***Chapter 14***

## ***Basic Graphical User Interface Components***

There is a large amount of material in the next few chapters. My notes will provide you with many examples of different components. However, your primary content responsibility will be on components associated with the programming assignments. For this reason, during the next few classes, I will primarily focus on the components most relevant to your assignments, skipping material that is interesting, yet not as relevant.

*Using Buttons*

The most widely used control is the push button. A push button is a component that contains a label and that generates an event when it is pressed.



Push buttons are objects of type **Button**. **Button** defines these two constructors:

 Button( )

 Button(String *str*)

After a button has been created, you can set its label by calling **setLabel( )**. You can retrieve its label by calling **getLabel( )**. These methods are as follows:

 void setLabel(String *str*)

 String getLabel( )

* Each time a button is pressed, **action( )** is called.
* The target field of its **Event** parameter contains a reference to the button that generated the action.
* Its **Object** parameter contains a reference to the string that is the label of the button.
* Usually, either value may be used to identify the button, as you will see in examples that follow.

Here is an example that creates three buttons labeled "Yes," "No," and "Undecided." Each time one is pressed, a message is displayed that reports which button has been pressed. In this version, the label of the button is used to determine which button has been pressed.

// Demonstrate Buttons

import java.awt.\*;

import java.applet.\*;

/\*

 <applet code="ButtonDemo" width=250 height=150>

 </applet>

\*/

public class ButtonDemo extends Applet {

 String msg = "";

 public void init( ) {

 Button yes = new Button("Yes");

 Button no = new Button("No");

 Button maybe = new Button("Undecided");

 add(yes);

 add(no);

 add(maybe);

}

// Recognize buttons by their labels.

public boolean action(Event evtObj, Object arg) {

 if (evtObj.target instanceof Button) {

 if (arg.equals("Yes"))

 msg = "You pressed Yes.";

 else if (arg.equals("No"))

 msg="You pressed No.";

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

 msg="You pressed Undecided.";

 repaint( );

 return true;

 }

 return false;

}

public void paint (Graphics g) {

 g.drawString(msg, 6, 100);

}

}Sometimes it is useful to know the type of an object during run time. Inside **action( )**, notice that the outer **if** statement checks if the object that generated the event is a button by using the **instanceof** operator. **instanceof** has the following form:

 *object* instanceof *type*

Here, *object* is an instance of a class, and *type* is a class type. If *object* is of the specified type or can be cast into the specified type, then the **instanceof** operator evaluates to **true**.

Since **action( )** will receive all events generated by all controls that have been added to the current window, it is a good idea to categorize your responses by control types (making your code more efficient, and easier to maintain). Once it has been determined that a button was pressed, the next **if / else** statements determine which button it was by comparing the string contained in **arg** to the strings used to label the buttons.

***Swing Implementationof Buttons***

With swing, we just **JButton**s, instead of **Button**s. **JButton**s, **JTextField**s, and the like, are all **JComponent**s.

In swing, **JButton**s are created in a **JFrame** that is constructed by **main**. After the **JButton**s are constructed and added to a **Container**, they are registered. Registering a **JComponent** is easy, and is done as follows:

 JComponentVariableName.addActionListener( handler );

When an event occurs on the **JComponent**, the handler class listed above will be called. It is the responsibility of this handler class to identify and deal with the event.

Every **JComponent** supports several different types of event handlers, including mouse events, key events, and others. When an event occurs, the event is dispatched only to the event listeners of the appropriate type.

Each event type has a corresponding event-listener interface. For example, ActionEvents are handled by ActionListeners, MouseEvents are handled by MouseListeners, and KeyEvents are handled by KeyListeners (to name a few).

In this example, **JButton**s are used. **JButton**s generate **ActionEvent**s, which can be processed by any **ActionListener** object. When an event is generated by a user interaction with a registered component, the correct listener for that component is called. In the case of an **ActionEvent**, the event is dispatched to every registered **ActionListener**’s **actionPerformed** method (the only method defined in the **ActionListener** interface class).

In the example below, you will see a class implementation called ButtonHandler, which implements the **ActionListener** inferface. The ButtonHandler object is built, and assigned to a variable named **handler**. The JButton components are then registered with **handler**. Events occurring on these registered JComponents will then automatically be passed to **handler**’s **actionPerformed** method.

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

public class ButtonDemo extends JFrame {

 private JButton yes, no, maybe;

 public ButtonDemo ( )

 {

 super( "Testing Buttons" );

 // get content pane and set its layout

 Container container = getContentPane();

 container.setLayout( new FlowLayout() );

 // create and add the buttons

 yes = new JButton( "Yes" );

 container.add(yes);

 no = new JButton( "No" );

 container.add(no);

 maybe = new JButton( "Maybe" );

 container.add(maybe);

 // create an instance of inner class ButtonHandler

 // to use for button event handling

 ButtonHandler handler = new ButtonHandler( );

 yes.addActionListener( handler );

 no.addActionListener( handler );

 maybe.addActionListener( handler );

 setSize( 275, 100 );

 setVisible( true );

 }

 // execute application

 public static void main( String args[] )

 {

 ButtonDemo application = new ButtonDemo( );

 application.setDefaultCloseOperation(

 JFrame.EXIT\_ON\_CLOSE );

 }

 // inner class for button event handling

 private class ButtonHandler implements ActionListener {

 // handle button event

 public void actionPerformed( ActionEvent event )

 {

 JOptionPane.showMessageDialog( null,

 "You pressed: " + event.getActionCommand( ) );

 }

 } // end private inner class ButtonHandler

} // end class ButtonTest

***The textbook does a fine job of providing swing based examples. To supplement and balance your reading, I will provide “heavyweight” non-swing based examples in the remainder of my notes. For homework, both approaches are acceptable.****Repainting*

As a general rule, an applet writes to its window only when its **update( )** or **paint( )** methods are called by the AWT. This raises an interesting question: How can the applet, itself, cause its window to be updated when its information changes? For example, if an applet is displaying a moving banner, what mechanism does the applet use to update the window each time this banner scrolls? The answer - **use repaint( )**.

Note: One of the fundamental architectural constraints imposed on an applet is that it must quickly return control to the AWT run-time system. It cannot create a loop inside **paint( )** that repeatedly scrolls the banner, for example. This would prevent control from passing back to the AWT. Given this constraint, it may seem that output to your applet's window will be difficult, at best. Fortunately, this is not the case. Whenever your applet needs to update the information displayed in its window, it simply calls **repaint( )**.

The repaint( ) method has four forms:

 void repaint( )

 void repaint(int *x*, int *y*, int *width*, int *height*)

 void repaint(long *maxDelay*)

 void repaint(long *maxDelay*, int *x*, int *y*, int *width*, int *height*)

*maxDelay* specifies the maximum number of milliseconds that can elapse before **update** is called.

*Applying Check Boxes*

A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group.



Check boxes are objects of the **Checkbox** class.

**Checkbox** supports these constructors:

 Checkbox( )

 Checkbox(String *str*)

With the above two constructors, the state of the check box is unchecked. The 3rd form (below) creates a check box whose label is specified by *str* and whose group is specified by *cdGroup* If this check box is not part of a group, then *cbGroup* (*cbGroup* to be discussed momentarily) must be **null**. The value of *on* determines the initial state of the check box.

 Checkbox(String *str*, CheckboxGroup *cbGroup*, boolean *on*)

To retrieve the current state of a check box, call **getState( )**. To set its state, call **setState( )**. You can obtain the current label associated with a check box by calling **setLabel( )**. To set the label, call **setLabel( )**. These methods are as follows:

 boolean getState( )

 void setState(boolean *on*)

 String getLabel( )

 void setLabel(String *str*)

Each time a check box is pressed, **action( )** is called. The target field of its **Event** parameter contains a reference to the check box that generated the action. Its **Object** parameter contains the state of the check box. It will be **true** if the box is checked and **false** if its is cleared. Typically, check boxes are not used to cause immediate actions. (Push buttons are normally used for this purpose.) Usually an application will simply obtain the state of a check box when it needs to know the state of that option.

Here is an example that creates four check boxes. The initial state of the 1st box is checked. The status of each check box is displayed. Each time you change the state of a check box, the status display is updated.

import java.awt.\*;

import java.applet.\*;

/\* <applet code = "CheckboxDemo" width=250 height=200>

 </applet>

\*/

public class CheckboxDemo extends Applet {

 String msg = "";

 Checkbox winXP, win2000, solaris, mac;

 public void init( ) {

 winXP = new Checkbox("Windows XP", null, true);

 win2000 = new Checkbox("Windows 2000");

 solaris = new Checkbox("Solaris");

 mac = new Checkbox("MacOS");

 add(winXP);

 add(win2000);

 add(solaris);

 add(mac);

 }

 public boolean action(Event evtObj, Object arg) {

 if (evtObj.target instanceof Checkbox) {

 repaint( );

 return true;

 }

 return false;

 }

 public void paint(Graphics g) {

 msg = "Current state: ";

 g.drawString(msg, 6, 80);

 msg = " Windows XP: " + winXP.getState( );

 g.drawString(msg, 6, 100);

 msg = " Windows 2000: " + win2000.getState( );

 g.drawString(msg, 6, 120);

 msg = " Solaris: " + solaris.getState( );

 g.drawString(msg, 6, 140);

 msg = " MacOS: " + mac.getState( );

 g.drawString(msg, 6, 160);

 }

}

*CheckboxGroup*

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called *radio buttons*, because they act like the station selector on a car radio - only one station can be selected at any one time.



To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes. Check box groups are objects of type **CheckboxGroup**.

You can determine which check box in a group is currently selected by calling **getCurrent( )**. You can set a check box by calling **setCurrent( )**. These methods are defined as follows:

 Checkbox getCurrent( )

 void setCurrent(Checkbox *which*)

Here is the preceding example rewritten so that the check boxes are part of a group:

import java.awt.\*;

import java.applet.\*;

/\* <applet code = "CBGroup" width=250 height=200>

 </applet>

\*/

public class CBGroup extends Applet {

 String msg = "";

 Checkbox winXP, win2000, solaris, mac;

 CheckboxGroup cbg;

 public void init( ) {

 cbg = new CheckboxGroup( );

 winXP = new Checkbox("Windows XP", cbg, true);

 win2000 = new Checkbox("Windows 2000", cbg, false);

 solaris = new Checkbox("Solaris", cbg, false);

 mac = new Checkbox("MacOS", cbg, false);

 add(winXP);

 add(win2000);

 add(solaris);

 add(mac);

 }

 public boolean action(Event evtObj, Object arg) {

 if (evtObj.target instanceof Checkbox) {

 repaint( );

 return true;

 }

 return false;

 }

 public void paint(Graphics g) {

 msg = "Current state: ";

 msg += cbg.getCurrent( ).getLabel( );

 g.drawString(msg, 6, 100);

 }

}*Choice Controls*

* Used to create a pop-up list of items from which the user may choose
* When inactive, takes up only enough space to show the currently selected item
* When clicked, whole list of choices pop-up
* Each list item is a left-justified label

 

 

To add item to list use:

void addItem(String *name*)

Items are added to the list in the order addItem is called

To determine which item is currently selected use:

 String getSelectedItem( )

 int getSelectedIndex( )

To obtain the number of items in the list use:

 int countItems( )

Given an index, you can obtain the name associated with the item at that index location using:

 String getItem(int *index*)

**How it works:** Each time a **Choice** item is selected, **action( )** is called. The **target** field of its **Event** parameter contains a reference to the menu that generated the action. Its **Object** parameter contains the name of the newly selected item.

Here is an example that creates two Choice menus. One selects the operating system, the other selects the browser:

import java.awt.\*;

import java.applet.\*;

public class ChoiceDemo extends Applet {

 Choice os, browser;

 String msg = "";

 public void init ( ) {

 os = new Choice( );

 browser = new Choice( );

 // add items to os list

 os.addItem("Windows XP");

 os.addItem("Windows 2000");

 os.addItem("Solaris");

 os.addItem("MacOS");

 // add items to browser list

 browser.addItem("Netscape 3.1");

 browser.addItem("Netscape 4.x");

 browser.addItem("Internet Explorer 6.0");

 browser.addItem("Internet Explorer 7.0");

 browser.addItem("Lynx 2.4");

 browser.select("Netscape 4.x");

 // add choice lists to window

 add(os);

 add(browser);

 }

 public boolean action(Event evtObj, Object arg) {

 if (evtObj.target instanceof Choice) {

 repaint( );

 return true;

 }

 return false;

 }

 public void paint(Graphics g) {

 msg = "Current OS: ";

 msg += os.getSelectedItem( );

 g.drawString(msg, 6, 120);

 msg = "Current Browser: ";

 msg += browser.getSelectedItem( );

 g.drawString(msg, 6, 140);

 }

}

*TextFields*

* implements a single-line text-entry area
* allows the user to enter strings edit the text using arrow, cut & paste keys
* **TextField** is a subclass of **TextComponent**



**TextField** defines the following constructors:

 TextField( )

 TextField(int *numChars*)

 TextField(String *str*)

 TextField(String *str*, int *numChars*)

 String getText( )

 void setText(String str)

 String getSectedText( )

 setEditable(boolean *canEdit*)

 setEchoCharacter(char *ch*)

Since text fields perform their own editing functions, your program will not respond to individual key events. You program will respond to the user pressing ENTER. When this occurs, **action( )** is called. The **target** field of its **Event** parameter contains a reference to the text field in which ENTER was pressed. Its **Object** parameter contains a reference to the string that is contained in the text field.

Here is an example that creates the classic user name and password screen:

// Demonstrate text field

import java.awt.\*;

import java.applet.\*;

/\*

 <applet code="textFieldDemo" width=380 height=150>

 </applet>

\*/

public class TextFieldDemo extends Applet {

 TextField name, pass;

 public void init( ) {

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

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

 name = new TextField(12);

 pass = new TextField(8);

 pass.setEchoCharacter('?');

 add(namep);

 add(name);

 add(passp);

 add(pass);

 }

 // 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, 60);

 g.drawString("Selected text in name: " + name.getSelectedText( ), 6, 80);

 g.drawString("Password: " + pass.getText( ), 6, 100);

 }

}

*Understanding Layouts*

All of the components introduced above have been positioned by the default layout manager. A layout manager automatically arranges your controls within a window. While it is possible to lay out Java controls by hand, you generally won't want to, for two main reasons:

* It is very tedious
* Sometimes, width & height info is not yet available when you need to arrange components (because the native tookit components haven't been realized)

*This is a chicken-and-egg situation; it is pretty confusing to figure out when it is okay to use the size of a given component to position it relative to another.*

Each **Container** object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout( )** method. Whenever a container is resized, the layout manager is used to position each of the components within it. The **setLayout( )** method has the following form:

 void setLayout(LayoutManager *layoutObj*)

To position components manually, pass **null** for *layoutObj*.

* Each layout manager keeps track of a list of components by name
* Layout manager is notified each time you add a component to a containter
* Resizing containers activates the layout manager
* Each component managed contains the **preferredSize( )** and **minimumSize( )**
* **LayoutManager** contains several predefined classes described next.

*FlowLayout*

The default layout manager used in previous examples

Simple layout style

Similar to how words flow in a text editor

Components are laid out from the upper-left to lower-right

Here are the constructors:

 FlowLayout( )

 FlowLayout(int how)

 FlowLayout(int how, int horz, int vert)

// Use left-aligned flow layout

import java.awt.\*;

import java.applet.\*;

/\* <applet code = "FlowLayoutDemo" width=250 height=200>

 </applet>

\*/

public class FlowLayoutDemo extends Applet {

 String msg = "";

 Checkbox winXP, win2000, solaris, mac;

 public void init( ) {

 // set left-aligned flow layout

 setLayout(new FlowLayout(FlowLayout.LEFT));

 winXP = new Checkbox("Windows XP", null, true);

 win2000 = new Checkbox("Windows 2000");

 solaris = new Checkbox("Solaris");

 mac = new Checkbox("MacOS");

 add(winXP);

 add(win2000);

 add(solaris);

 add(mac);

 }

 public boolean action(Event evtObj, Object arg) {

 if (evtObj.target instanceof Checkbox) {

 repaint( );

 return true;

 }

 return false;

 }

 public void paint(Graphics g) {

 msg = "Current state: ";

 g.drawString(msg, 6, 80);

 msg = " Windows XP: " + winXP.getState( );

 g.drawString(msg, 6, 100);

 msg = " Windows 2000: " + win2000.getState( );

 g.drawString(msg, 6, 120);

 msg = " Solaris: " + solaris.getState( );

 g.drawString(msg, 6, 140);

 msg = " MacOS: " + mac.getState( );

 g.drawString(msg, 6, 160);

 }

}

  

*BorderLayout*

* Has 4 narrow, fixed-width components at the edges, and one large area in the center
* Regions are referred to by names "North", "South", "East", and "West"
* "Center " is middle area



BorderLayout( )

 BorderLayout((int *how*, int *horz*, int *vert*))

When adding components, you use the names of the regions using the following form of **add( )** :

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

// Demonstrate BorderLayout

import java.awt.\*;

import java.applet.\*;

import java.util.\*;

/\*

 <applet code="BorderLayoutDemo" width=400 height=200>

 </applet>

\*/

public class BorderLayoutDemo extends Applet {

 public void init( ) {

 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));

 }

}

*GridLayout*

* GridLayout lays out components in a two-dimensional grid
* When you instantiated, you define the number of rows and columns

Here are the constructors:

 GridLayout(int *numRows*, int *numColumns*)

 GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*)



// Demonstrate GridLayout

import java.awt.\*;

import java.applet.\*;

/\*

 <applet code= "GridLayoutDemo" width=300 height=200>

 </applet>

\*/

public class GridLayoutDemo extends Applet {

 static final int n = 4;

 public void init( ) {

 setLayout(new GridLayout(n, n));

 setFont(new Font("Helvetica", Font.BOLD, 24));

 for (int i = 0; i < n; i++) {

 for (int j = 0; j < n; j++) {

 int k = i \* n + j;

 if (k > 0)

 add(new Button("" + k));

 }

 }

 }

}

*Handling Events*

* Applets are event-driven programs
* Most events are generated by the user using the mouse

*The Event Class*

Defines several variables that describe the event

Location of the mouse when a mouse event occurs is stored in variables **x** and **y**

When a key is pressed, the key is stored in the **key** variable

AWT automatically routes mouse and keyboard events to a set of predefined methods that your applet will override

*Mouse Events*

The following methods can be overridden by your applet. When it does so, the method must return **true** if it handles the event, and **false** if it does not. This causes the event to be passed on to the event handler of a parent window, should one exist.

The two most important methods are **mouseDown( )** and **mouseUp( )** .

* **mouseDown( )** is called whenever any mouse button is pressed.
* **mouseUp( )** is called whenever any mouse button is released.
* Java does not distinguish between mouse buttons.

(Not all systems have the same number of mouse buttons…

Java is designed for the lowest common denominator: a one-button mouse)

|  |  |
| --- | --- |
| boolean mouseDown (Event *evtObj*, int *x*, int *y*) | Called when a mouse button is pressed. The event object that describes the event is passed in *evtObj*. The coordinates of the mouse pointer at the time the event was generated are passed in *x* and *y*. This method must return **true** if it handles the event. |
| boolean mouseDrag (Event *evtObj*, int *x*, int *y*) | Called when the mouse is moved when a button is pressed. The event object that describes the event is passed in *evtObj*. The coordinates ofthe mouse pointer at the time the event was generated are passed in *x* and *y*. This method must return **true** if it handles the event. Mouse drag events continue to occur as long as the mouse is being moved within the window and a button is pressed. |
| boolean mouseEnter (Event *evtObj*, int *x*, int *y*) | Called when the mouse moves into the window. The event object that describes the event is passed in *evtObj*. The coordinates of the mouse pointer at the time the event was generated are passed in *x* and *y*. This method must return **true** if it handles the event. |
| boolean mouseExit (Event *evtObj*, int *x*, int *y*) | Called when the mouse moves out of the window. The event object that describes the event is passed in *evtObj*. The coordinates of the mouse pointer at the time the event was generated are passed in *x* and *y*. This method must return **true** if it handles the event. |
| boolean mouseMove (Event *evtObj*, int *x*, int *y*) | Called when the mouse is moved. The event object that describes the event is passed in *evtObj*. The coordinates of the mouse pointer at the time the event was generated are passed in *x* and *y*. This method must return **true** if it handles the event. Mouse move events continue to occur as long as the mouse is being moved within the window and no button is pressed. |
| boolean mouseUp (Event *evtObj*, int *x*, int *y*) | Called when the mouse button is released. The event object that describes the event is passed in *evtObj*. The coordinates of the mouse pointer at the time the event was generated are passed in *x* and *y*. This method must return **true** if it handles the event. |

The following simple program echoes keystrokes to the applet's window:

import java.awt.\*;

import java.applet.\*;

public class MouseEvents extends Applet {

 String msg = "";

 int mouseX = 0, mouseY = 0; // coordinates of mouse

 public boolean mouseDown(Event evtObj, int x, int y) { // Handle button press

 mouseX = x;

 mouseY = y;

 msg = "Down";

 repaint( );

 return true;

 }

 public boolean mouseUp (Event evtObj, int x, int y) { // Handle button release

 mouseX = x;

 mouseY = y;

 msg = "Up";

 repaint( );

 return true;

 }

 public boolean mouseMove (Event evtObj, int x, int y) { // Handle mouse move

 showStatus("Moving mouse at " + x + ", " + y);

 return true;

 }

 public boolean mouseDrag (Event evtObj, int x, int y) { // Handle button drag

 mouseX = x;

 mouseY = y;

 msg = "\*";

 showStatus("Dragging mouse at " + x + ", " + y);

 repaint( );

 return true;

 }

 public boolean mouseEnter (Event evtObj, int x, int y) { // Handle button enter

 // save coordinates

 mouseX = 0;

 mouseY = 10;

 msg = "Mouse just entered.";

 repaint( );

 return true;

 }

 public boolean mouseExit (Event evtObj, int x, int y) { // Handle button exit

 mouseX = 0;

 mouseY = 10;

 msg = "Mouse just left.";

 repaint( );

 return true;

 }

public void paint (Graphics g) {// Display msg in applet window at current X, Y location

 g.drawString(msg, mouseX, mouseY);

}

}



*A Simple Painter Program*

import java.applet.Applet;

import java.awt.\*;

import java.awt.event.\*;

public class Painter

 extends Applet {

 private int xValue = -10, yValue = -10;

 public void paint(Graphics g) {

 g.drawString("Drag the mouse to draw", 10, 20);

 g.fillOval(xValue, yValue, 4, 4);

 }

//Override Component class update method to allow all ovals // to remain on the screen by not clearing the background

 public void update(Graphics g) {

 paint(g);

 }

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

 xValue = x;

 yValue = y;

 repaint();

 return true;

 }

}



**Assignment for Chapter 14**

Modify the program above to incorporate colors. In a separate window, provide a “toobar” of *RadioButton* objects that lists the following six colors: red, black, magenta, blue, green and yellow. The toolbar should be implemented as a subclass of *Frame* called *ToolBarWindow* and should consist of six buttons, each with the appropriate color name. When a new color is selected, drawing should occur in the new color.

*ToolBarWindow* must have a method that returns a *Color*, titled *getCurrentColor*. Determine the currently selected color by calling the public method *getCurrentColor* on the *ToolBarWindow*.

Extra Credit: Add a “*Clear*” button (see below) that clears the image. *Pressing the clear button in the frame must cause the drawing to immediately clear itself.* This will require interprocess communication between the *Frame* and your *main applet* (i.e., your Frame must be given the address of your main applet so that it can communicate to your main applet to clear itself).

