***Chapter 25***

## ***Working with Frame Windows and other GUI Components***

* After the applet, the type of window most often created is derived from **Frame**
* It is used to create child windows within applets

Creates a standard window without title

Constructors: Frame( )

Frame(String *title*)

With title

There are several methods that you will use when working with **Frame** windows:

**Dimension** is an object in which the

width is stored in **width**, and the

height is stored in **height**

void resize(int *newWidth*, int *newHeight*)

void resize(Dimension *newSize*)

You can obtain the current size of a window by calling **size( )**. The **size( )**

method returns **Dimension** (the current size of the window contained within

the **width** and **height** fields).

Dimension size( )

void hide( )

Once a Frame window has been created, it will not be visible until you call

**show( )**. To hide a window (remove it from view), call **hide( )**.

void show( )

void setTitle(String *newTitle*)***Responding the WINDOW\_DESTROY Event***

*newTitle* is the new title for the window

All objects derived from **Component** receive events, such as mouse clicks and keypresses. This includes frame windows, which can override event methods such as **mouseDown( )** (as described last week). There is one additional event that your frame windows need to handle --- the one generated when the window is closed.

To do so, your applet must override its main event handler, **handleEvent( )**. **handleEvent( )** is the top-level event handler for your window. All events that relate to your window are routed through it.

As we've seen, *evtObj* contains the **Event** object that describes the event. For mouse and keyboard events, **handleEvent( )** simply passes *evtObj* along to the appropriate mouse or keyboard handler. This is why we did not need to use **handleEvent( )** when creating applets last week. However, there are several other types of events that can only be handled by **handleEvent( )** directly. One of these occurs when the user closes the window.

Constructor: **boolean handleEvent(Event evtObj)**

When overriding **handleEvent( )**, you must return **true** if you handle the event. Otherwise, you must pass the event along to the superclass implementation by calling **super.handleEvent( )**. With **handleEvent( )**, you can determine what event has occurred by examining the **id** field of the **Event** object passed to it as an argument. The **Event** class defines several constants that are the ID codes for the various events. Each time an event is generated, its ID is put in **id**. The only ID we need to use for the purposes of this chapter is **WINDOW\_DESTROY**. Here is a skeleton of an overriden **handleEvent( )** method that handles **WINDOW\_DESTROY**.

public boolean handleEvent(Event evtObj) {

if (evtObj.id == Event.WINDOW\_DESTROY) {

// respond to event

return true;

}

return super.handleEvent (evtObj);

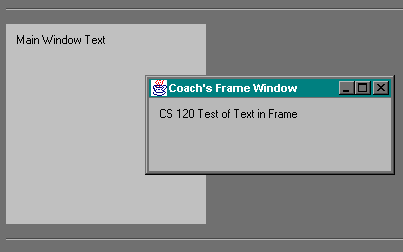
}

How you respond to the **WINDOW\_DESTROY** message depends upon whether the window is an application's main window or a child window of an applet:

* Applet: it must remove the window from the screen, using **hide( )**. If you don't do this, then the window is never fully removed from the system.
* Top-level windows: you must exit the Java run-time system by calling **System.exit( )**.

***Creating a Frame Window in an Applet***

While it is possible to create a window by creating an instance of **Frame**, you shouldn't do this. For example, you will not be able to receive or process events that occur within it, or easily output information to it. Most of the time, you will create a subclass of **Frame**. Doing so lets you override **Frame**'s methods and event handling.



To create a new frame window from within an applet:

* create a subclass of **Frame,**
* override any of the standard window methods, such as **init( )**, **start( )**, **stop( )**, and **paint( )**,
* override **handleEvent( )** so that it hides the window when the **WINDOW\_DESTROY** event is received.

Once you have defined a **Frame** subclass, you can create an object of that class. This causes a frame window to come into existence, but it will not be initially visible. You make it visible by calling **show( )**. When created, the window is given a default height and width. You can set the size of the window explicitly by calling the **resize( )** method.

The following applet creates a subclass of **Frame** called **MyFrame**. A window of this subclass is instantiated within the **init( )** method of **CreateAFrame**. Notice that **CreateAFrame** calls **Frame**'s constructor. This causes a standard frame window to be created with the title passed in **title**.

This example overrides the applet window's **start( )** and **stop( )** methods so that they show and hide the child window, respectively. This causes the window to be removed automatically when you terminate the applet, when you close the window, or, if using a browser, when you move to another page. It also causes the child window to be shown when the browser returns to the applet.

// Create a child frame window from within an applet

import java.awt.\*;

import java.applet.\*;

// Create a subclass of Frame

class MyFrame extends Frame {

MyFrame(String title) {

super(title);

}

// Hide window when terminated by user

public boolean handleEvent(Event evtObj) {

if (evtObj.id == Event.WINDOW\_DESTROY) {

hide( );

return true;

}

return super.handleEvent(evtObj);

}

public void paint(Graphics g) {

g.drawString("CS 120 Test of Text in Frame", 10, 40);

}

}

// Create the applet window.

public class CreateAFrame extends Applet {

MyFrame f;

// Create a frame window

public void init( ) {

f = new MyFrame("Coach's Frame Window");

f.show( );

f.resize(250, 100);

}

// Remove frame window when stopping applet

public void stop( ) {

f.hide( );

}

// Show frame window when starting applet

public void start( ) {

f.show( );

}

// Display msg in applet window

public void paint(Graphics g) {

g.drawString("Main Window Text", 10, 20);

}

}

***Events in Framed Windows***

Frame is a subclass of Component. Therefore, it inherits all of the capabilities of a Component, including the ability to paint, repaint, and handle events. In fact, whenever an event occurs in a window, the event is directed to that window's corresponding event handling routines --- each window handles its own events!

***Scroll Bars***

* Used to select continuous values
* May be horizontal or vertical
* Has several individual parts (i.e., scroll arrows, scroll box and scroll bar)
* Slider box can be dragged to a new position, or repositioned using arrowheads

Constructors:

**Scrollbar( )**

**Scrollbar(int *style*)**

**Scrollbar(int *style*, int *initialValue*, int *scrollboxSize*, int *min*, int *max*)**

*style:*

|  |  |
| --- | --- |
| Scrollbar.VERTICAL | Creates a vertical scroll bar |
| Scrollbar.HORIZONTAL | Creates a horizontal scroll bar |

*initialValue:* Initial value of scroll bar

*scrollboxSize*: The number of units represented by the height of the scroll box

*min and max:* The minimum and maximum values for the scroll bar

Additional constructors that work with scroll bars:

**void setValues(int *initialValue*, int *scrollboxSize*, int *min*, int *max*)**

**int getValue( )**

**void setValue(int *newValue*)**

**int getMinimum( )**

**int getMaximum( )**

**void setLineIncrement(int *newIncr*)**

**void setPageIncrement(int *newIncr*)**

Scroll bar events are not passed to **action( )**, but are instead processed by **handleEvent( )**.

* The **target** field of the **Event** object that is passed to **handleEvent( )** will contain a reference to the scroll bar causing the event.
* The **Event**'s **arg** field is an **Integer** object containing the current scroll bar value.
* The **id** field will contain a value describing the event.

import java.awt.\*;

import java.applet.\*;

public class ScrollBarDemo extends Applet {

String msg = "";

Scrollbar verticalScrollBar, horizontalScrollBar;

public void init( ) {

int width = Integer.parseInt(getParameter("width"));

int height = Integer.parseInt(getParameter("height"));

verticalScrollBar = new Scrollbar(Scrollbar.VERTICAL, height/2, 1, 0, height);

horizontalScrollBar = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, width);

add(verticalScrollBar);

add(horizontalScrollBar);

}

// Repaint the window whenever a scroll bar is moved

public boolean handleEvent(Event evtObj) {

if (evtObj.target instanceof Scrollbar) {

repaint( );

return true;

}

return super.handleEvent(evtObj);

}

// Update scroll bars to reflect mouse dragging

public boolean mouseDrag(Event evtObj, int x, int y) {

verticalScrollBar.setValue(y);

horizontalScrollBar.setValue(x);

repaint( );

return true;

}

// Display the current values of the scroll bars

public void paint(Graphics g) {

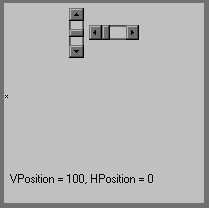
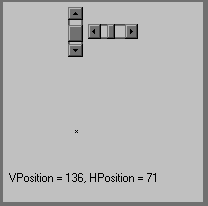
msg = "VPosition = " + verticalScrollBar.getValue( ) +

", HPosition = " + horizontalScrollBar.getValue( );

g.drawString(msg, 6, 180);

// show current mouse drag position

g.drawString("\*", horizontalScrollBar.getValue( ), verticleScrollBar.getValue( ));

} }

***TextAreas***

**TextArea**s are useful when you want to display a section of text that exceeds a single line. **TextArea** constructors are:

TextArea( )

TextArea(int *numLines*, int *numChars*)

TextArea(String *str*)

TextArea(String *str*, int *numLines*, int *numChars*)

* numLines specifies the height (in # of lines) of the **TextArea**.
* numChars specifies the width of the **TextArea**.

Text can be manipulated in the text area using:

void appendText(String *str*)

void insertText(String *str*, int *index*)

void replaceText(String *str*, int *startIndex*, int *endIndex*)

An example of **TextArea**s can be seen in my BorderLayoutDemo handout from last week.

***Insets***

To leave space between the container that holds your components and the window that contains it, use insets. This is done by overriding the **insets( )** method contained in **Container**. The constructor for **Insets** is:

Insets(int *top*, int *left*, int *bottom*, int *right*)

*top*, *left*, *bottom*, and *right* specify the amount of space between the container and its enclosing window.

Insets insets( )

The **insets( )** method has this form:

When overriding **insets( )**, you must return a new **Insets** object containing the spacing you desire.

Here is last weeks BorderLayoutDemo with insets:// Demonstrate BorderLayout

import java.awt.\*;

import java.applet.\*;

import java.util.\*;

public class InsetsDemo extends Applet {

public void init( ) {

setBackground(Color.yellow);

setLayout(new BorderLayout( ));

add("North", new Button("This is across the top."));

add("South", new Label("The footer message might go here"));

add("East", new Button("Right"));

add("West", new Button("Left"));

String msg = "The reasonable man adapts " +

"himself to the world;\n" +

"the unreasonable one persists in " +

"trying to adapt the world to himself.\n" +

"Therefore all progress depends " +

"on the unreasonable man.\n\n";

add("Center", new TextArea(msg));

}

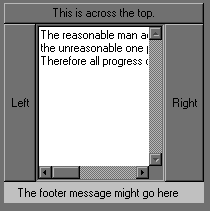
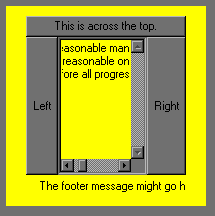
// add insets

public Insets insets( ) {

return new Insets(10, 20, 10, 20);

}

}



***GridBagLayouts***

* The most complex of the predefined layout managers
* Similar to **GridLayout**
* More flexible since components can vary in size and can be added in any order
* Requires that a **GridBagConstraints** object be constructed.

**GridBagConstraints** tells the system how a component is to be laid out

|  |  |
| --- | --- |
| GridBagConstraints  Instance Variable | Description |
| **anchor** | Specifies where the component is placed (*NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, NORTHWEST, CENTER*) - default is *CENTER*. |
| **fill** | Specifies how much of the component's area is occupied (*NONE, VERTICAL, HORIZONTAL*, or *BOTH*) - default is *NONE*. |
| **gridx** | Column in which the component is placed. Combined with gridy, this represents where the upper-left corner of the component is placed. (0,0) is the top/left-most postion. |
| **gridy** | Row in which the component is placed |
| **gridwidth** | # of columns component occupies |
| **gridheight** | # of rows component occupies |
| **weightx** | The portion of extra space to allocate horizontally. The components in a row can become "taller" when extra space becomes available due to a resize |
| **weighty** | The "wider" counterpart to weightx |

// Demonstrate GridBagLayout

import java.applet.\*;

import java.awt.\*;

public class GridBagDemo extends Applet {

private GridBagLayout gbLayout;

private TextArea ta;

private Choice cb;

private Button yes = new Button("Yes");

private Button no = new Button("No");

private Button maybe = new Button("Undecided");

private GridBagConstraints gbConstraints = new GridBagConstraints( );

private Label namep = new Label("Name: ", Label.RIGHT);

private Label passp = new Label("Password: ", Label.RIGHT);

private TextField name = new TextField(12);

private TextField pass = new TextField(8);

public void init( ) {

gbLayout = new GridBagLayout( );

setLayout(gbLayout);

// create a text area to be displayed

ta = new TextArea("The reasonable man adapts " +

"himself to the world;\n" +

"the unreasonable one persists in " +

"trying to adapt the world to himself.\n" +

"Therefore all progress depends " +

"on the unreasonable man.\n\n", 4, 10);

// set password echo character

pass.setEchoCharacter('\*');

// create a choice button to be displayed

cb = new Choice( );

cb.addItem("Windows 95");

cb.addItem("Windows NT");

cb.addItem("Solaris");

cb.addItem("MacOS");

// text area-Demo 2nd time with // infront of line below

depth

width

gbConstraints.fill = GridBagConstraints.BOTH;

addComponent(ta, gbLayout, gbConstraints, 0, 0, 1, 3);

// choice button

row

column

addComponent(cb, gbLayout, gbConstraints, 0, 1, 1, 1);

// buttons

addComponent(yes, gbLayout, gbConstraints, 1, 1, 1, 1);

addComponent(no, gbLayout, gbConstraints, 2, 1, 1, 1);

addComponent(maybe, gbLayout, gbConstraints, 3, 1, 1, 1);

// text fields

addComponent(name, gbLayout, gbConstraints, 3, 0, 1, 1);

addComponent(pass, gbLayout, gbConstraints, 4, 0, 1, 1);

}

// User pressed Enter

public boolean action(Event evtObj, Object arg) {

if (evtObj.target instanceof TextField) {

repaint( );

return true;

}

return false;

}

public void paint(Graphics g) {

g.drawString("Name: " + name.getText( ), 6, 180);

g.drawString("Password: " + pass.getText( ), 6, 195);

}

private void addComponent(Component c, GridBagLayout g, GridBagConstraints gc,

int row, int column, int width, int height)

{ gc.gridx = column;

gc.gridy = row;

gc.gridwidth = width;

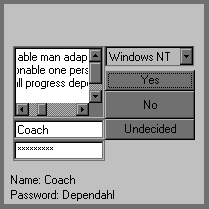
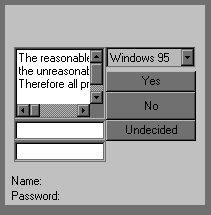
gc.gridheight = height;

g.setConstraints(c, gc);

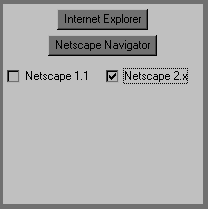
add(c);

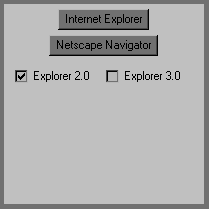
}

}



***CardLayouts***

**CardLayout** stores several different layouts. Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time. This is useful for user interfaces that have optional components which can be dynamically enabled and disabled upon user input. You can prepare the other layouts and have them hidden, ready to be activated when needed.



Constructors:

CardLayout( )

CardLayout(int *horz*, int *vert*)

*horz* and *vert* enable the specification of horizontal and vertical spacing.

The cards that form the deck are typically objects of type **Panel**. This requires you to:

* create a panel that contains the deck, and a panel for each card in the deck
* add the components that form each card to the deck panel
* then add these panels to the panel for which **CardLayout** is the layout manager
* finally, you add this panel to the main applet panel

You must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck.

When card panels are added to a panel, they are usually given a name. Most of the time, you will use this form of **add( )** when adding cards to a panel:

Component add(String *name*, Component *panelObj*);

Here, *name* is the name of the card whose panel is specified by *panelObj*. A reference to the component being added is returned. The advantage of naming each card is that you can bring a card to the surface by specifying its name.

After you have created a deck, you program activates a card by calling one of the following methods defined by **CardLayout**:

CardLayout Methods:

void first(Container *deck*)

void last(Container *deck*)

void next(Container *deck*)

void previous(Container *deck*)

void show(Container *deck*, String *cardName*)

Here, *deck* is a reference to the container (usually a panel) that holds the cards, and *cardName* is the name of a card. Calling **first( )** causes the 1st card in the deck to be shown. To show the next card, call **next( )**, and so on.

Here is an example that creates a two-level card deck that allows the user to select a browser. Netscape-based browsers are displayed in one card, Internet Explorer in another:

import java.awt.\*;

import java.applet.\*;

public class CardLayoutDemo extends Applet {

Checkbox explorer2, explorer3, netscape2, netscape3;

Panel browserCards;

CardLayout cardLO;

public void init( ) {

Button Explorer = new Button("Internet Explorer");

Button Netscape = new Button("Netscape Navigator");

add(Explorer);

add(Netscape);

cardLO = new CardLayout( );

browserCards = new Panel( );

browserCards.setLayout(cardLO); // set panel layout to card layout

explorer2 = new Checkbox("Explorer 2.0", null, true);

explorer3 = new Checkbox("Explorer 3.0");

netscape1 = new Checkbox("Netscape 1.1");

netscape2 = new Checkbox("Netscape 2.x");

Panel ExplorerPanel = new Panel( ); // add Explorer check boxes to a panel

ExplorerPanel.add(explorer2);

ExplorerPanel.add(explorer3);

Panel NetscapePanel = new Panel( ); // add Netscape check boxes to a panel

NetscapePanel = new Panel( );

NetscapePanel.add(netscape1);

NetscapePanel.add(netscape2);

browserCards.add("Exp", ExplorerPanel);

browserCards.add("Nav", NetscapePanel);

add(browserCards); // add cards to man applet panel

}

public boolean mouseDown(Event evtObj, int x, int y) { // Cycle through panels

cardLO.next(browserCards);

return true;

}

public boolean action(Event evtObj, Object arg) { // Display panel selected by button

if (evtObj.target instanceof Button) {

if (arg.equals("Internet Explorer"))

cardLO.show(browserCards, "Exp");

else if (arg.equals("Netscape Navigator"))

cardLO.show(browserCards, "Nav");

return true;

}

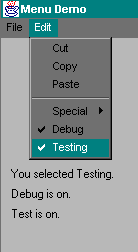
return false;

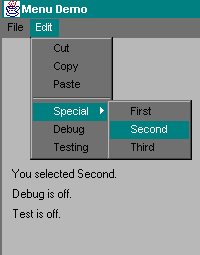
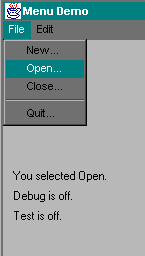
}

}

***Menu Bars and Menus***

A top-level window can have a menu bar associated with it. A menu bar displays a list of choices. Each is associated with a drop-down menu implemented in Java using: **MenuBar**, **Menu**, and **MenuItem**.

* menu bars contain one or more **Menu** objects
* each object contains a list of **MenuItem** objects
* each **MenuItem** object represents an item that the user can select
* since **Menu** is a subclass of **MenuItem**, a hierarchy of nested submenus can be created
* checkable menu items can be included using **CheckboxMenuItem**



To create a menu bar:

* 1st create an instance of MenuBar
* next, create instance of Menu that define the selections

Menu Constructors:

Menu(String *optionName*)

Menu(String *optionName*, boolean *removable*)

*optionName* specifies the name of the menu selection. If *removable* is true, the pop-up menu can be removed and allowed to float free. Otherwise, it will remain attached to the menu bar.

Individual menu items are of type **MenuItem**:

MenuItem Constructor:

MenuItem(String *itemName*)

*Menu Methods:*

|  |  |
| --- | --- |
| void disable( ) | Disables a menu item |
| void enable( ) | Enables a menu item |
| boolean isEnabled( ) | Determines a menu item's state |
| void setLabel(String *newName*) | Changes name of the invoking menu item |
| String getLabel( ) | Gets menu item name |

You can create a checkable menu item by using a subclass of **MenuItem** called **CheckboxMenuItem**:

CheckboxMenuItem(String *itemName*)

Here, itemName is the name shown in the menu. Checkable items operate as toggles. Each time one is selected, its state changes. You can obtain the status of a checkable item by calling **getState( )**. you can set it to a state using **setState( )**:

boolean getState( )

void setState(boolean *checked*)

Menu items are added using:

MenuItem add(MenuItem *item*)

Once you have added all items to a Menu object, you can add that object to the menu bar by using:

Menu add(Menu *menu*)

Menus only generate events when an item of type **MenuItem** is selected. They do not generate events when a menu bar is accessed to display a drop-down menu. Each time a menu item is selected, **action( )** is called. The target field of its **Event** parameter contains a reference to the item that generated the action. Its **Object** parameter contains a reference to the string that is the name of the option. Usually either value may be used to identify the selection. The following example adds a series of nested menus to a pop-up window:

// Illustrate menus

import java.awt.\*;

import java.applet.\*;

// Create a subclass of Frame

class MenuFrame extends Frame {

String msg = "";

CheckboxMenuItem debug, test;

MenuFrame(String title) {

super(title);

// create menu bar and add it to frame

MenuBar mbar = new MenuBar( );

setMenuBar(mbar);

// create menu bar and add it to frame

// create the menu items

Menu file = new Menu("File");

file.add(new MenuItem("New … "));

file.add(new MenuItem("Open …"));

file.add(new MenuItem("Close … "));

file.add(new MenuItem("-"));

file.add(new MenuItem("Quit … "));

mbar.add(file);

Menu edit = new Menu("Edit");

edit.add(new MenuItem("Cut"));

edit.add(new MenuItem("Copy"));

edit.add(new MenuItem("Paste"));

edit.add(new MenuItem("-"));

Menu sub = new Menu("Special");

sub.add(new MenuItem("First"));

sub.add(new MenuItem("Second"));

sub.add(new MenuItem("Third"));

edit.add(sub);

// these are checkable menu items

debug = new CheckboxMenuItem("Debug");

edit.add(debug);

test = new CheckboxMenuItem("Testing");

edit.add(test);

mbar.add(edit);

}

// Hide window when terminated by user

public boolean handleEvent(Event evtObj) {

if (evtObj.id == Event.WINDOW\_DESTROY) {

hide( );

return true;

}

return super.handleEvent(evtObj);

}

// Display men choices

public boolean action(Event evtObj, Object arg) {

if (evtObj.target instanceof MenuItem) {

msg = "You selected ";

if (arg.equals("New…"))

msg += "New.";

else if (arg.equals("Open…"))

msg += "Open.";

else if (arg.equals("Close…"))

msg += "Close.";

else if (arg.equals("Quit…"))

msg += "Quit.";

else if (arg.equals("Edit"))

msg += "Edit.";

else if (arg.equals("Cut"))

msg += "Cut.";

else if (arg.equals("Copy"))

msg += "Copy.";

else if (arg.equals("Paste"))

msg += "Paste.";

else if (arg.equals("First"))

msg += "First.";

else if (arg.equals("Second"))

msg += "Second.";

else if (arg.equals("Third"))

msg += "Third.";

else if (arg.equals("Debug"))

msg += "Debug.";

else if (arg.equals("Testing"))

msg += "Testing.";

repaint( );

return true;

}

return false;

}

public void paint(Graphics g) {

g.drawString(msg, 10, 140);

if (debug.getState( ))  
 g.drawString("Debug is on.", 10, 160);

else g.drawString("Debug is off.", 10, 160);

if (test.getState( ))  
 g.drawString("Test is on.", 10, 180);

else g.drawString("Test is off.", 10, 180);

}

}

// Create frame window

public class MenuDemo extends Applet {

Frame f;

public void init( ) {

f = new MenuFrame("Menu Demo");

int width = Integer.parseInt(getParameter("width"));

int height = Integer.parseInt(getParameter("height"));

resize(width,height);

f.show( );

}

public void start( ) {

f.show( );

}

public void stop( ) {

f.hide( );

}

}

*Programming Assignment for Chapter 25*

Write a color chooser program containing three **Scrollbar** objects and three **TextField** objects. Each **Scrollbar** represents the values from 0 to 255 for the red, green and blue parts of a color. Use the red, green and blue values as the arguments to the **Color** constructor to create a new **Color** object. Display the current value of each **Scrollbar** in the corresponding **TextField**. When the user changes the value of the **Scrollbar**, the **TextField** should be changed accordingly. Display the current **Color** value. This could be done by adding a panel, and manipulating its color, or by changing the background color of your applet area.

*Extra Credit:* Modify the above program to allow the user to type an integer value into a **TextField** to set the red, green or blue value. When the user presses Enter in the **TextField**, the corresponding **Scrollbar** should be set to the appropriate value. Also, manipulate the color of the individual scrollbars, making them individually display their proportional shades of red, green, and blue.

Date Due 🡪 At 5PM, the last day of scheduled classes (check the bottom of your syllabus). No assignments will be accepted after this date.