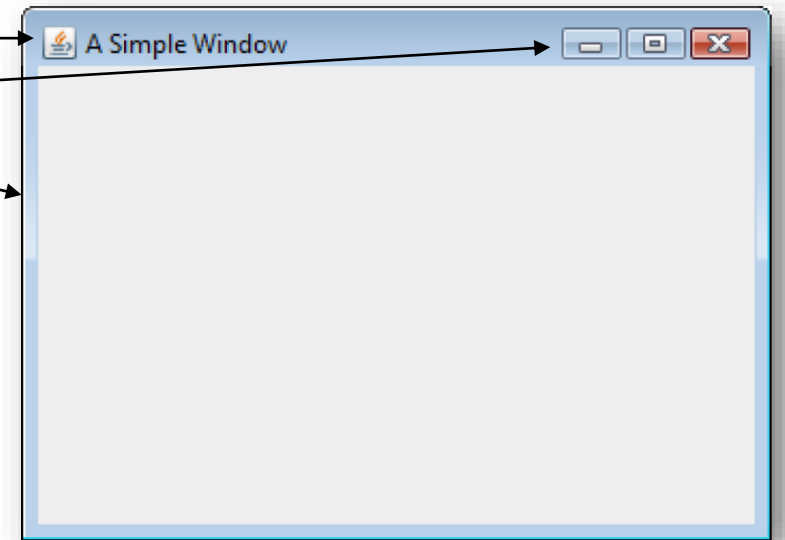
- The dialog boxes displayed by the JOptionPane class are *modal* dialog boxes.
- A ***modal dialog box*** suspends execution of any other statements until the dialog box is closed.
- When a dialog box is presented, statements in the code will not execute until the user closes the message box.
 - The application's GUI will be “frozen” until the user clears the dialog.

Creating Windows

- A ***window*** in a graphical application is a visual container that holds other components.
- In Java, the container object that can be displayed as a window is called a ***frame***.
- Using the Swing libraries, frames are created using the **JFrame** class.
- Must be imported: `import javax.swing.JFrame;`

Creating Windows

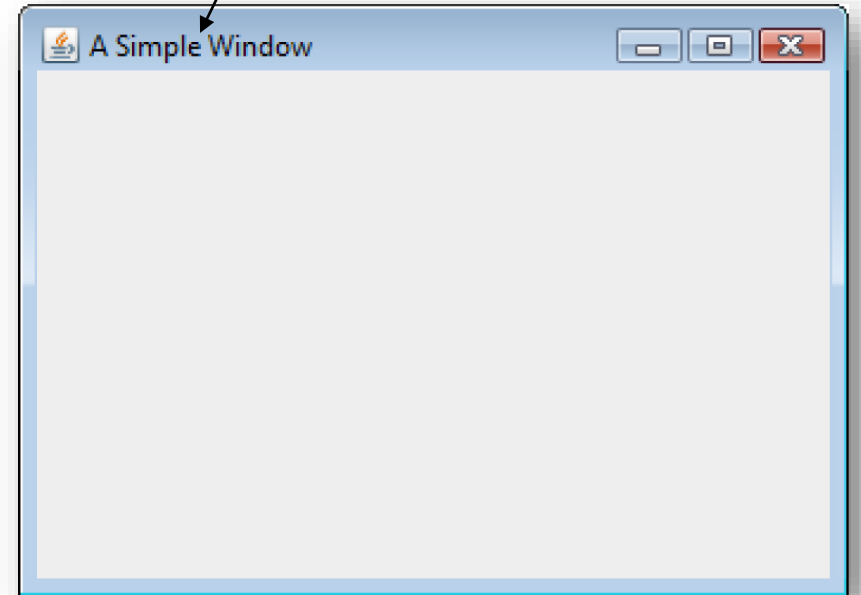
- A JFrame will create a basic window that has:
 - A border
 - A title bar
 - A set of buttons for:
 - Minimizing
 - Maximizing
 - Closing the window.
- These standard features are sometimes referred to as ***window decorations***.



Creating a JFrame

```
JFrame frame = new JFrame("A Simple Window");
```

- The String that is passed to the constructor will appear in the window's title bar when it is displayed.
- JFrames are not visible when they are first created.
 - You'll want to make any adjustments to the frame and add any components **before** making it visible.



Setting the Size of a JFrame

```
JFrame frame = new JFrame("A Simple Window");  
frame.setSize(350, 250);
```

- The JFrame's setSize method adjusts the dimensions of the window.
- The first argument is the frame's width, the second argument is the frame's height.
 - The border/decorations are included in these dimensions.
- The dimensions are measured in pixels.
 - A ***pixel*** (*picture element*) is one of the small dots that make up a screen's display.

Setting the Close Operation

```
JFrame frame = new JFrame("A Simple Window");  
frame.setSize(350, 250);  
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

- A JFrame should specify what happens the user clicks on the window's close button.
- The JFrame's setDefaultCloseOperation method specifies the action to take.
 - `JFrame.EXIT_ON_CLOSE` - causes the entire application to end.
 - `JFrame.HIDE_ON_CLOSE` - causes the window to be hidden from view, but the application does not end.
 - `JFrame.DISPOSE_ON_CLOSE` - causes the window to be exited, but does not cause the entire application to end (unless it was the only window).
- The default action is `JFrame.HIDE_ON_CLOSE`

Making the JFrame Visible

```
JFrame frame = new JFrame("A Simple Window");  
frame.setSize(350, 250);  
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
frame.setVisible(true);
```

- The JFrame's setVisible method controls if the window is visible or not.
- The setVisible method takes a boolean argument.
- Only call this method when you are finished adding components to the JFrame.

Extending JFrame

- It's impractical to put all of your GUI's code in the main method.
 - Especially if your GUI has a lot of functionality, has multiple windows/frames, etc.
- A good design choice is to create a subclass of JFrame and place all relevant code inside of it.
- The JFrame subclass can be constructed back in the main method (or wherever/whenever you want to make the window).

Extending JFrame

```
import javax.swing.JFrame;

public class MyWindow extends JFrame {

    public MyWindow(String titleIn) {
        super(titleIn);
        super.setSize(350, 250);
        super.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        super.setVisible(true);
    }

}
```

Extending JFrame

```
public class WindowExample {  
  
    public static void main(String[] args) {  
  
        MyWindow myWindow = new MyWindow("My Simple Window");  
  
    }  
  
}
```


Creating a Panel

- A JFrame isn't actually able to display buttons, text fields, and other components by itself.
 - It only handles making the window.
- The JFrame requires a container inside of it that is capable of containing specific components like buttons, text fields, etc.

Creating a Panel

- A ***panel*** is a container object used to display and organize graphical components (including other panels).
- Using the Swing libraries, panels are created using the **JPanel** class.
 - Components are typically added to a panel and then the panel is added to the JFrame's content pane.
- Must be imported: `import javax.swing.JPanel;`

Creating a Panel

```
import javax.swing.JFrame;

public class MyWindow extends JFrame {

    public MyWindow(String titleIn) {
        super(titleIn);
        super.setSize(350, 250);
        super.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel panel = new JPanel();
        //(Add stuff to the panel)
        super.add(panel);

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

Creating a Label

- A ***label*** is a graphical component (“*widget*”) that displays text.
 - Labels are not able to be edited by a user.
 - A label’s text can be changed programmatically, however.
- Using the Swing libraries, labels are created using the **JLabel** class.
- Must be imported: `import javax.swing.JLabel;`

Creating a Label

```
JLabel label = new JLabel("This is a label.");
```

- The String passed as an argument to the constructor will supply the JLabel with initial text.

```
label.setText("This is my new value.");
```

- The JLabel's setText method can be used to later change the text of the JLabel.

Creating a Label

```
JFrame frame = new JFrame("A Simple Window");  
frame.setSize(350, 250);  
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
JPanel panel = new JPanel();  
JLabel label = new JLabel("This is a label.");  
panel.add(label);
```

```
frame.add(panel);  
frame.setVisible(true);
```

Creating a Button

- A ***button*** is a rectangular, graphical component that the user can press/click to trigger an event.
- Using the Swing libraries, buttons are created using the **JButton** class.
- Must be imported: `import javax.swing.JButton;`

Creating a Button

```
 JButton button = new JButton("Click Here");
```



- The String passed as an argument to the constructor will supply the JButton with initial text.

```
 button.setText("Press Here");
```

- The JButton's setText method can be used to later change the text of the JButton.

Creating a Button

```
JFrame frame = new JFrame("A Simple Window");  
frame.setSize(350, 250);  
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
JPanel panel = new JPanel();
```

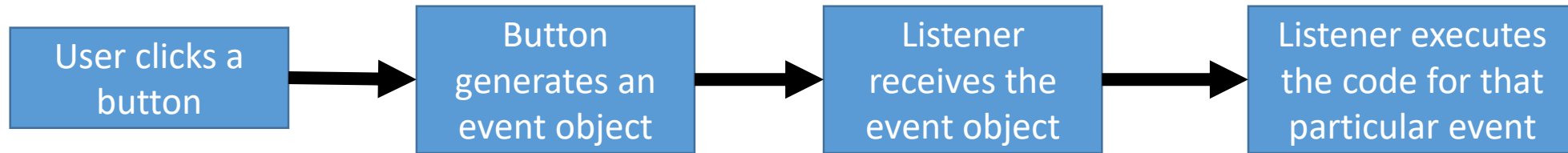
```
 JButton button = new JButton("Click Here");  
panel.add(button);
```

```
frame.add(panel);  
frame.setVisible(true);
```

Events

- An ***event*** is an object generated by a GUI component (the *event source*).
 - When a button is clicked, it will create an event object.
- An ***event listener*** is an object that responds to specific types of events when they are generated and executes code when those events occur.
 - Event listeners are registered to GUI components (like a button.)
 - If a GUI component does not have an event listener registered to it, nothing will happen when it is clicked.

Events and Event Listeners



Event Listeners

- Event listener objects are specific to each application.
 - Every program will do something different when a button is clicked.
- For this reason, event listener classes will utilize an interface.
 - There are different interfaces for different types of events.
- Like with any interface, this ensures the listener class will have the required methods.

ActionEvents

- The type of event a JButton generates when pressed is an **ActionEvent**.
- Must be imported: `import java.awt.event.ActionEvent;`

ActionListener Interface

- Components that generate `ActionEvents` require an ***ActionListener*** to execute code for the event.
- Classes that implement the `ActionListener` interface must meet the following requirements:
 - It must implement the `ActionListener` interface in the class header.
 - It must have a void method named **`actionPerformed`** that accepts one argument, an `ActionEvent` object.
- Must be imported: `import java.awt.event.ActionListener;`

ActionListener Interface

- This class, when registered to a JButton as its ActionListener, will display a dialog box when the JButton generates an(ActionEvent) object.

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JOptionPane;

public class ButtonListener implements ActionListener {

    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null, "Button Pressed");
    }

}
```

Registering an ActionListener with a JButton

- To register an ActionListener to a JButton, pass an ActionListener instance to the JButton's **addActionListener** method.

```
JButton button = new JButton("Click Here");  
button.addActionListener(new ButtonListener());
```

- ButtonListener's actionPerformed method will execute when the JButton is pressed/creates its ActionEvent.

Action Commands

- An **Action Command** is a String that is included with an ActionEvent.
- Action Commands are useful for cases where you have more than one component generating ActionEvents.
 - ActionEvents with the same command would execute the same code.
 - ActionEvents with a unique command would execute different instructions.

Setting Action Commands

- No special objects are needed.
- Call the component's `setActionCommand` method and pass a `String` as the argument.

```
JButton button = new JButton("Click Here");  
button.addActionListener(new ButtonListener());  
button.setActionCommand("buttonpressed");
```

Getting Action Commands

- In the actionPerformed method of the listener registered to the component, you can get the command from the(ActionEvent) argument.

```
public void actionPerformed(ActionEvent e) {  
    if(e.getActionCommand().equals("buttonpressed")) {  
        JOptionPane.showMessageDialog(null, "Button Pressed");  
    }  
    else if(e.getActionCommand().equals("somethingelse")) {  
        //Do something else  
    }  
}
```

- The benefit is this listener can now handle actions for events from multiple, different components/buttons.

Creating a Text Field

- A ***text field*** is a rectangular, graphical component that the user can enter text into.
- Using the Swing libraries, text fields are created using the **JTextField** class.
- Must be imported: `import javax.swing.JTextField;`

Creating a Text Field

```
JTextField field = new JTextField();
```

- A String passed as an argument to the constructor will supply the JTextField with initial text.

```
JTextField field = new JTextField("Starting text");
```

- An int passed as an argument to the constructor will supply the JTextField with initial width.

```
JTextField field = new JTextField(10);
```

Creating a Text Field / Setting its Text

- A String and int passed as arguments to the constructor will supply the JTextField with initial text and an initial width.

```
JTextField field = new JTextField("Starting text", 10);
```

- The JTextField's setText method can be used to later change the text of the JTextField.
 - Any existing text in the field will be deleted and replaced.

```
field.setText("Something different");  
field.setText(""); //Makes the text field empty
```

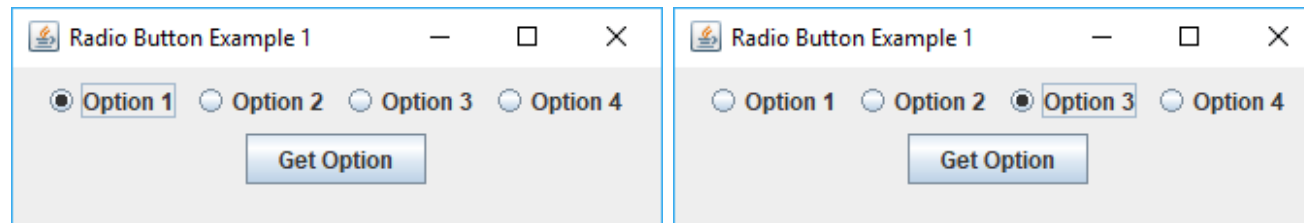
Retrieving Text from a Text Field

- The JTextField's getText method can be used to retrieve the text currently in the JTextField.
 - The existing text in the field will not be deleted/replaced when this method is called.

```
String text = field.getText();
```

Creating a Radio Button

- A **radio button** is a circular, graphical component that the user can select.
 - Radio buttons often appear in groups, where only one can be selected at a time.



- Using the Swing libraries, radio buttons are created using the **JRadioButton** class.
- Must be imported: `import javax.swing.JRadioButton;`

Creating a Radio Button

- A String passed as the argument to the constructor will supply the JRadioButton with its title.

```
JRadioButton radio2 = new JRadioButton("Option 2");
```



Button Groups

- As previously stated, radio buttons are normally grouped together.
- In a radio button group, only one of the radio buttons in the group may be selected at a time.
- Clicking on a radio button selects it and automatically deselects any other radio button in the same group.

Creating a Button Group

- A ***button group*** is a collection of *mutually exclusive* radio buttons.
 - The button group does not physically group the radio buttons together in the GUI.
 - It groups them so that only one radio button in the group may be selected at a time.
- Using the Swing libraries, radio buttons are created using the **ButtonGroup** class.

```
ButtonGroup group = new ButtonGroup();
```

- Must be imported: **import javax.swing.ButtonGroup;**

Creating a Button Group

- The ButtonGroup object handles creating the relationship between the radio buttons that it contains.
- Radio buttons are added to a ButtonGroup using the ButtonGroup's add method.

```
JRadioButton radio1 = new JRadioButton("Option 1");  
JRadioButton radio2 = new JRadioButton("Option 2");  
JRadioButton radio3 = new JRadioButton("Option 3");  
JRadioButton radio4 = new JRadioButton("Option 4");  
  
ButtonGroup group = new ButtonGroup();  
  
group.add(radio1);  
group.add(radio2);  
group.add(radio3);  
group.add(radio4);
```

Button Groups

- ButtonGroup objects are not containers like JPanels.
- If you wish to add the radio buttons to a panel, you must add them individually.

```
panel.add(radio1);  
panel.add(radio2);  
panel.add(radio3);  
panel.add(radio4);
```

Determining a Radio Button's State

- Depending on what your program is doing, you may not want an event to occur when a radio button is selected.
 - But, at some point, your program may need to find out which radio button is currently selected.
- The JRadioButton's isSelected method returns a boolean value indicating if the radio button is selected.

```
if(radio1.isSelected()) {  
    // Code here executes if the radio1  
    // button is selected.  
}
```

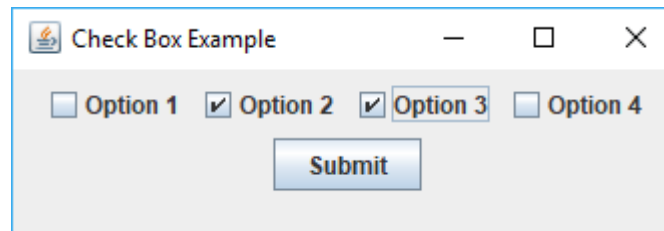
Selecting a Radio Button Programmatically

- To select (or deselect) a radio button in the source code, use the `setSelected` method.

```
radio1.setSelected(true);
```

Creating a Check Box

- A **check box** is a square, graphical component that the user can select or deselect.
 - Unlike radio buttons, more than one check box can be selected at a time.



- Using the Swing libraries, check boxes are created using the **JCheckBox** class.
- Must be imported: `import javax.swing.JCheckBox;`

Creating a Check Box

- A String passed as the argument to the constructor will supply the JCheckBox with its title.

```
JCheckBox check2 = new JCheckBox("Option 2");
```



Determining a Check Box's State

- A JCheckBox's **isSelected** method will determine whether a JCheckBox component is selected.
 - Returns a boolean.

```
if(check2.isSelected()) {  
    // Code here executes if the check2  
    // check box is selected.  
}
```

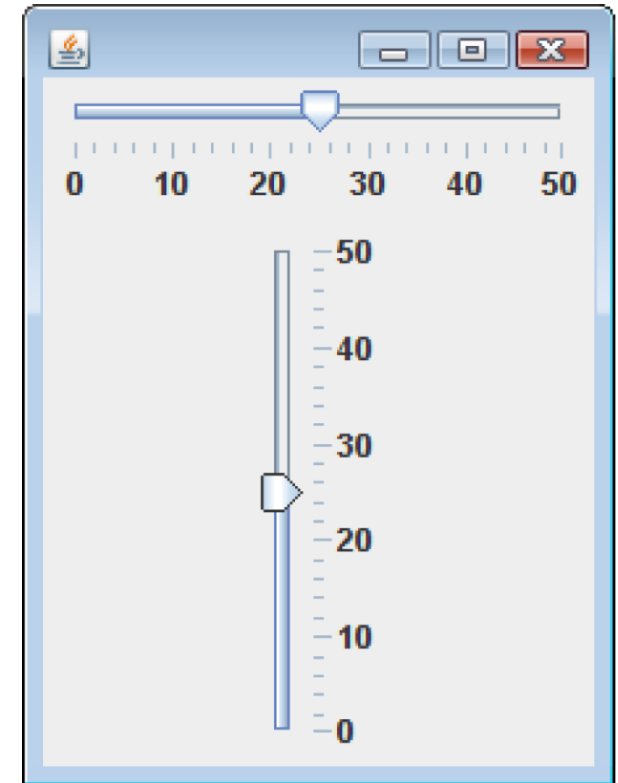
Selecting a Check Box Programmatically

- To select (or deselect) a check box in the source code, use the check box's **setSelected** method.

```
check2.setSelected(true);
```

Creating a Slider

- A ***slider*** is a component that allows the user to graphically adjust a number within a range.
 - Sliders display an image of an arrow that can be dragged along a track.
- Using the Swing libraries, sliders are created using the **JSlider** class.
- Must be imported: `import javax.swing.JSlider;`



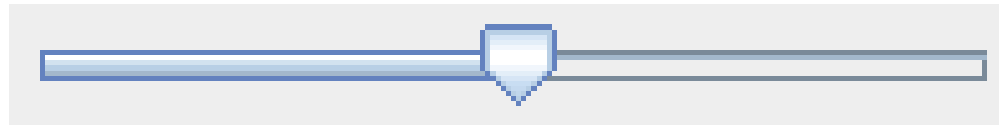
Creating a Slider

- A slider's constructor requires four arguments:
 - Its orientation
 - Its minimum value (int)
 - Its maximum value (int)
 - Its default selected value (int)
- The two possible orientations are:

`JSlider.HORIZONTAL`
`JSlider.VERTICAL`

Creating a Slider

```
JSlider slider = new JSlider(JSlider.HORIZONTAL, 0, 10, 5);
```



- By default, sliders do not display labels or tick marks.

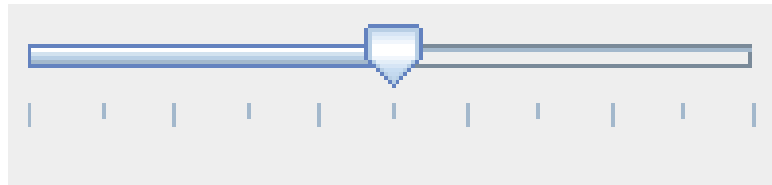
Setting Slider Tick Marks

- The major and minor tick mark spacing can be set with the JSlider's **setMajorTickSpacing** and **setMinorTickSpacing** methods.

```
slider.setMajorTickSpacing(2);  
slider.setMinorTickSpacing(1);
```

- The tick marks can be made visible by calling the JSlider's **setPaintTicks** method.

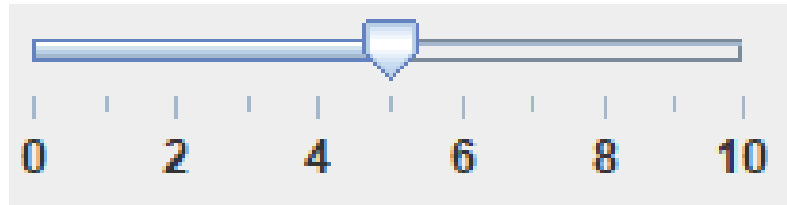
```
slider.setPaintTicks(true);
```



Setting Slider Tick Mark Labels

- The major tick mark labels can be made visible by calling the JSlider's **setPaintLabels** method.

```
slider.setPaintLabels(true);
```



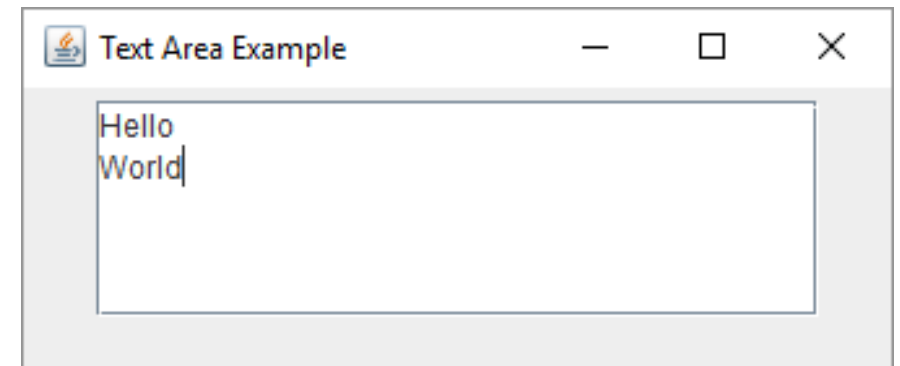
Getting the Current Value of a Slider

- To retrieve the current value selected in a JSlider, use the JSlider's `getValue` method.
 - Returns an `int`.

```
currentValue = slider.getValue();
```

Creating a Text Area

- A ***text area*** is a component that allows the user to enter text.
 - Unlike a text field that is one line, text areas can display multiple lines.
- Using the Swing libraries, text areas are created using the **JTextArea** class.
- Must be imported: **import javax.swing.JTextArea;**



Creating a Text Area

- Two ways to construct a text area.
 - Only specifying the height and width (both ints)

```
JTextArea textArea = new JTextArea(5, 25);
```

- Specifying an initial text value (String) and the height and width.

```
JTextArea textArea = new JTextArea("Starting Text", 5, 25);
```

- The height and width only specify the visual size of the text area, **not** how much text can be entered.

Getting/Setting the Text of a Text Area

- The JTextArea's getText method returns the text currently entered in the text area.
 - Returned as one String.

```
String textValue = textArea.getText();
```

- The JTextArea's setText method updates the text currently displayed in the text area with the argument it is provided.

```
textArea.setText("New Text");
```

Line Wrapping

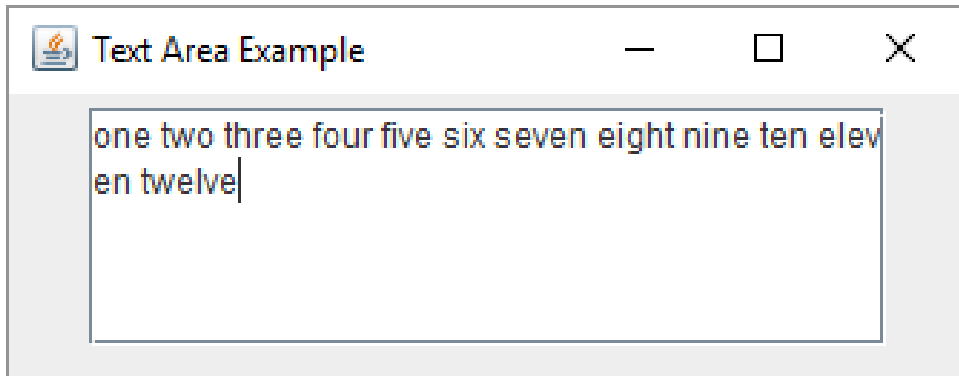
- By default, JTextArea components do not perform line wrapping.
 - ***Line wrapping*** is the continuation of one line in a text area to the next line.
- JTextArea's line wrapping is disabled by default.
 - Use the JTextArea's setLineWrap method to enable/disable ***character wrapping***- line breaks occur between characters of the same word.

```
textArea.setLineWrap(true);
```

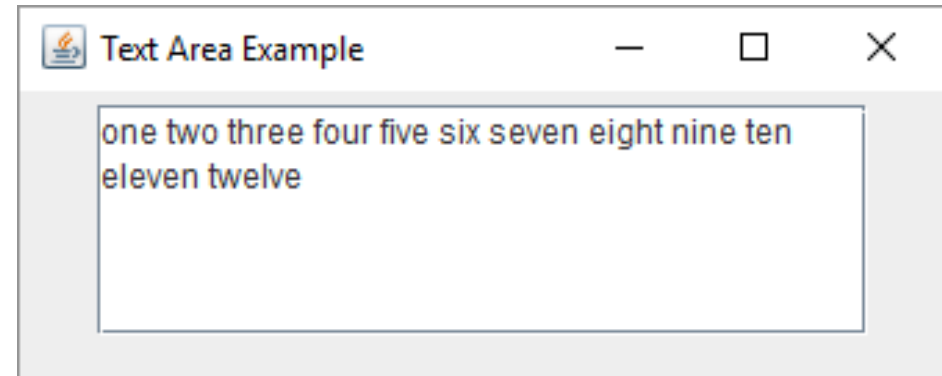
Line Wrapping

- The JTextArea's setWrapStyleWord method enables/disables ***word wrapping***- line breaks occur between words.

```
textArea.setWrapStyleWord(true);
```



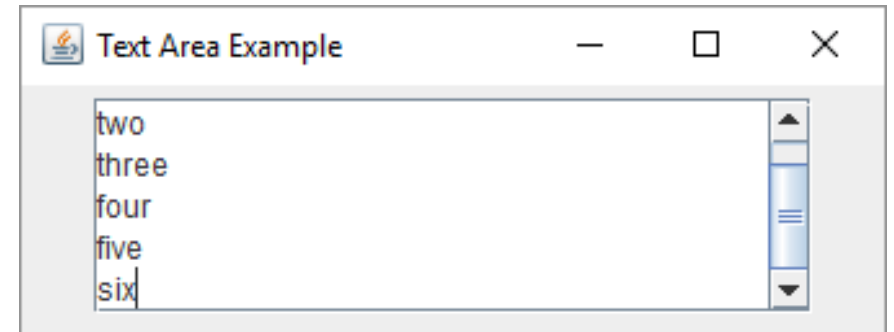
Character Wrapping



Word Wrapping

Scroll Bars

- JTextArea components do not automatically display scroll bars.
 - A **scroll bar** is a graphical component that allows viewing the contents of a component that exceed the panel/components size.
- A text area must be given to a scroll pane.
 - Using the Swing libraries, scroll panes are created using the **JScrollPane** class.
- Must be imported: **import javax.swing.JScrollPane;**



Creating a Scroll Pane

- The JScrollPane object displays both vertical and horizontal scroll bars on a text area.
 - By default, the scroll bars are not displayed until they are needed.
- A JScrollPane is constructed with the child component given as an argument.
 - This also works for JPanels.

```
JTextArea textArea = new JTextArea(5, 25);  
JScrollPane scrollPane = new JScrollPane(textArea);
```

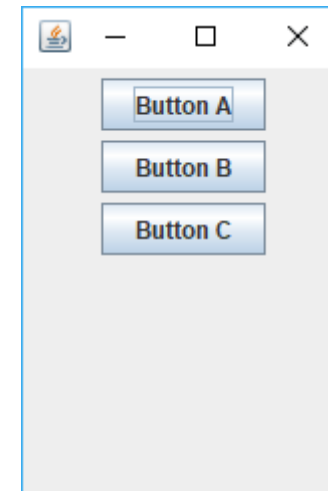
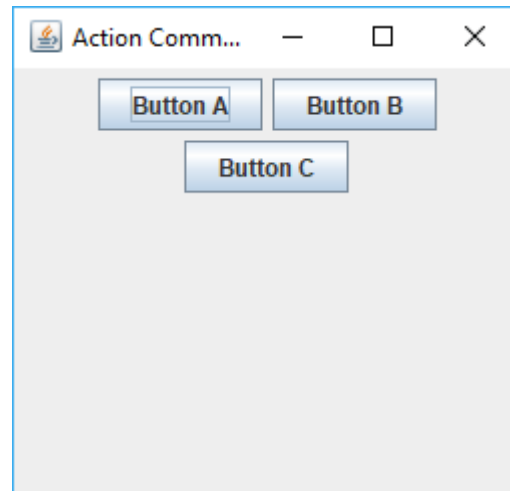
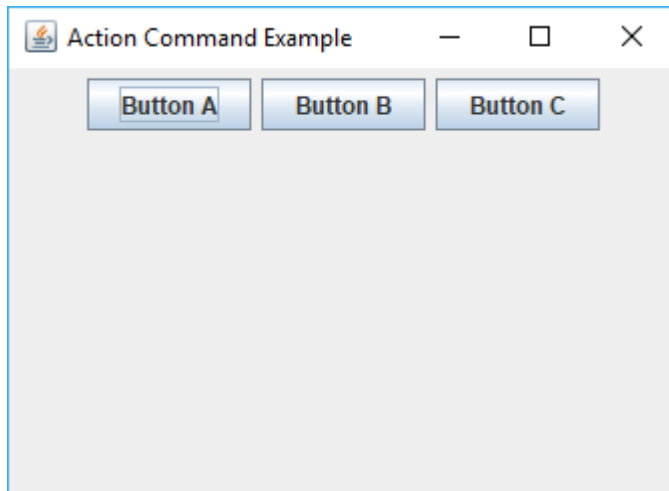
- Only the JScrollPane needs to be added to a parent panel.

Layout Managers

- A ***Layout Manager*** is a class that is used by Container classes (like a JPanel) that will control how components appear in the Container.
- The AWT libraries come with a number of Layout Managers.
 - The ones demonstrated here are Flow Layout, Border Layout, and Grid Layout Managers.
 - These types of layout managers (and how they behave) are common across multiple programming languages.
- Each will need to be imported:
`import java.awt.FlowLayout;`
`import java.awt.BorderLayout;`
`import java.awt.GridLayout;`

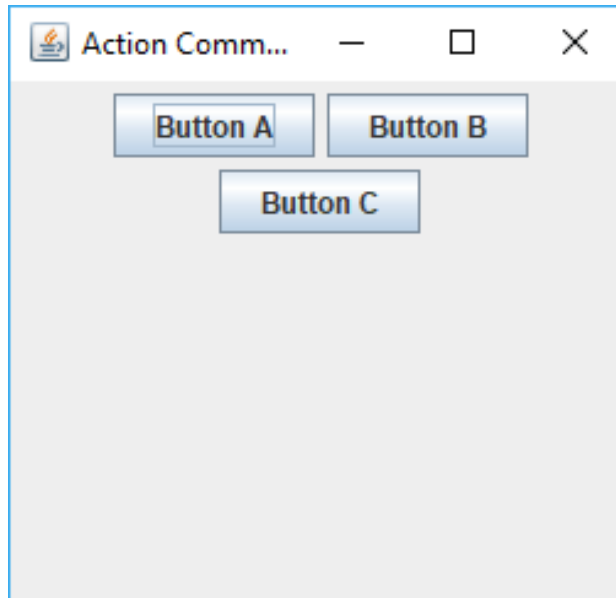
FlowLayout

- The FlowLayout Manager organizes components horizontally, from left to right, in the order they were added to the container.
 - When there is not enough room to display the components, the components “flow” to the next row.



FlowLayout

- By default, a JPanel uses a FlowLayout with center alignment and 5-pixel padding.
 - This can be changed by setting the JPanel's layout manager to use a FlowLayout with your specifications (alignment and gap distance).



Setting a JPanel's Layout Manager

- The JPanel's `setLayout` method allows us to pass an instance of a layout manager object to the JPanel.

```
JPanel panel = new JPanel();  
panel.setLayout(new FlowLayout());
```

- The above wouldn't change the normal functionality of a JPanel, as this `FlowLayout` object defaults to center alignment and 5-pixel padding.

Setting a JPanel's Layout Manager

- Specifying one of the following as the first argument to the constructor will set the FlowLayout's alignment:

`FlowLayout.RIGHT`
`FlowLayout.LEFT`
`FlowLayout.CENTER`

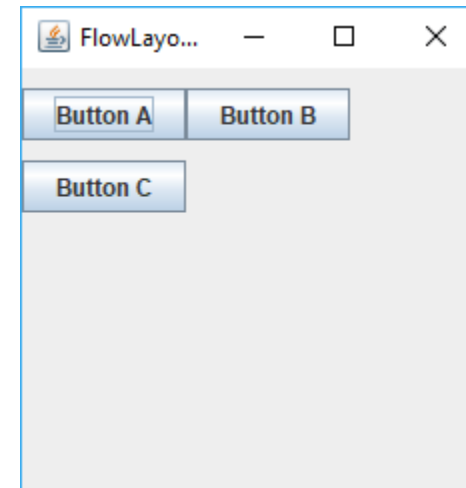
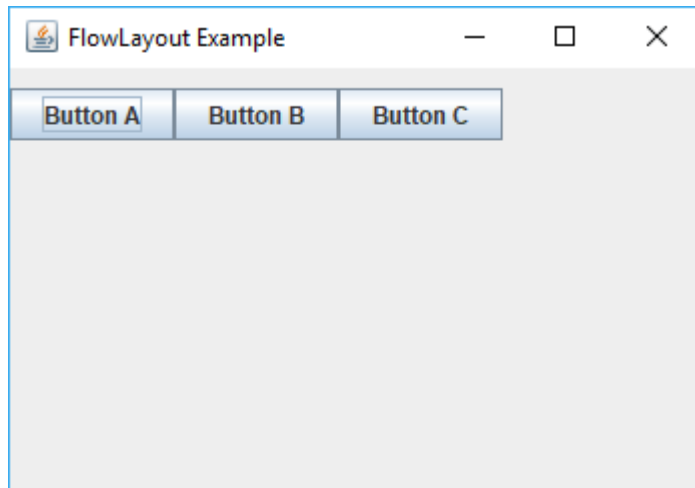
- The FlowLayout object aligns components to the left (and will still default to 5-pixel padding.)

```
JPanel panel = new JPanel();  
panel.setLayout(new FlowLayout(FlowLayout.RIGHT));
```

Setting a JPanel's Layout Manager

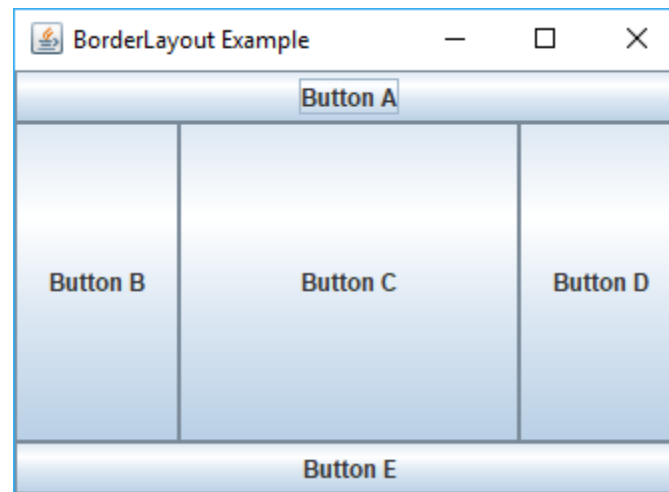
- Two additional arguments to the constructor will set the FlowLayout's horizontal and vertical padding:

```
JPanel panel = new JPanel();  
panel.setLayout(new FlowLayout(FlowLayout.RIGHT, 0, 10));
```



BorderLayout

- The BorderLayout Manager organizes components into five regions: north, south, east, west, and center.
 - Each region is limited to **one** component.
 - By default, has no padding between regions.



Setting a JPanel's Layout Manager

- The JPanel's setLayout method allows us to pass an instance of a layout manager object to the JPanel.

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

- Specifying two int arguments to the BorderLayout constructor will set horizontal and vertical padding.

```
JPanel panel = new JPanel();  
panel.setLayout(new BorderLayout(5, 7));
```


Setting a JPanel's Layout Manager

- Components added to a BorderLayout must specify one of the following regions:

`BorderLayout.NORTH`

`BorderLayout.SOUTH`

`BorderLayout.EAST`

`BorderLayout.WEST`

`BorderLayout.CENTER`

- If a region is not specified, the component is placed in the center region.

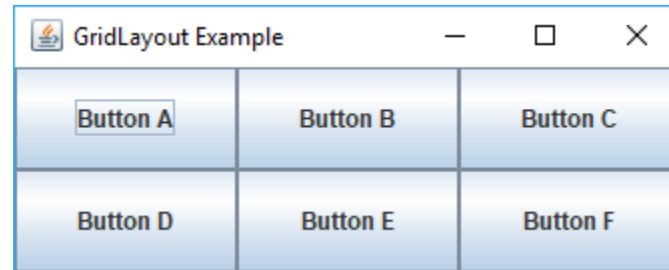
Adding a Component to a BorderLayout

- The JPanel's add method will accept two arguments, the component to add and the region to place the component:

```
panel.add(buttonA, BorderLayout.NORTH);  
panel.add(buttonB, BorderLayout.WEST);
```

GridLayout

- The GridLayout Manager organizes components into a series of rows and columns.
 - All cell widths will be equal in size; All cell heights will be equal in size.
 - Each cell is limited to **one** component.



GridLayout

- Components would be added to a GridLayout in the following order (for a 5×5 grid).

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Setting a JPanel's Layout Manager

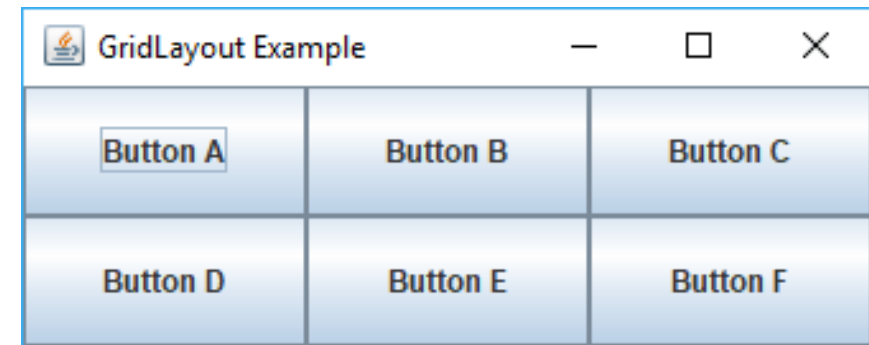
- The JPanel's `setLayout` method allows us to pass an instance of a layout manager object to the JPanel.
 - The two `int` arguments in the `GridLayout` constructor will set the number of rows and columns.

```
JPanel panel = new JPanel();  
panel.setLayout(new GridLayout(2, 3));
```

Adding a Component to a GridLayout

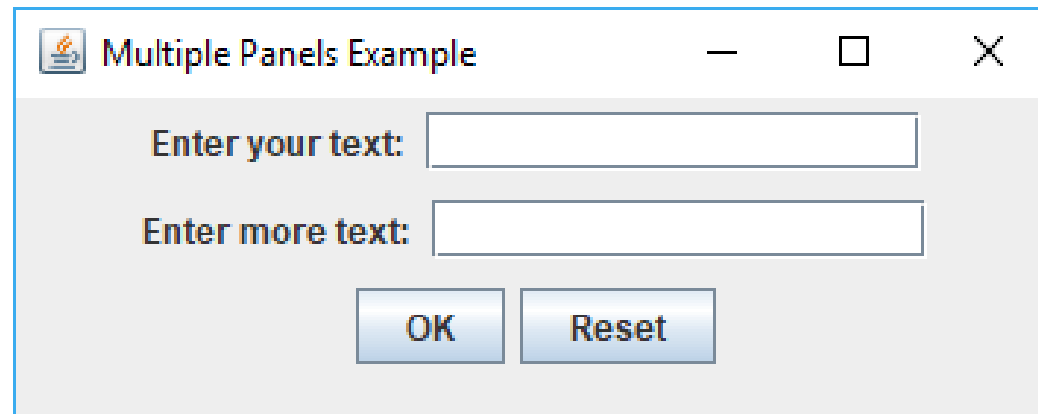
- There is no way to specify which cell to add the component.
 - They are placed in the order they were added to the panel.

```
panel.add(buttonA);  
panel.add(buttonB);  
panel.add(buttonC);  
panel.add(buttonD);  
panel.add(buttonE);  
panel.add(buttonF);
```



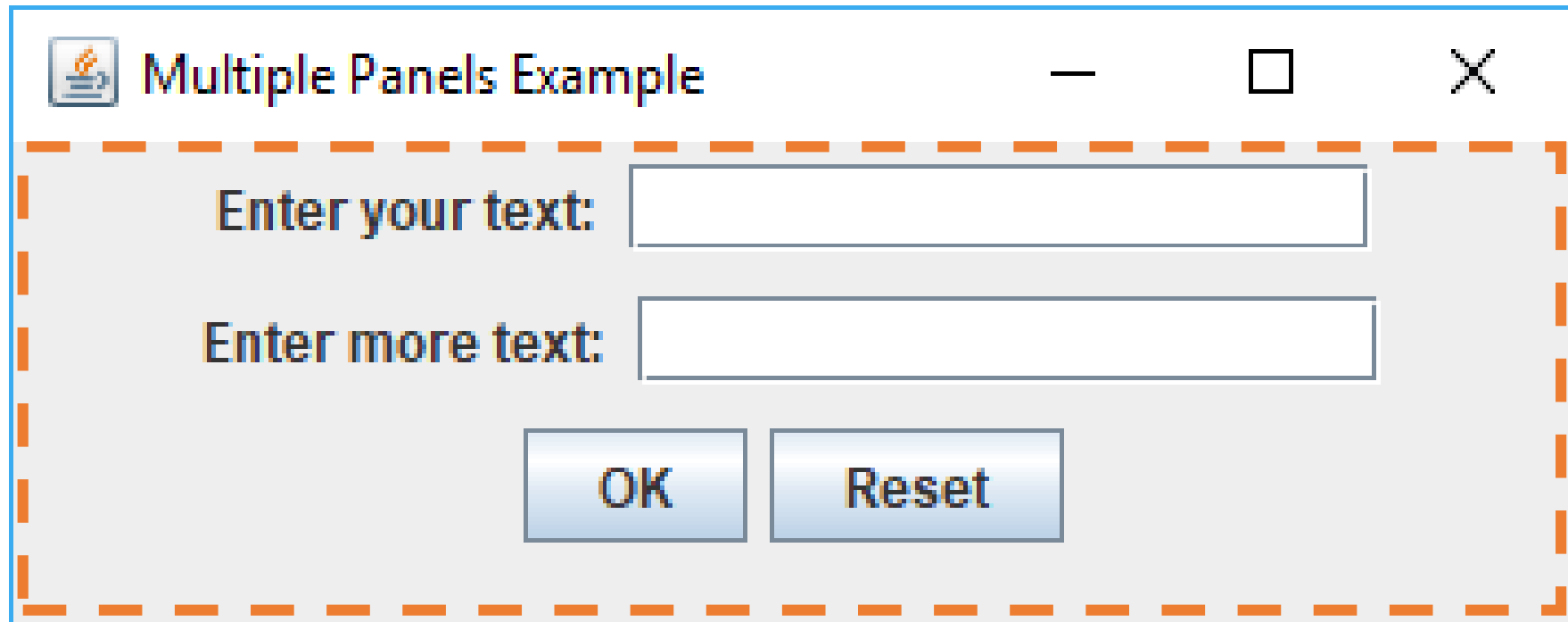
Panels and Layouts

- Adding components to panels and then nesting the panels inside the regions of other panels can overcome the single component limitation of layout regions.



Panels and Layouts

PANEL



The image shows a Java Swing window titled "Multiple Panels Example". The window has a standard title bar with a minimize button, a maximize button, and a close button. Inside the window, there is a light gray rectangular area enclosed by a dashed orange border, which represents a panel. Within this panel, there are two text input fields. The first field is preceded by the label "Enter your text:" and the second by "Enter more text:". Below these fields are two buttons labeled "OK" and "Reset".

Panels and Layouts

BORDER

NORTH

CENTER

Multiple Panels Example

Enter your text:

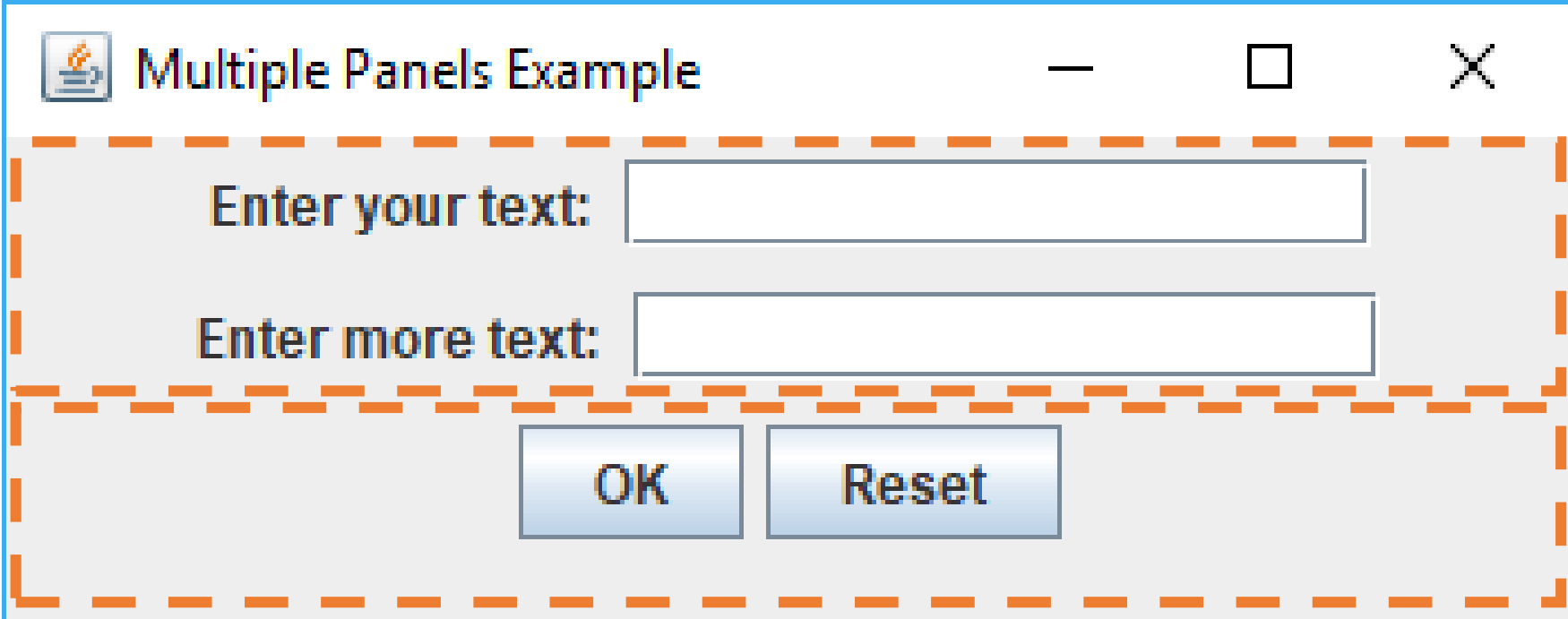
Enter more text:

OK Reset

Panels and Layouts

PANEL

PANEL



The image shows a Java Swing window titled "Multiple Panels Example". The window is divided into two panels by a dashed orange line. The top panel contains two text input fields with labels "Enter your text:" and "Enter more text:". The bottom panel contains two buttons labeled "OK" and "Reset". The word "PANEL" is written in orange text to the left of each panel. The window has a standard title bar with a minimize button, a maximize button, and a close button.

Panels and Layouts

GRID

FLOW

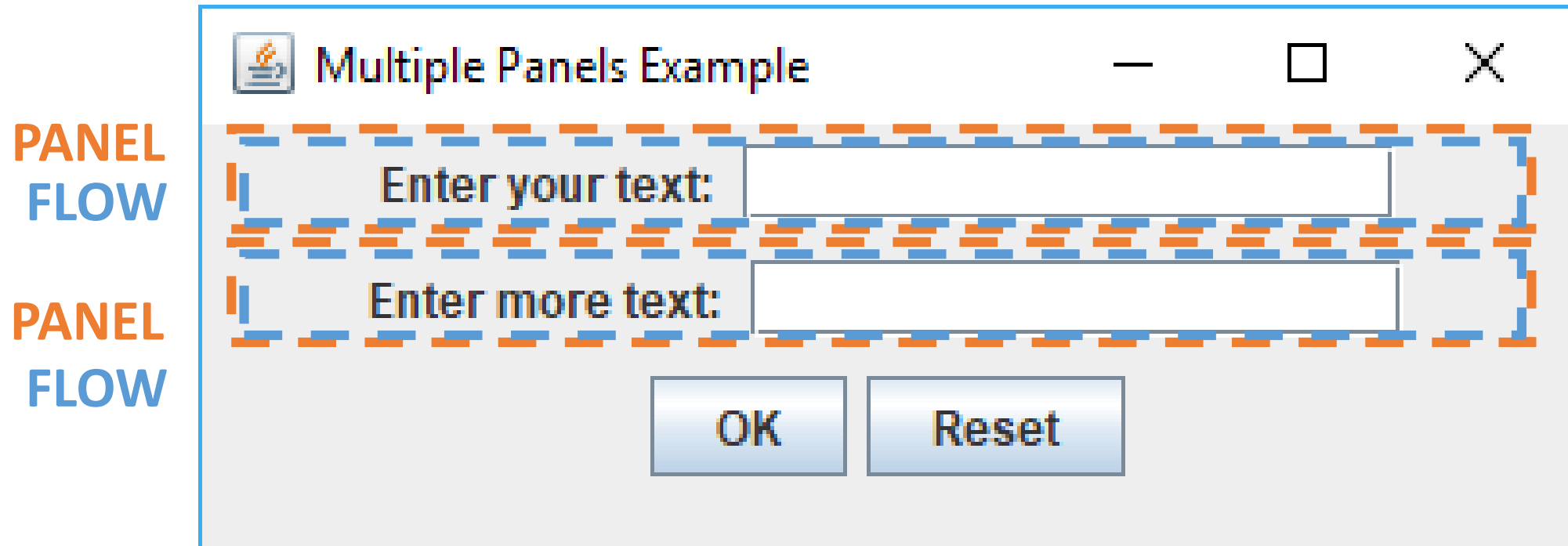
Multiple Panels Example

Enter your text:

Enter more text:

OK Reset

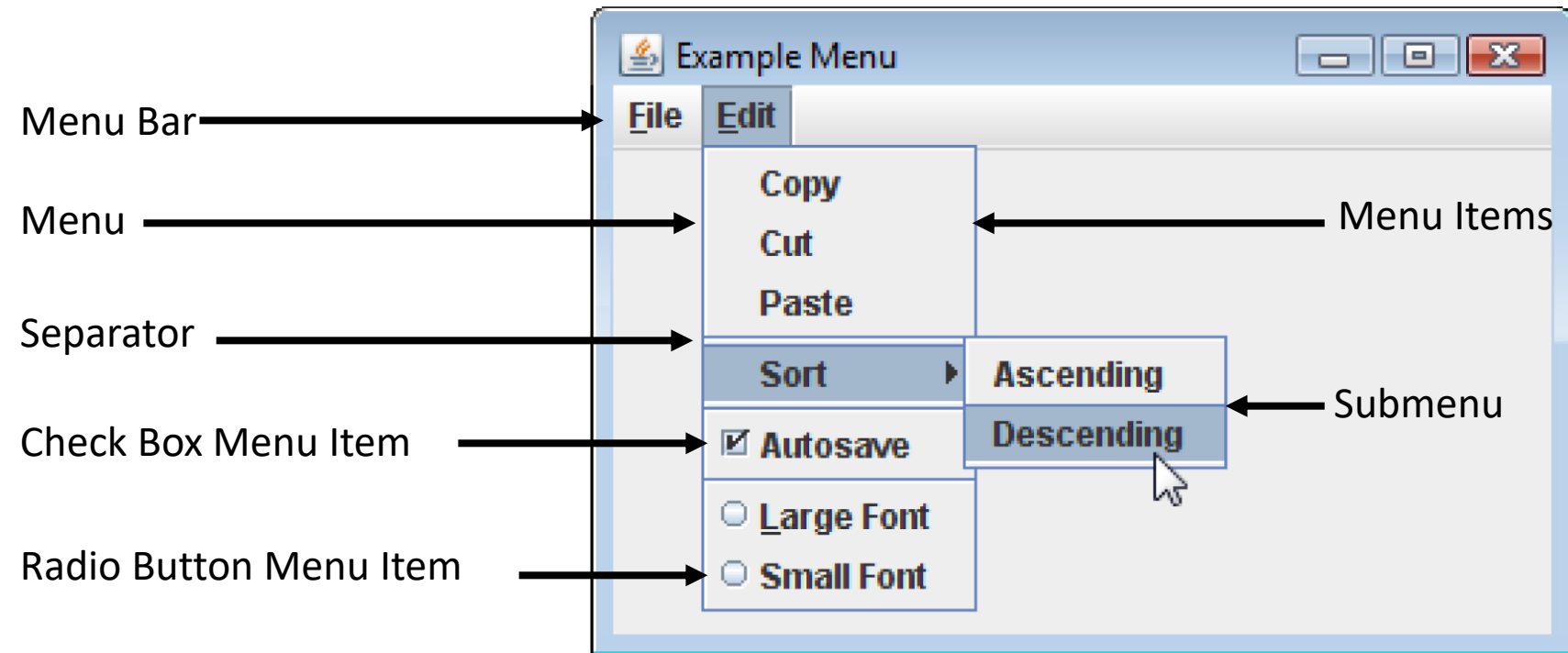
Panels and Layouts



Menu Systems

- A **menu system** is a collection of commands organized in one or more “drop-down” menus.
- A menu system commonly consists of:
 - **Menu Bar** – Lists the titles of one or more menus.
 - **Menu** – A single drop-down list of menu items.
 - **Menu Item** – A selectable item in a menu.
 - **Check Box Menu Item** – Menu item that appears with a small check box beside it.
 - **Radio Button Menu Item** – Menu item that appears with a small radio button beside it.
 - Can be grouped using a ButtonGroup.
 - **Separator bar** – Horizontal line that is used to separate groups of menu items.
 - **Submenu** – A menu within a menu item.

Menu Systems



Creating a Menu Bar

- Using the Swing libraries, menu bars are created using the **JMenuBar** class.
- Must be imported: `import javax.swing.JMenuBar;`

Creating a Menu Bar

```
JMenuBar menuBar = new JMenuBar();
```

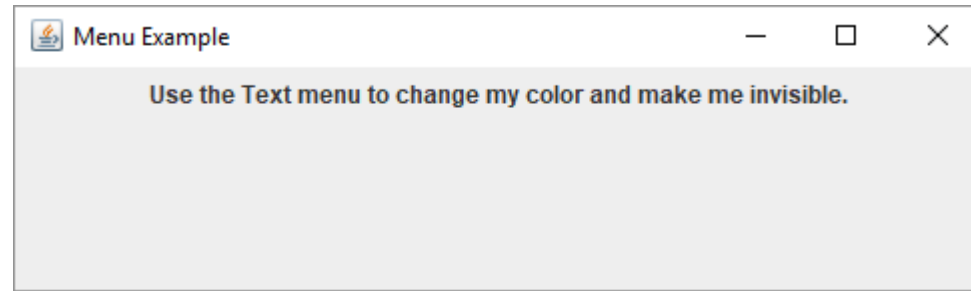
- The JMenuBar can be set to its parent frame by using the frame's setJMenuBar method.

```
frame.setJMenuBar(menuBar);
```

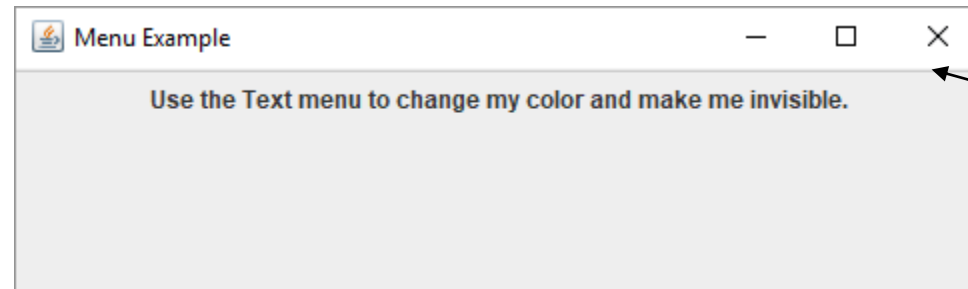
- If you subclassed JFrame, then you would set it using the superclass.

```
super.setJMenuBar(menuBar);
```


Creating a Menu Bar



JFrame with no Menu Bar



JFrame with Menu Bar

If no menus have been added, the JMenuBar will appear as a small sliver.

Creating a Menu

- Using the Swing libraries, menus are created using the **JMenu** class.
- Must be imported: `import javax.swing.JMenu;`

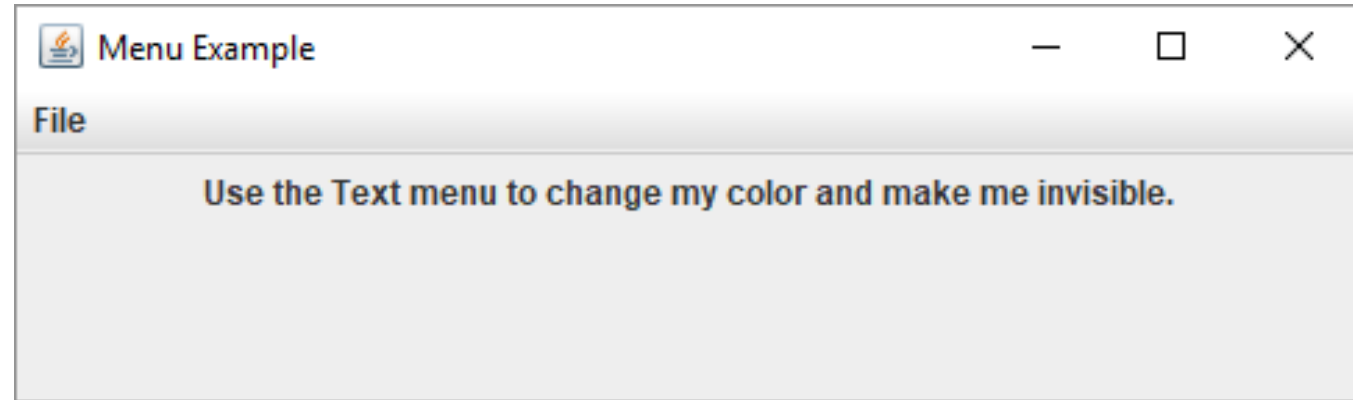
Creating a Menu

```
JMenu fileMenu = new JMenu("File");
```

- The String passed as an argument to the constructor will supply the JMenu with initial text.
- Add the JMenu to a JMenuBar using the JMenuBar's add method.

```
menuBar.add(fileMenu);
```

Creating a Menu



JFrame with one Menu in the Menu Bar

Creating a Menu Item

- Using the Swing libraries, menu items are created using the **JMenuItem** class.
- Must be imported: `import javax.swing.JMenuItem;`

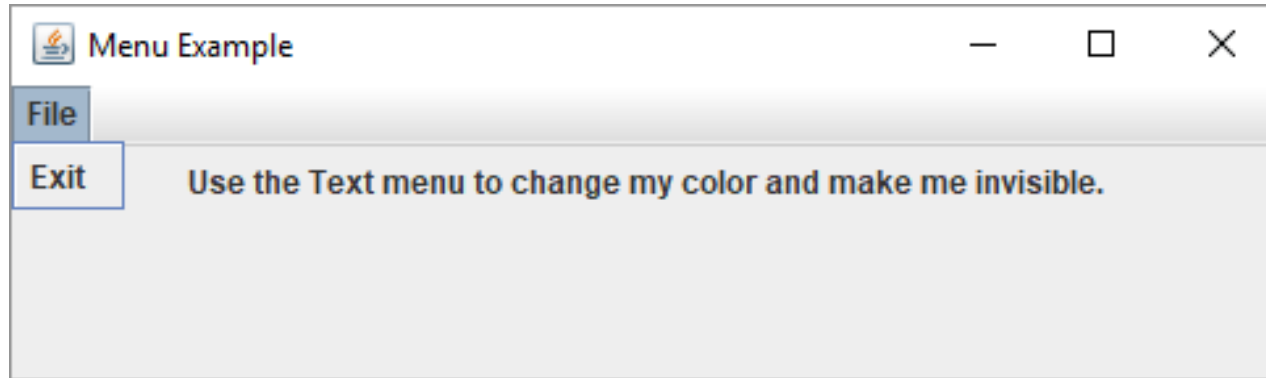
Creating a Menu Item

```
JMenuItem exitItem = new JMenuItem("Exit");
```

- The String passed as an argument to the constructor will supply the JMenuItem with initial text.
- Add the JMenuItem to a JMenu using the JMenu's add method.

```
fileMenu.add(exitItem);
```

Creating a Menu Item



Menu with one Menu Item

Menu Item Action Events

- JMenuItem components generate an Action Event when clicked.
- A class that implements the ActionListener interface must be registered to the component to handle the event.
 - Exactly how you would handle an Action Event from a component like a JButton.
- You will want to set an Action Command to each menu item.
 - So you know which item was selected.

Menu Item Action Events

- The following registers an Action Listener to the menu item and gives it an Action Command.
 - Assuming this class is implementing the ActionListener interface.

```
exitItem.addActionListener(this);  
exitItem.setActionCommand("exit");
```

Creating a Radio Button Menu Item

- Using the Swing libraries, radio button menu items are created using the **JRadioButtonMenuItem** class.
- Must be imported: `import javax.swing.JRadioButtonMenuItem;`



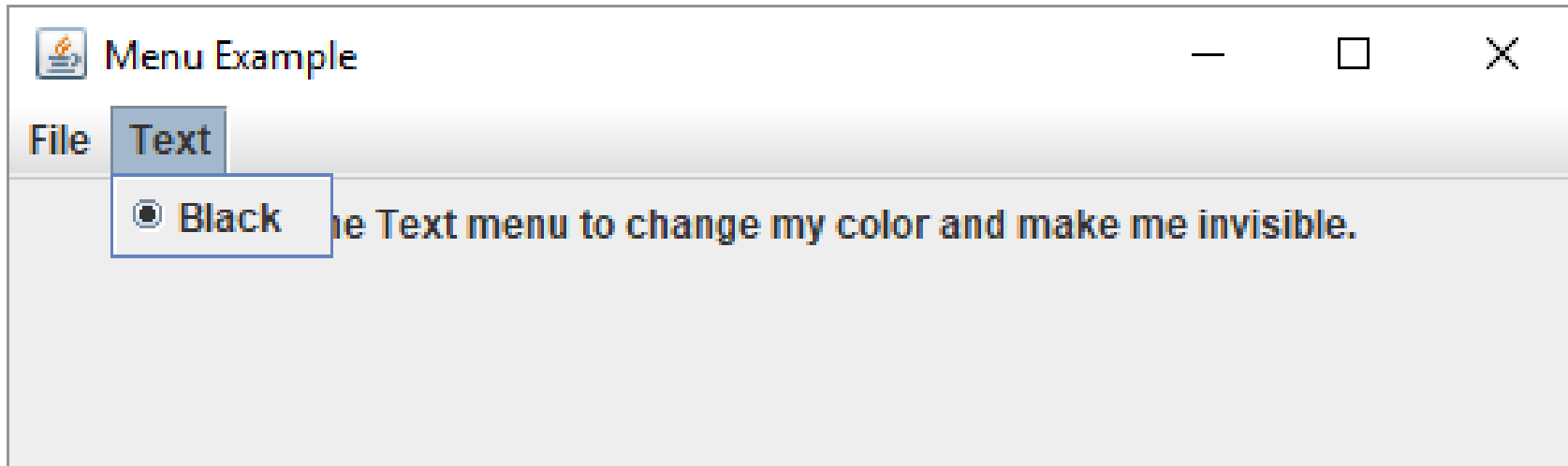
Creating a Radio Button Menu Item

```
JRadioButtonMenuItem blackItem = new JRadioButtonMenuItem("Black", true);
```

- The String passed as an argument to the constructor will supply the JRadioButtonMenuItem with initial text.
- The second, optional, boolean argument determines if the button is pre-selected.
 - If the argument is false or not present, the button will not be selected at creation.
- Add the JRadioButtonMenuItem to a JMenu using the JMenu's add method.

```
textMenu.add(blackItem);
```

Creating a Radio Button Menu Item



Creating a Group of Radio Button Menu Items

- JRadioButtonMenuItem components can be grouped together in a ButtonGroup object so that only one of them can be selected at a time.
 - Just as you would group JRadioButtons in a ButtonGroup.
 - The JRadioButtonMenuItem components still need to be individually added to the menu.

```
ButtonGroup colorGroup = new ButtonGroup();  
JRadioButtonMenuItem blackItem = new JRadioButtonMenuItem("Black", true);  
colorGroup.add(blackItem);  
textMenu.add(blackItem);
```

Creating a Group of Radio Button Menu Items

```
ButtonGroup colorGroup = new ButtonGroup();

JRadioButtonMenuItem blackItem = new JRadioButtonMenuItem("Black", true);
JRadioButtonMenuItem redItem = new JRadioButtonMenuItem("Red", true);
JRadioButtonMenuItem blueItem = new JRadioButtonMenuItem("Blue", true);

colorGroup.add(blackItem);
colorGroup.add(redItem);
colorGroup.add(blueItem);

textMenu.add(blackItem);
textMenu.add(redItem);
textMenu.add(blueItem);
```

Radio Button Menu Item States

- The JRadioButtonMenuItem's isSelected method returns true if the item is selected or false if the item is not selected.

`blackItem.isSelected();`

- This allows the program to determine the state of the Radio Button Menu Item.

Creating a Check Box Menu Item

- Using the Swing libraries, radio button menu items are created using the **JCheckBoxMenuItem** class.
- Must be imported: `import javax.swing.JCheckBoxMenuItem;`



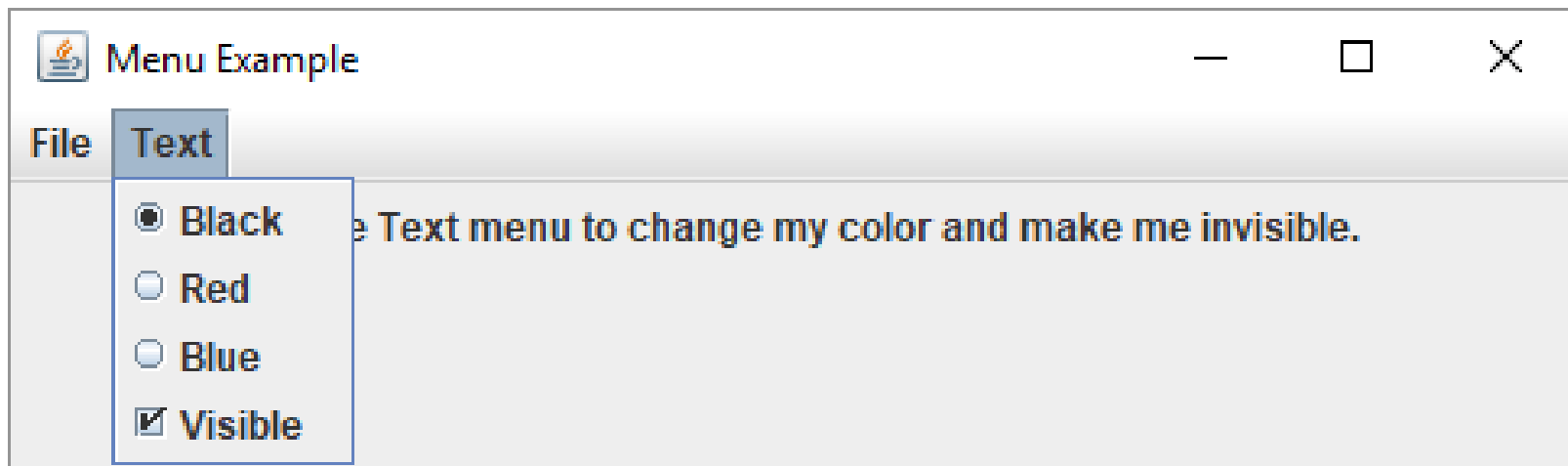
Creating a Check Box Menu Item

```
JCheckBoxMenuItem visibleItem = new JCheckBoxMenuItem("Visible", true);
```

- The String passed as an argument to the constructor will supply the JCheckBoxMenuItem with initial text.
- The second, optional, boolean argument determines if the button is pre-selected.
 - If the argument is false or not present, the button will not be selected at creation.
- Add the JCheckBoxMenuItem to a JMenu using the JMenu's add method.

```
textMenu.add(visibleItem);
```

Creating a Check Box Menu Item



Check Box Menu Item States

- The JcheckBoxMenuItem's isSelected method returns true if the item is selected or false if the item is not selected.

`visibleItem.isSelected();`

- This allows the program to determine the state of the Check Box Menu Item.

Menu Item Action Events

- Both `JRadioButtonMenuItem` and `JCheckBoxMenuItem` components generate Action Events when clicked.
- A class that implements the `ActionListener` interface must be registered to the component to handle the event.
- You will want to set an Action Command to each menu item.

Submenus

- To create a submenu, create and add a new JMenu to an existing JMenu.

```
JMenu colorMenu = new JMenu("Colors");
```

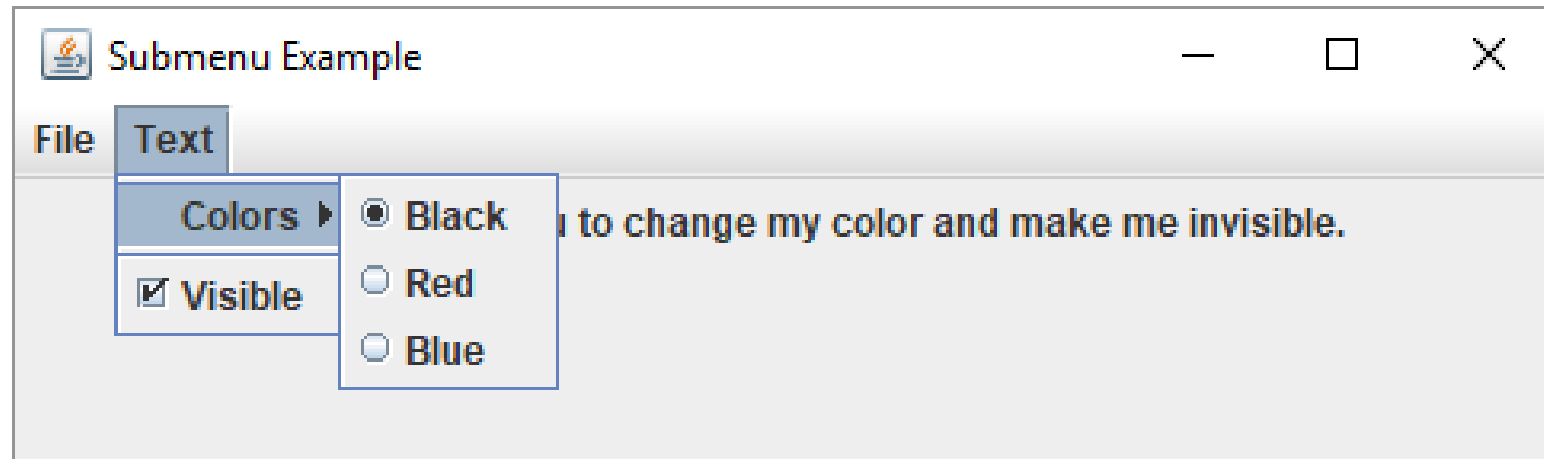
```
colorMenu.add(blackItem);
```

```
colorMenu.add(redItem);
```

```
colorMenu.add(blueItem);
```

```
textMenu.add(colorMenu);
```

Submenus



Mnemonics

- A ***mnemonic*** is a key that you press in combination with the Alt key to quickly access a component.
 - These are sometimes referred to as ***hot keys***.
 - Mnemonics allow the GUI program to still be used, even in the absence of a mouse.
- A hot key is assigned to a component through the component's setMnemonic method.
- The argument passed to the method is an integer code that represents the key you wish to assign.

Mnemonics

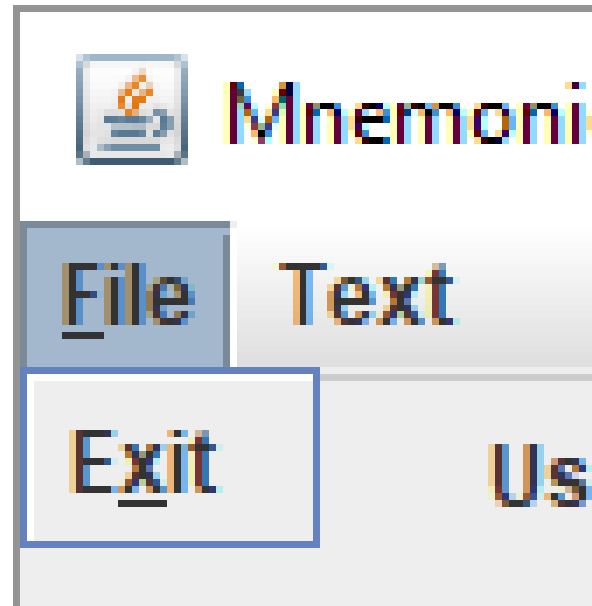
- The key codes are predefined constants in the KeyEvent class.
- Must be imported: `import java.awt.event.KeyEvent;`
- The KeyEvent constants take the form of:
 - KeyEvent.VK_X, where X is a key on the keyboard.
 - The letters VK in the constants stand for “virtual key”.
 - To assign the A key as a mnemonic, you would use:
 - KeyEvent.VK_A

Mnemonics

```
fileMenu.setMnemonic(KeyEvent.VK_F);  
exitItem.setMnemonic(KeyEvent.VK_X);
```

- If the letter is in the component's text, the first occurrence of that letter will appear underlined.
- If the letter does not appear in the component's text, then no letter will appear underlined.

Mnemonics



Fonts

- Text in a component displays according to their font characteristics:
 - The font is the name of the typeface.
 - The style can be plain, bold, and/or italic.
 - The size is how large the text appears.



Fonts

- A Font object accepts three arguments:
 - The name of the font (String)
 - Java guarantees the following fonts:
 - Dialog, DialogInput, Monospaced, Serif
 - The style is created using Font class constants.
 - `Font.BOLD`, `Font.PLAIN`, and `Font.ITALIC`
 - The size specifies how large the text should appear (int)
- Must be imported: `import java.awt.Font;`

Fonts

- A component's `setFont` method will change the appearance of the text in the component.

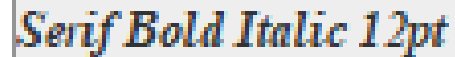
```
testLabel.setFont(new Font("Serif", Font.BOLD, 12));
```



Serif Bold 12pt

- Styles can be combined using the addition operator.

```
testLabel.setFont(new Font("Serif",  
                           Font.BOLD + Font.ITALIC,  
                           12));
```



Serif Bold Italic 12pt

Creating a Canvas

- A ***canvas*** is a component that allows the drawing of lines and shapes.
 - The background of a canvas is transparent.
- Using the AWT libraries, canvases are created using the **Canvas** class.
- Must be imported: `import java.awt.Canvas;`

Creating a Canvas

- The proper way to use a Canvas is to create a subclass of the Canvas class.
 - Similar to how we have been subclassing JFrames.
- At a minimum, you will need to override the Canvas class's paint method.
 - This method is called when the Canvas is created.
 - This method handles drawing any shapes/graphics on the canvas.

Creating a Canvas

- The Canvas class's paint method accepts a Graphics argument.
 - This parameter is what is used to draw the shapes/graphics.
 - The Graphics argument is passed to this method when the Canvas (or a subclass of the Canvas class) is instantiated.
- Needs to be imported: `import java.awt.Graphics;`

Subclassing the Canvas class

```
import java.awt.Canvas;
import java.awt.Graphics;

public class MyCanvas() extends Canvas {

    public void paint(Graphics g) {
        //Called when this class is instantiated.
        //All code for drawing on this canvas
        //goes here.
    }
}
```

Overrides the superclass's
paint method.

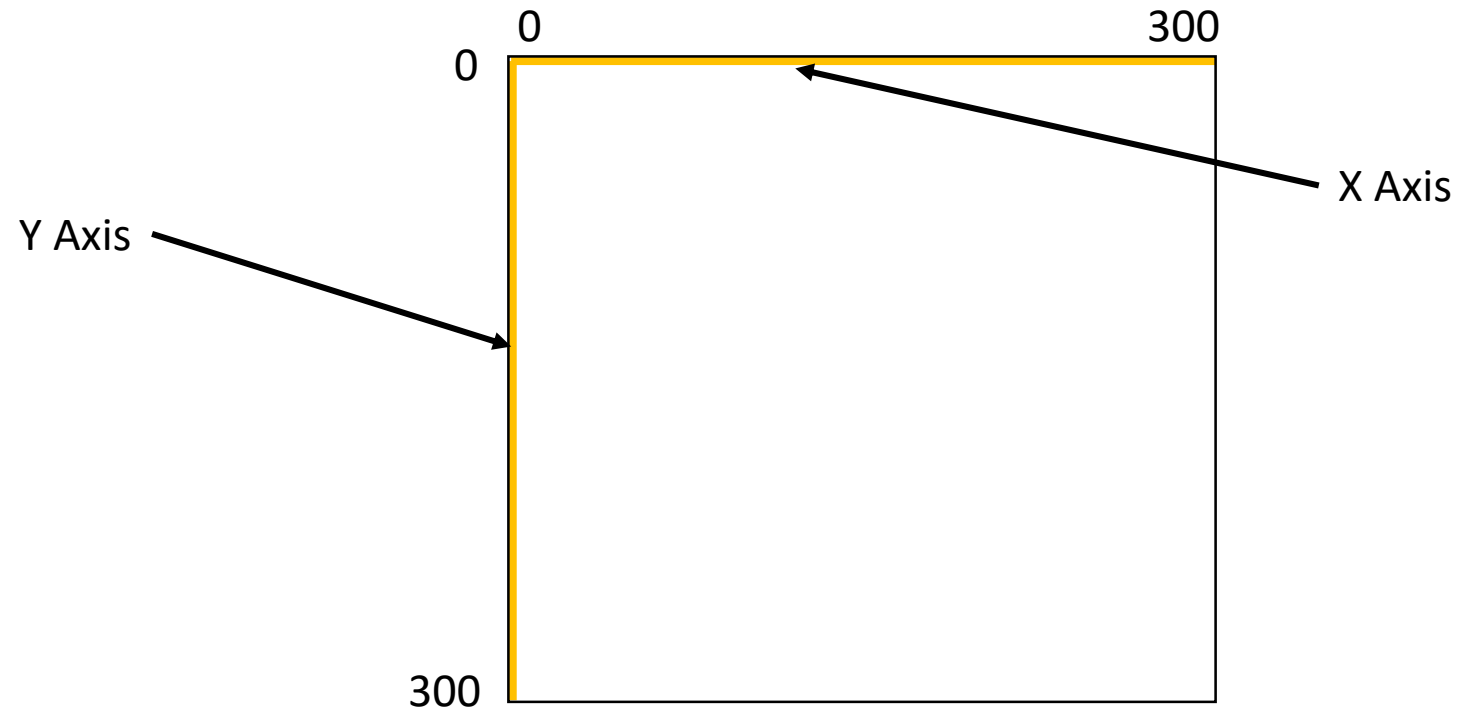
(No constructor is needed)

}

}

Coordinate System

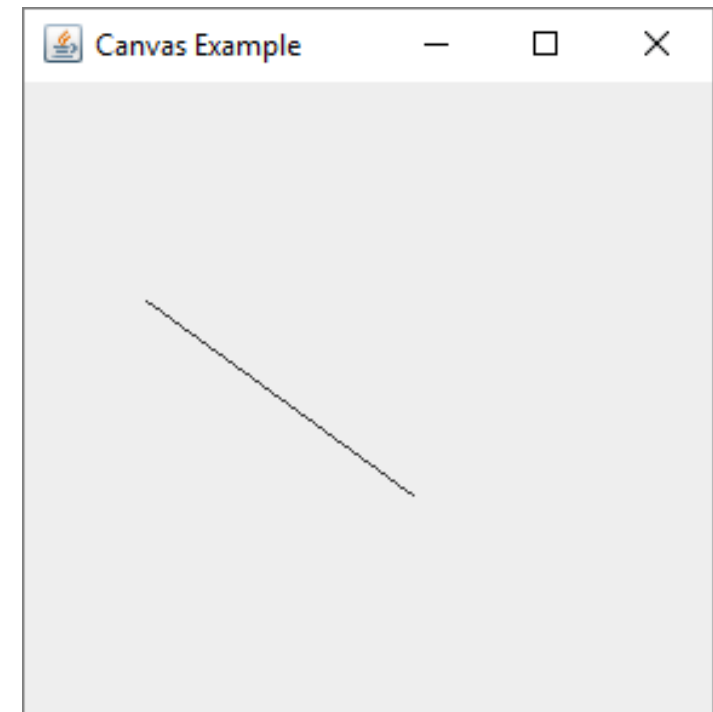
- Shapes and graphics will be drawn on a canvas using coordinates.



Drawing Lines

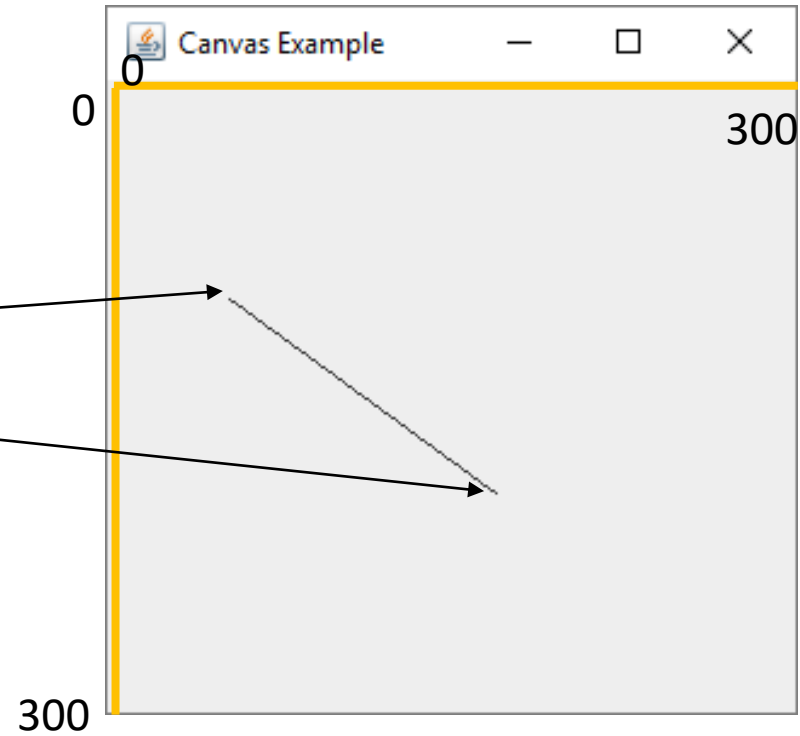
- The Graphics object's drawLine method will draw a straight line.
 - A pair of x and y coordinates (4 ints) are required.

```
import java.awt.Canvas;  
import java.awt.Graphics;  
  
public class MyCanvas() extends Canvas {  
  
    public void paint(Graphics g) {  
        g.drawLine(50, 90,  
                   160, 170);  
    }  
  
}
```



Drawing Lines

```
g.drawLine(50, 90,  
            160, 170);
```



Colors

- By default, lines and shapes are drawn in black.
- The current color of the Graphics object being used to draw the lines and shapes can be changed using its setColor method.
 - Accepts a Color class argument.
- Using the AWT libraries, Color objects are created using the **Color** class.
- Must be imported: `import java.awt.Color;`

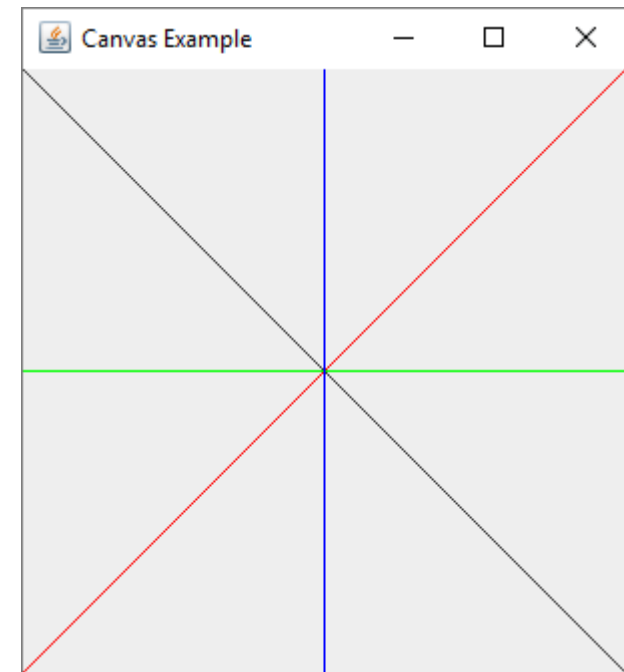
Colors

- It's possible to create custom colors, but the Color class has some colors/constants already defined.
- BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE, and YELLOW.

Colors

- Once the color is changed, it will remain that color until it is changed again.

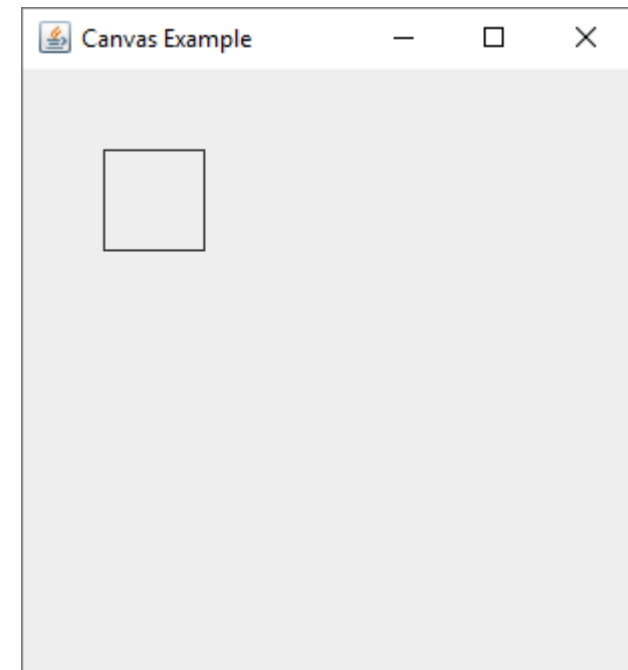
```
public void paint(Graphics g) {  
    g.drawLine(0, 0,  
               300, 300);  
    g.setColor(Color.RED);  
    g.drawLine(0, 300,  
               300, 0);  
    g.setColor(Color.GREEN);  
    g.drawLine(0, 150,  
               300, 150);  
    g.setColor(Color.BLUE);  
    g.drawLine(150, 0,  
               150, 300);  
}
```



Drawing Rectangles

- The Graphics object's drawRect method will draw an unfilled rectangle.
 - A pair of x and y coordinates (2 ints) of the top left corner are required.
 - The width and height (2 ints) are also required.

```
public void paint(Graphics g) {  
    g.drawRect(40, 40,  
               50, 50);  
}
```



Drawing Rectangles

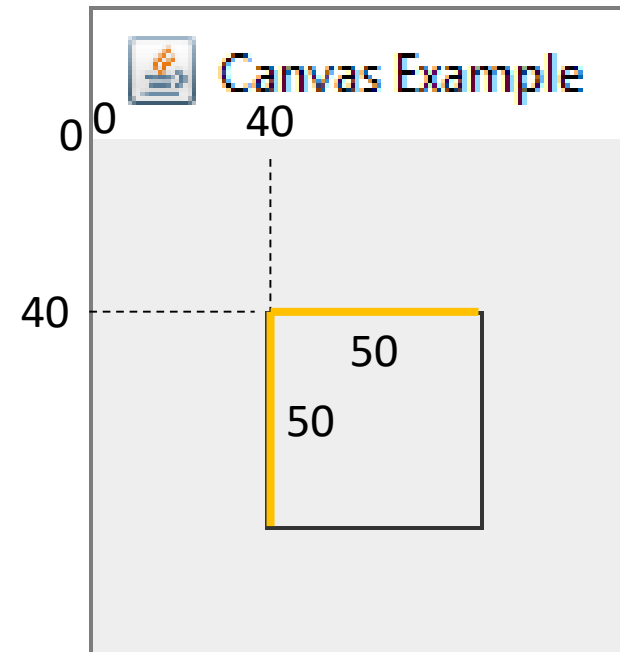
```
public void paint(Graphics g) {
```

```
    g.drawRect(40, 40,  
               50, 50);
```

Diagram illustrating the parameters for `g.drawRect(x, y, width, height)`:

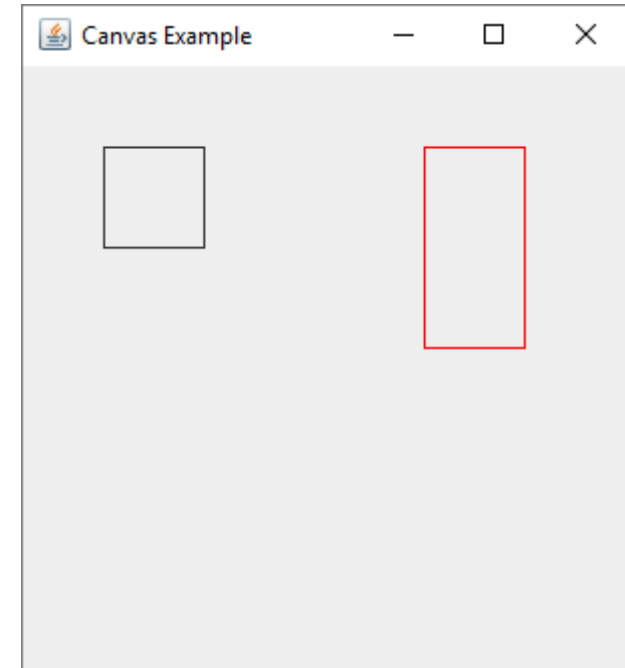
- `x` points to the first `40`.
- `y` points to the second `40`.
- `width` points to the first `50`.
- `height` points to the second `50`.

```
}
```



Drawing Rectangles

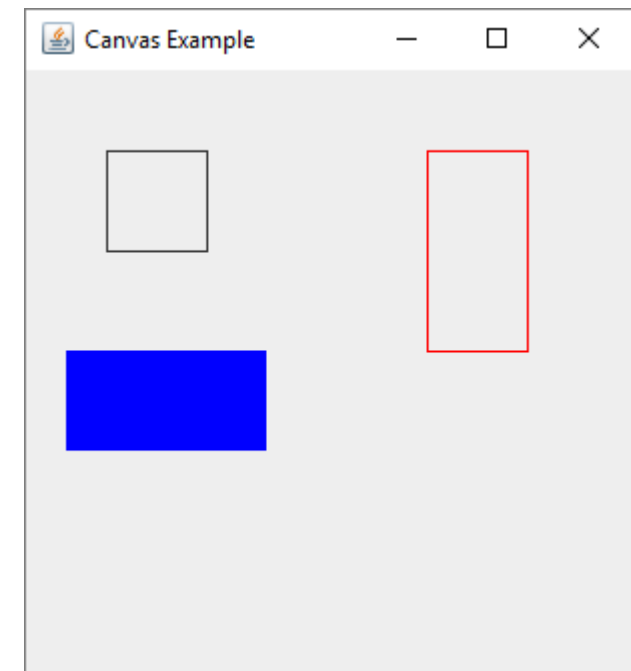
```
public void paint(Graphics g) {  
    g.drawRect(40, 40,  
               50, 50);  
  
    g.setColor(Color.RED);  
    g.drawRect(200, 40,  
               50, 100);  
}
```



Drawing Rectangles

- The Graphics object's fillRect method will draw a filled rectangle.
 - A pair of x and y coordinates (2 ints) of the top left corner are required.
 - The width and height (2 ints) are also required.

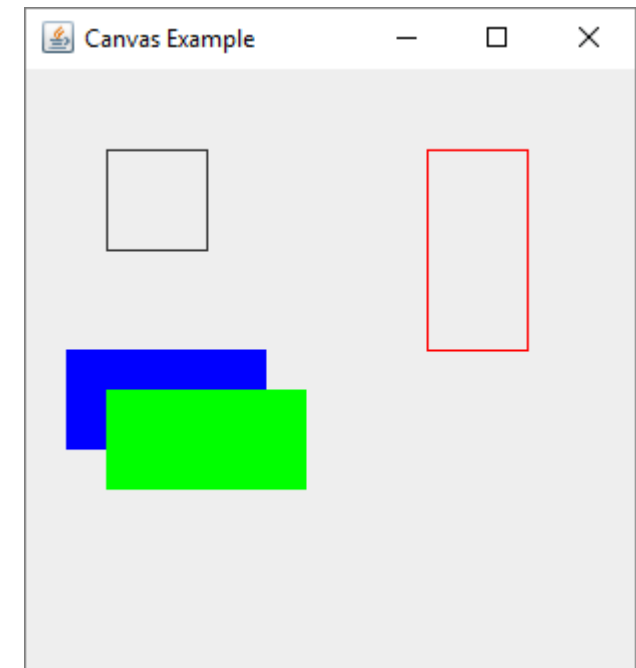
```
public void paint(Graphics g) {  
    g.drawRect(40, 40,  
               50, 50);  
    g.setColor(Color.RED);  
    g.drawRect(200, 40,  
               50, 100);  
    g.setColor(Color.BLUE);  
    g.fillRect(20, 140,  
               100, 50);  
}
```



Drawing Rectangles

- Lines and shapes appear in the order they are drawn.
 - Shapes may overlap ones drawn previously.

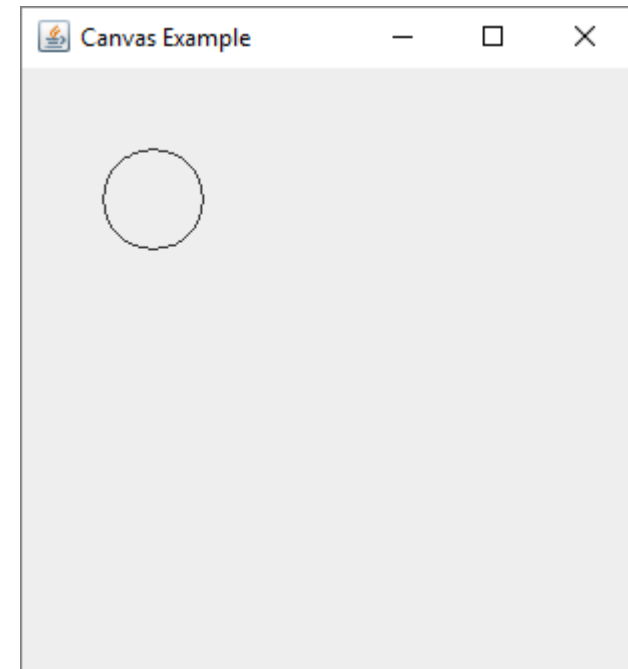
```
public void paint(Graphics g) {  
    g.drawRect(40, 40,  
               50, 50);  
    g.setColor(Color.RED);  
    g.drawRect(200, 40,  
               50, 100);  
    g.setColor(Color.BLUE);  
    g.fillRect(20, 140,  
               100, 50);  
    g.setColor(Color.GREEN);  
    g.fillRect(40, 160,  
               100, 50);  
}
```



Drawing Circles and Ovals

- The Graphics object's drawOval method will draw unfilled circles and ovals.
 - A pair of x and y coordinates (2 ints) of the top left corner are required.
 - The width and height (2 ints) are also required.

```
public void paint(Graphics g) {  
    g.drawOval(40, 40,  
              50, 50);  
}
```



Drawing Circles and Ovals

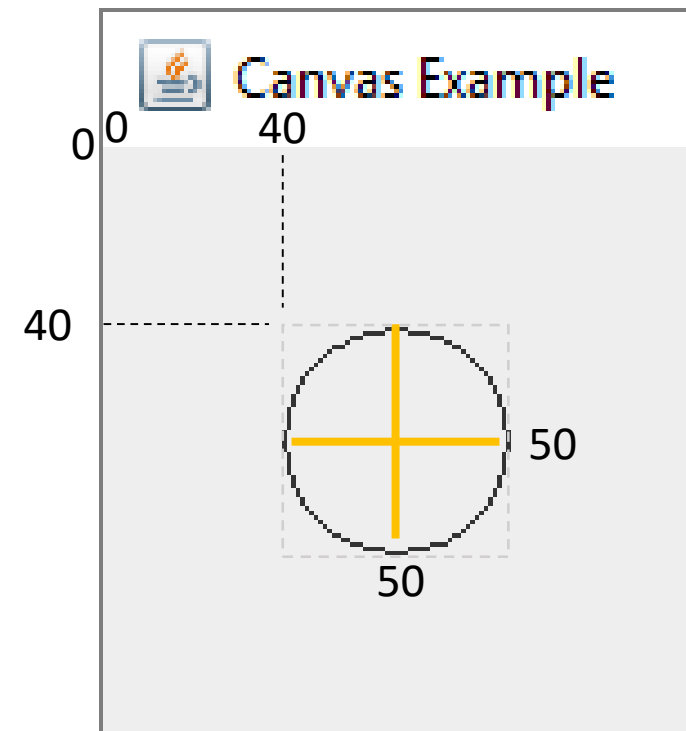
```
public void paint(Graphics g) {
```

```
    g.drawOval(40, 40,  
               50, 50);
```

Diagram illustrating the parameters for `g.drawOval`:

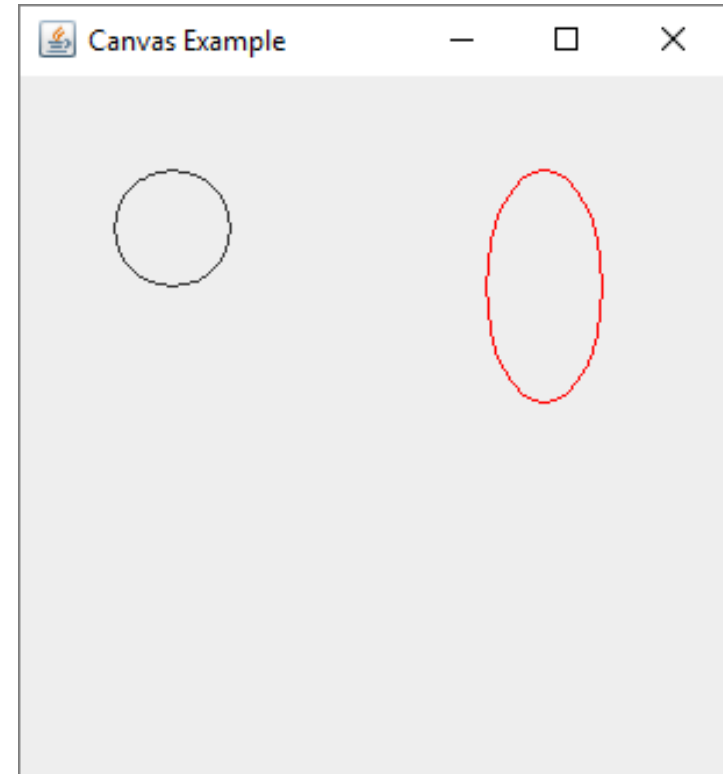
- `40` is the x-coordinate (labeled `x`).
- `40` is the y-coordinate (labeled `y`).
- `50` is the width (labeled `width`).
- `50` is the height (labeled `height`).

```
}
```



Drawing Circles and Ovals

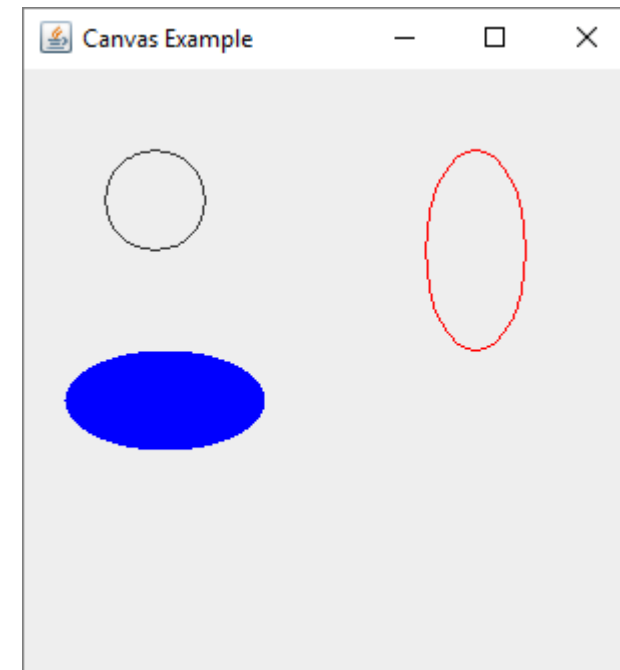
```
public void paint(Graphics g) {  
    g.drawOval(40, 40,  
              50, 50);  
  
    g.setColor(Color.RED);  
    g.drawOval(200, 40,  
              50, 100);  
}
```



Drawing Circles and Ovals

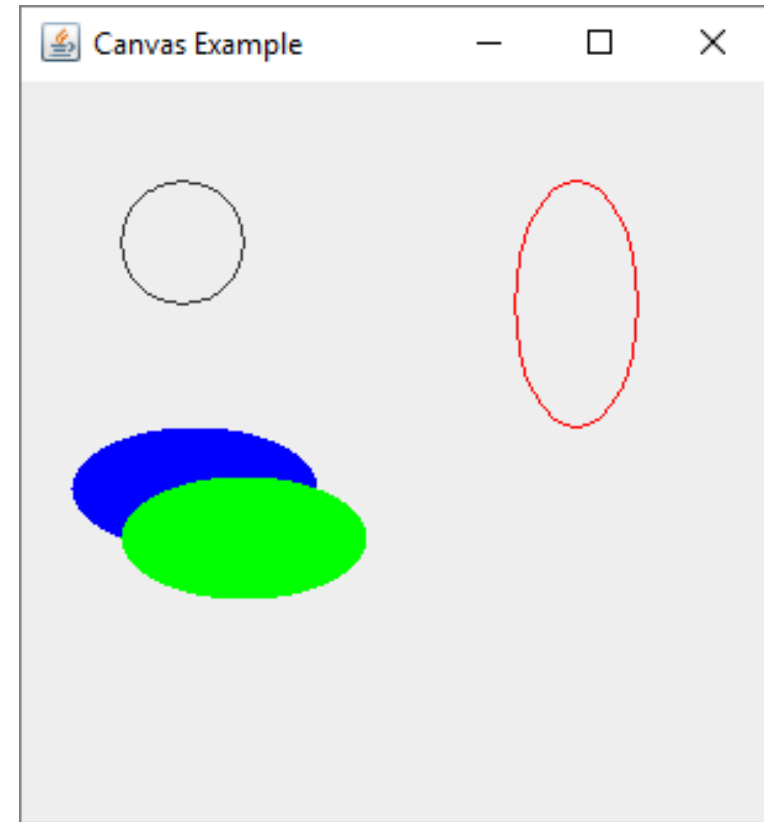
- The Graphics object's fillOval method will draw filled circles and ovals.
 - A pair of x and y coordinates (2 ints) of the top left corner are required.
 - The width and height (2 ints) are also required.

```
public void paint(Graphics g) {  
    g.drawOval(40, 40,  
              50, 50);  
    g.setColor(Color.RED);  
    g.drawOval(200, 40,  
              50, 100);  
    g.setColor(Color.BLUE);  
    g.fillOval(20, 140,  
              100, 50);  
}
```

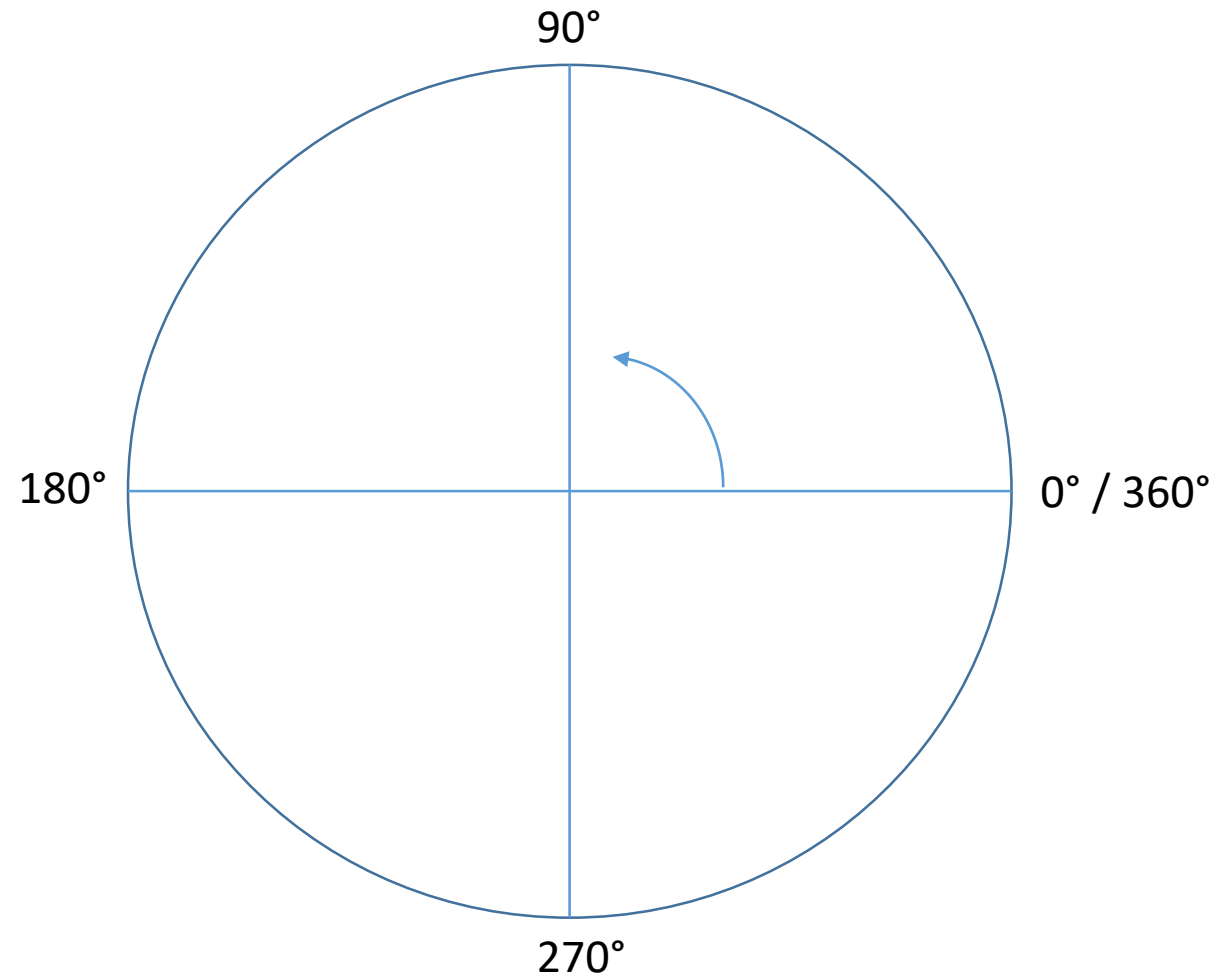


Drawing Circles and Ovals

```
public void paint(Graphics g) {  
    g.drawOval(40, 40,  
              50, 50);  
    g.setColor(Color.RED);  
    g.drawOval(200, 40,  
              50, 100);  
    g.setColor(Color.BLUE);  
    g.fillOval(20, 140,  
              100, 50);  
    g.setColor(Color.GREEN);  
    g.fillOval(40, 160,  
              100, 50);  
}
```



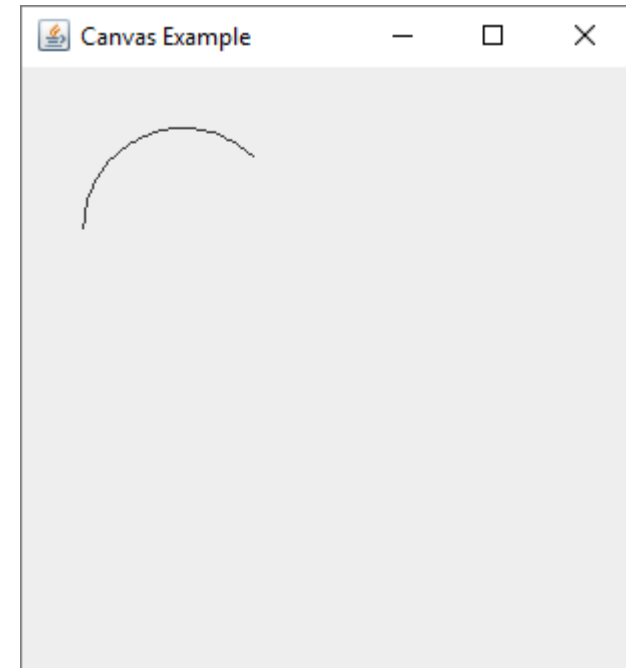
Drawing Arcs



Drawing Arcs

- The Graphics object's drawArc method will draw arcs.
 - A pair of x and y coordinates (2 ints) of the top left corner are required.
 - The width and height (2 ints) are also required.
 - The starting degree (an int)
 - The number of degrees from the start (an int)

```
public void paint(Graphics g) {  
    g.drawArc(30, 30,  
              100, 100,  
              45, 135);  
}
```



Drawing Circles and Ovals

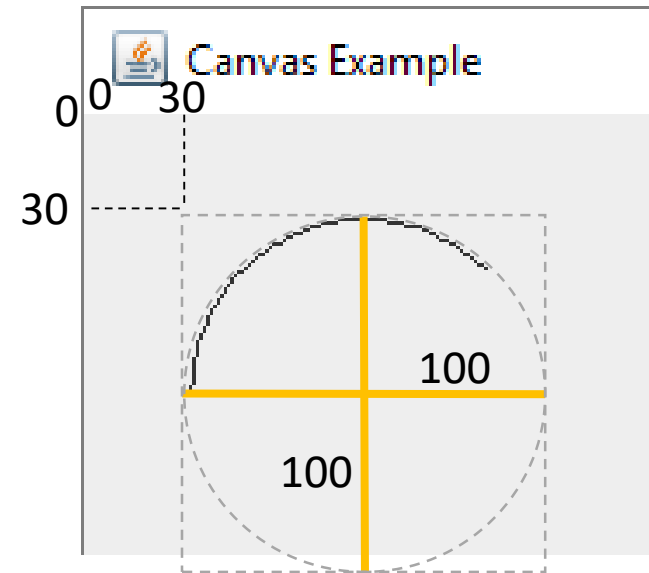
```
public void paint(Graphics g) {
```

```
    g.drawArc(30, 30, 100, 100, 45, 135);
```

Diagram illustrating the parameters for `g.drawArc(x, y, width, height, start, degrees from start)`:

- `x`: 30
- `y`: 30
- `width`: 100
- `height`: 100
- `start`: 45
- `degrees from start`: 135

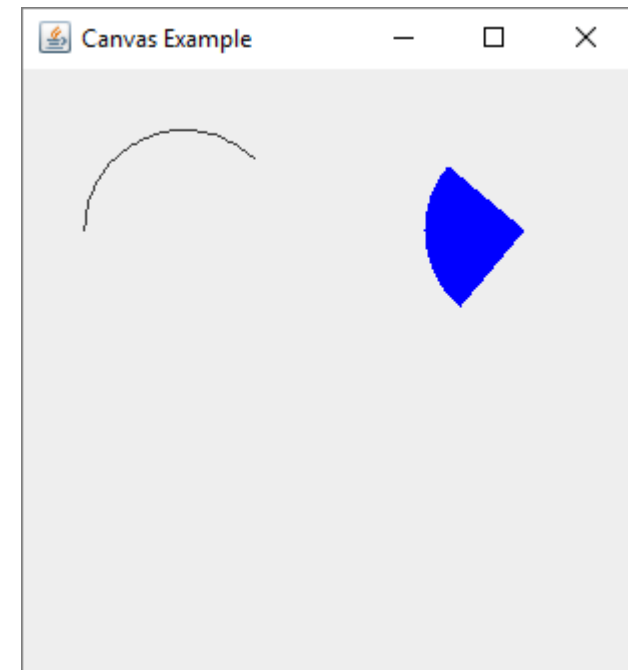
```
}
```



Drawing Arcs

- The Graphics object's fillArc method will draw filled arcs.
 - A pair of x and y coordinates (2 ints) of the top left corner are required.
 - The width and height (2 ints) are also required.
 - The starting degree (an int)
 - The number of degrees from the start (an int)

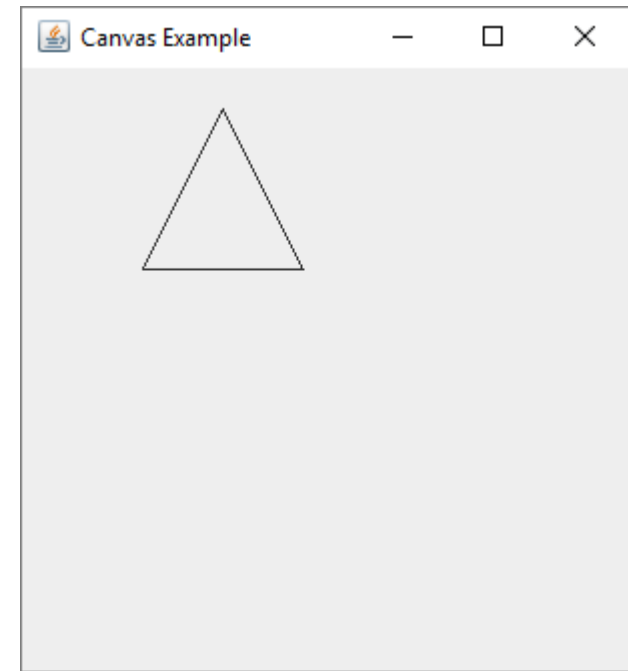
```
public void paint(Graphics g) {  
    g.drawArc(30, 30,  
              100, 100,  
              45, 135);  
    g.setColor(Color.BLUE);  
    g.fillArc(200, 30,  
              100, 100,  
              140, 90);  
}
```



Drawing Polygons

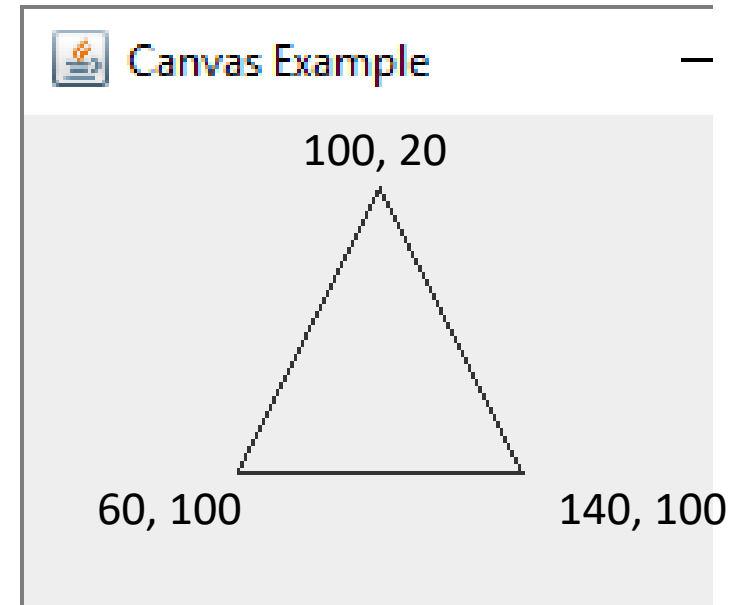
- The Graphics object's drawPolygon method will draw polygons.
 - An array of x coordinates (ints) is required.
 - An array of y coordinates (ints) is required.
 - The number of vertices (an int) is required.

```
public void paint(Graphics g) {  
    int[] shape1_x = {100, 60, 140};  
    int[] shape1_y = {20, 100, 100};  
    g.drawPolygon(shape1_x, shape1_y, 3);  
}
```



Drawing Polygons

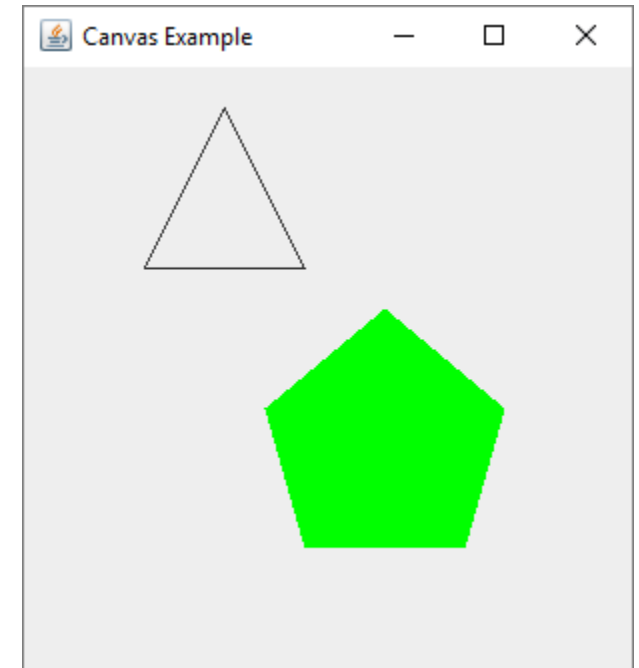
```
public void paint(Graphics g) {  
    int[] shape1_x = {100, 60, 140};  
    int[] shape1_y = {20, 100, 100};  
    g.drawPolygon(shape1_x, shape1_y, 3);  
}
```



Drawing Polygons

- The Graphics object's fillPolygon method will draw filled polygons.
 - An array of x coordinates (ints) is required.
 - An array of y coordinates (ints) is required.
 - The number of vertices (an int) is required.

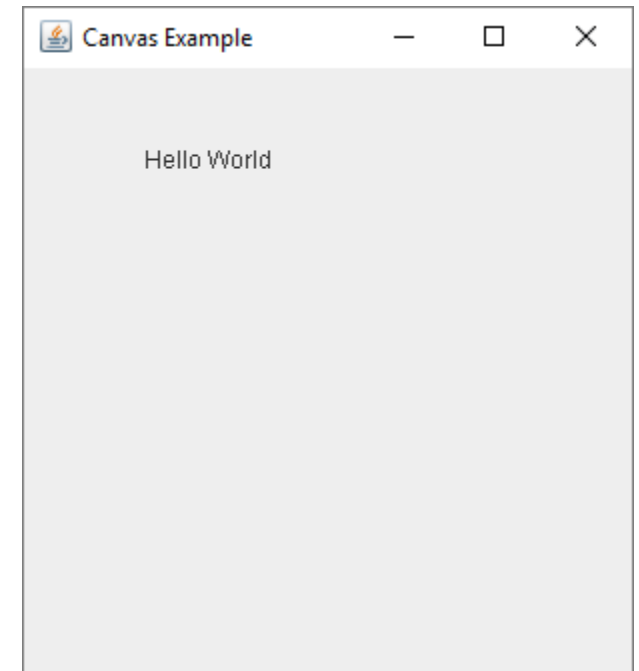
```
public void paint(Graphics g) {  
    int[] shape1_x = {100, 60, 140};  
    int[] shape1_y = {20, 100, 100};  
    g.drawPolygon(shape1_x, shape1_y, 3);  
  
    g.setColor(Color.GREEN);  
    int[] shape2_x = {180, 120, 140, 220, 240};  
    int[] shape2_y = {120, 170, 240, 240, 170};  
    g.fillPolygon(shape2_x, shape2_y, 5);  
}
```



Drawing Text

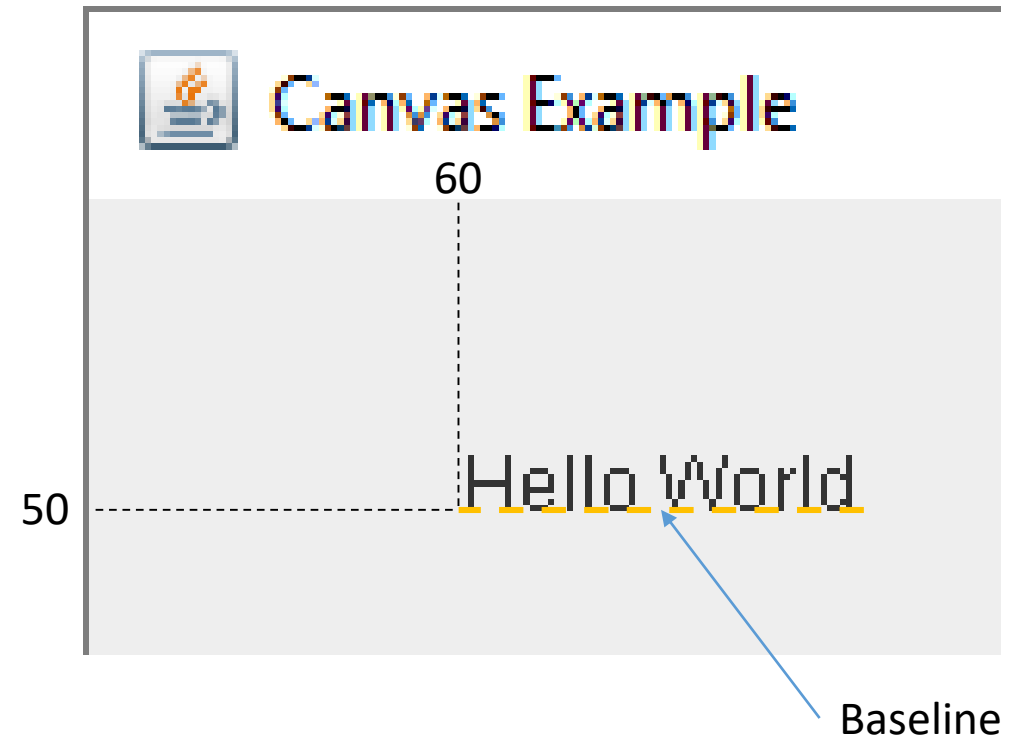
- The Graphics object's drawString method will draw text.
 - The text to draw (a String)
 - A pair of x and y coordinates (2 ints) of the baseline.

```
public void paint(Graphics g) {  
    g.drawString("Hello World", 60, 50);  
}
```



Drawing Text

```
public void paint(Graphics g) {  
    g.drawString("Hello World", 60, 50);  
}
```



Colors and Fonts

- By default, text is drawn in black.
- The current color of the Graphics object being used to draw the text can be changed using its setColor method.
 - Just like with lines and shapes.
- By default, the text's font is Dialog.
- The current font of the Graphics object being used to draw the text can be changed using its setFont method.

Colors and Fonts

```
public void paint(Graphics g) {  
    g.drawString("Hello World", 60, 50);  
  
    g.setColor(Color.GREEN);  
    g.setFont(new Font("Serif", Font.BOLD, 16));  
    g.drawString("Hello World", 200, 50);  
}
```

