Java User Interface

3.1 Overview: The AWT Package
Introduction

• User Interface (UI) refers to all sorts of communication between a program and its users
• Include not only what user sees, but what user hears and feels
• Among all UIs, the Graphical User Interface (GUI) is the most effective and easily implemented user interface
• The Java package called the Abstract Windows Toolkit (AWT) contains a complete set of classes for writing GUI programs
The AWT Component Class

java.awt.component

- Container
  - Scrollbar
  - ScrollPane
  - Dialog
- Button
- Choice
- Canvas
  - List
- TextComponent
  - Label
  - TextField
  - TextArea
- Window
- Panel
- Frame
- Applet
AWT Components

• Except menu, every GUI component is implemented with a subclass of the AWT component class.

• These components include Button, Checkbox, Choice, List, and TextField, etc.

• Other ways of getting user input include: Sliders, Scrollbars, Text Areas, and Labels are also included in the AWT component class.
Click here to see all AWT Components
AWT Containers

• GUI components must be grouped together using a container
• The AWT `component` class provides three types of containers: ScrollPane, Panel, and Window
• `ScrollPane`: to display a potentially large component in a limited amount of space
A. AWT Panel

- *Panel* is the superclass of the *Applet* class
- It is the default container for applets
- Panels do not create their own windows
- Panels can be nested into another container, e.g. panel
- Useful for grouping related visual objects
B. AWT Window

- The window subclass contains **Frame** and **Dialog** containers
- **Frames** create normal, full-fledged windows
- Frames can exist **outside a browser** and are **typically used for application windows**
- **Menu bar** can be added to frames but not to other containers
- A Frame is the **default container for standalone applications**

The AWT components window is a frame
C. AWT Dialog

- **Dialog** is dependent on another window (e.g. a Frame)
- When the window is destroyed, so are its dependent dialogs
- Dialogs are like Frames but they can be modal
- Modal means that the dialog can be made to halt input to any other container until some action is performed in the dialog
- Its subclass **FileDialog** is often used for opening or saving files
See a Dialog brought up by a Frame
Other AWT Classes

- The **AWT** contains much more than components
- Contain a variety of classes related to drawing, event handling, graphics, image, etc.
  - `java.awt.event`: event handling (e.g. mouse events)
  - `java.awt.Graphics`: drawing graphics
  - `java.awt.Font & java.awt.FontMetrics`: text fonts
  - `java.awt.Image`: represent image data
  - `java.awt.Color`: handle the color of objects
  - `java.awt.Dimension`: sizes and shapes of objects
AWT Layout Managers

• The business of positioning and shaping the components in a container falls to a layout manager
• A number of classes implement Layout Manager
  – java.awt.BorderLayout
  – java.awt.CardLayout
  – java.awt.FlowLayout
  – java.awt.GridBagConstraints
  – java.awt.GridBagLayout
  – java.awt.GridLayout
GridBag Layout

- The outermost layout manager is always `GridBagLayout`
- Arrange components in **horizontal and vertical** dimensions
- Sizes and relative placement of components determined by the constraints defined in the `GridBagConstraints` class
- Other Layout Manager can be placed in the cells of `GridBagLayout`
- One can nest `GridBagLayout` with another `GridBagLayout`
Java User Interface

3.2 Visual GUI Builder
Visual GUI Builder

- Programming directly using AWT package can be complicated and difficult
- Java Workshop 2.0 provides a simple interface for GUI programming - Visual GUI Builder
- The Visual GUI Builder serves as a graphical interface between programmers and the AWT packages
- Users can place their components in a WYSIWYG manner
- The system automatically generates the code required
- However, users still need to provide code to response to various inputs from the GUIs
Building Your First GUI

• You are going to build a GUI which has a List component placed inside a panel

• The output will be shown in a form of applet
Step 1: Create the layout window

- Open Project Manager and make Personal the current portfolio
- Choose File - New - Project
- In the Create Project wizard, make these entries and choices:
  - Enter the name **FirstGUI** for the project
  - Click applet for the project type
  - Click “The GUI will be created using the Visual Java GUI Builder.”

  ![Diagram of a layout window](image)

- Click the **Next** button. In this second wizard dialog, specify a directory for the project’s source files and choose **No** to existing Java source files
- Click **Next** and choose the generated HTML page
- Click **Finish**
• A **Panel** is created for you to place the components
• Panel is the **default container** for **applet**
• By default, the **GridBagLayout** manager is automatically invoked for you to **place the components in a grid form**
Step 2: Add the component

- Make only one cell by deleting the other cells (click on the cells and press delete)
- From the **GUI component palette**, choose the list component and click on it
- Move the mouse pointer to the cell in the layout window. Click again
Step 3: Set the attributes and preview result

- Open the Attribute Editor
- Enter this list of items in the items text field, separate each item with a comma (no space): Red, Yellow, Blue. Click Apply. See the result.
- See what happen if you add two more items, say orange and purple.
- Change to preview mode, see the result.
- Save the GUI by selecting GUI - Save & Generate GUI file.
Generate Source Code

• It is seen that the building of GUI is simply a **click and pick process**. No coding is needed
• However, the GUI built is not useful at all since it can do nothing except scrolling among different items
• To let the program actually do something useful, **users should add code to response to the choice from the GUI**
Step 4: Modify the GUI

- Modify the GUI to add a **multi-line text field** to the right of the **list** (Use the **arrow** keys to add more rows and columns)
- Open the **Attribute Editor** for the **multi-line text field** to modify the **name** field to **textOut**. Click **Apply**
- Open the **Attribute Editor** for the **list** to modify the **name** field to **itemList**. Click **Apply**
Step 5: Add user code

- Click **Edit Operation** in the **Attribute Editor** for the **list**. Click **Insert** and then **Action**
- Select **Action Type to Execute Code**
- Type the following code and then click **OK**. Save the GUI when finish

```java
import java.awt.List;
import java.awt.TextArea;
String newline = System.getProperty("line.separator");
List listBody = (List) gui.itemList.getBody();
TextArea textBody = (TextArea) gui.textOut.getBody();
textBody.setEditable(false);
String colorItem = listBody.getSelectedItem();
textBody.append("Select the " + colorItem + " item." + newline);
```
Step 6: Build and Run

- When the GUI is saved, the applet can be built using the Java Workshop 2.0
- If success, run the applet

Now a text line will be displayed if a particular entry is double-clicked
**Exercise:** From the distributed work sheet, build the *Rolo* project
Java User Interface

3.3 Working Principles
Files Created by the GUI Builder

- **Visual GUI Builder** generates a `.gui` file as well as four `.java` source files.
- Three of these files (*Root*, *Main*, and *Ops*) support the running of program and should not be modified manually.
- The last file (*projectname.java*) is where the source code is added for the task that a program must accomplish.
• `projectname.gui`:
  Describe the GUI for the project; update whenever the layout is saved.
  One .gui file for each group.

• `projectnameRoot.java`:
  Initialize the components for the project.

• `projectnameOps.java`:
  Contain the operations for the project.
  Need if operation is defined for component.

• `projectnameMain.java`:
  This is where the program start.
  Contain the main() method in case of standalone application.

• `projectname.java`:
  Contains a template for the project; generate only once.
  Modify if event handler, action handler and other functionality are to be implemented.
To define the operations in the lifetime of the applet

To define the operations in the lifetime of the group

To define the attributes of the components

To define the operations associated with the components
The Concept of Groups and Shadows

- **Visual GUI Builder** simplifies the programming job by introducing an interface - formed by **groups and shadows** - between users and the AWT.
- A **shadow** is a wrapper class for an AWT component. A **group** is a collection of shadows and other groups.
Shadows

- Shadows are **wrapper classes** for AWT components
- Responsible for **interfacing** the AWT components to groups and hence applications
- **Transfer the AWT events** to the groups
- **Custom functionality may also be included** in the shadow classes. For example, the window shadow resizes the window to its preferred size
Groups

- A group **encapsulates** a set of shadows and subgroups
- Groups inside other group are **subgroups**
- The top-level is **base group**
- **When a GUI is built** in the GUI Builder, **a new group is defined** out of existing shadows and groups
- An **application** may contain a number of GUIs, each GUI **can be a subgroup** contains a number **shadow components**
- **Messages** sent by subgroup propagate up until they are handled or until they reach the base group
- From external viewpoint, a group is a single component that has a set of **attributes, public methods and messages**
A. Group Messages

- A group will create messages and post them to their parent groups
- A group need not have any knowledge of its parent
- The parent can listen for a particular message from its subgroup and perform the corresponding action
- A message can be sent by subgroup, or can be sent as a result of AWT event delivery
B. Message Handling

• The **operation** required to handle message is defined in the group

• When an **AWT event** is generated by an AWT component, it is **translated into a message** and then sent to the group that contains that shadow component

• To **receive the message**, the group concerned should **override** one of the following methods:
  
  • `handleMessage`, `handleEvent`, `action`, `mouseDown`, `mouseDrag`, `mouseUp`, `mouseMove`, `mouseEnter`, `mouseExit`, `keyDown`, `keyUp`, `gotFocus`, `lostFocus`
• For the **group** that wants to receive the message or the AWT event, one of the above methods should be **overridden**
• E.g., if the group wants to response to the **mouse down** event, the method **mouseDown()** should be overridden
C. `handleMessage()`

- `handleMessage()` is responsible for handling messages, whether it is a message from subgroup or an AWT event.
- If a group wants to response to a message, the `handleMessage()` method should be overridden.
- Otherwise, it can either ignore that method or implement it by calling the default method `super.handleMessage()`.
- By default, message will not propagate up in the tree.
- If a group wants to let the parent to handle the message, `handleMessage()` should be overridden by returning `false`.

```java
// Excerpt from FirstGUI.java
public boolean handleMessage(Message msg) {
    return super.handleMessage(msg);
}
```
D. `handleEvent()`

- If the group wants to respond specific to the AWT events, the `handleEvent()` method should be overridden.
- Otherwise, it can either ignore that method or implement it by calling the default method `super.handleEvent()`.
- By default, event will not propagate up in the tree.
- If a group wants to let the parent to handle the event, `handleEvent()` should be overridden by directly calling the method `postMessageToParent()`.

```java
// Excerpt from FirstGUI.java
public boolean handleEvent(Message msg, Event evt) {
    return super.handleEvent(msg, evt);
}
```
E. Individual Event

- If the group wants to response specific to an AWT event, the corresponding method should be overridden.
- Otherwise, it can simply ignore them.
- E.g., in FirstGUI, if the group wants to make some sound every time a selection is made, the method action should be overridden.

```java
import java.awt.Toolkit;
public boolean action(Message msg, Event evt, Object what) {
    Toolkit.getDefaultToolkit().beep();
    return true;
}
```
F. Overview of Events

- **Action events** - *Buttons* and other *simple components* generate these to indicate that user wants an action to occur.
- **Adjustment events** - Generated by *scrollbar* when their value change.
- **Component events** - Generated by any *component* that its size, position or visibility has been changed.
- **Container events** - Generated by any *container* that a component has been added to or removed from it.
- **Focus events** - Generated by any *component* that it gains or loses the ability of receiving keyboard input.
- **Item events** - Generated by *checkboxes*, *pop-up lists of choices*, and scrolling *lists* to indicate user has made a choice.
• **Key events** - Generated by any component with the current keyboard focus. Notifies the listener of key press

• **Mouse events** - Generated by any component to notify mouse click or user moving the cursor into or out of the drawing area

• **Mouse-motion events** - Generated by any component on the change of cursor’s position over the component

• **Text events** - Generated by text components when text value changes

• **Window events** - Generated by windows when they are opened, closed, activated, deactivated, etc.
Exercise 1: Add the code on p.38 to FirstGUI, see and hear the result

Exercise 2: Modify the `handleMessage()` method of FirstGUI as follows. See and hear the difference from Exercise 1

```java
import java.awt.Toolkit;
public boolean handleMessage(Message msg, Event evt) {
    Toolkit.getDefaultToolkit().beep();
    return true;
}
```
Using Operation Editor of GUI Builder

- **GUI Builder** further simplifies the operation setting by introducing the **Operation Editor**
- Invoke by clicking the “**Edit operations ...**” button of **Attribute Editor**
- Facilitate the setting of the responses of users **associated directly** to the component concerned

- Contain two parts - **filter** and **action**
  - **Filter** - narrow the selection of event
  - **Action** - define the operation (e.g. adding code)
A. Filter

Select filter type: event or message

Select event type for the item List (different component can have different type)

If it is key event, select the key that triggers the event

If it is key event, select the modifier for the key

If it is mouse event, select the click count
B. Action

```java
import java.awt.List;
import java.awt.TextArea;

String newline = System.getProperty("line.separator");
List listBody = (List) gui.itemList.getBody();
TextArea textBody = (TextArea) gui.textOut.getBody();
String colorItem = listBody.getSelectedItem();
textBody.append("Select the " + colorItem + " item." + newline);
```
import java.awt.List;
import java.awt.TextArea;

// Define a string `newline` which represents a line feed
String newline = System.getProperty("line.separator");

// `gui` is the default variable for the root class
// We can use `getBody()` to retrieve the component from the shadow
List listBody = (List) gui.itemList.getBody();
TextArea textBody = (TextArea) gui.textOut.getBody();

// We can use `getSelectedItem()` to get our selection to the component
// It is the standard API for the `List` class
String colorItem = listBody.getSelectedItem();

// append is the standard API for the `TextArea` class
// It adds an item to the end of the text area
textBody.append("Select the " + colorItem + " item." + newline);
Java User Interface

3.4 Working with Sound, Graphics and Images
Playing Sound

• Currently, Java supports only one sound format: 8-bit, μ law, 8000 Hz sampling, one-channel, Sun “.au” file
• Sound file not in this format has to be converted using a sound conversion tool
• Some sound-related Applet APIs:
  • `AudioClip getAudioClip(URL)`: get a audio clip from `URL`
  • `AudioClip getAudioClip(URL, String)`: get a audio clip from `URL` + the relative location given by `String`
  • `void play(URL)`: play the audio clip from `URL`
  • `void play(URL, String)`: play the audio clip from `URL` + the relative location given by `String`
The AudioClip interface further defines the following methods:

- `void loop()`: Start playing the clip repeatedly
- `void play()`: Play the clip only once
- `void stop()`: Stop the clip. Working with both looping and one-time sounds

Example:
Add three buttons to the `FirstGUI` for generating
- a one-time train sound
- a looping train sound
- to stop a looping train sound
Step 1: Name and Add code to the buttons

• Add three buttons to the panel layout of FirstGUI (create three more cells by using the arrow keys)
• From the Attribute Editor, label the three buttons as stop, train loop and train
• Name the three buttons as stopButton, trainloopButton and trainButton
• From the Operation Editor, add the following codes to each of the button
  • For stop button: group.stopButtonCallback ( );
  • For trainloop button: group.trainloopButtonCallback ( );
  • For train button: group.trainButtonCallback ( );
• These codes let the corresponding callback method of the group be called when the button is pressed
Step 2: Add code to FirstGUI.java

```java
public void trainButtonCallback() {
    Applet ap = getApplet();
    AudioClip onceClip;
    onceClip = ap.getAudioClip(ap.getCodeBase(),"train.au");
    onceClip.play();
}
```

To find out the current applet

Call when the train button is pressed

Get the audio clip from the URL as specified by the passed parameters

Play the audio clip

To find out the URL where the current applet located

The exact name of the audio clip file
private boolean looping = false;
private AudioClip currentClip = null;

public void trainloopButtonCallback() { // call when the trainloop is pressed
    Applet ap = getApplet();
    AudioClip onceClip;
    onceClip = ap.getAudioClip(ap.getCodeBase(),"train.au");
    currentClip = onceClip;
    looping = true;
    onceClip.loop();
}

public void stopButtonCallback() { // call when the stop button is pressed
    if (looping) {
        currentClip.stop();
        currentClip = null;
        looping = false;
    }
}
Work with Graphics

- The Graphics class defines methods for drawing the following kinds of shapes:

  - Lines - `drawline`
  - Rectangles - `drawRect`, `fillRect`, and `clearRect`
  - Raised or lowered rectangles - `draw3DRect` and `fill3DRect`
  - Round-edged rectangles - `drawRoundRect` and `fillRoundRect`
  - Ovals - `drawOval` and `fillOval`
  - Arcs - `drawArc` and `fillArc`
  - Polygon - `drawPolygon`, `drawPolyline`, and `fillPolygon`
• The custom components, such as, Canvas, Panel, or Applet subclasses, are good for drawing graphics
• Standard AWT components, such as, buttons and text fields need no drawing - they paint automatically
• The major method for drawing is **repaint**, a method of the component class (the parent of all components)

```
repaint -> update -> paint
```

- Clear the background
- Do nothing

• If one wants to draw something useful, **paint** should be overridden
• If one does not want the background be cleared, **update** should be overridden
Four forms of `repaint` methods:

- `repaint ()`
  - request the AWT call `update` as soon as possible. Most frequently used form of `repaint`

- `repaint (long time)`
  - request the AWT call `update` as much as `time` milliseconds from now

- `repaint (int x, int y, int width, int height)`
  - request the AWT call `update` as soon as possible but paint only the specified part of the component

- `repaint (long time, int x, int y, int width, int height)`
  - request the AWT call `update` as much as `time` milliseconds from now, but repaint only the specified part of the component
Example:

- By using **VJCanvas** (a subclass of Canvas that is used in GUI Builder), draw a rectangle of size 10x10 at the point where the mouse is clicked on the applet.
- Use **label** to show the exact location where the mouse is clicked.
Step 1: Setting layout and Naming the components

- Add a project called *canvasGUI*. Choose to use GUI Builder
- From the *panel* created, delete until only two cells remain
- From the *component palette*, add the items *canvas* and *label* to the cells
- Name the components as *canvasArea* and *labelArea*, respectively
- Set the *background color* of canvas be *white*
- In the file *canvasGUI.java*, import two packages
  
  ```java
  java.awt.event.* and sunsoft.jws.visual.rt.awt.*
  ```
Step 2: Modify `handleEvent` in `canvasGUI.java`

```java
public boolean handleEvent(Message msg, Event evt) {
    if (msg.target == gui.canvasArea) {
        switch(evt.id) {
            case Event.MOUSE_DOWN:
                // Draw the rectangle
                //
        }
    }
    return super.handleEvent(msg, evt);
}
```

Modify the `handleEvent` to handle any event from `canvasArea`

Action is taken only when the event target is `canvasArea`

If mouse is clicked, do drawing at where the click is made

If we do not handle the event, let the super class handles it
Step 3: Add code for drawing rectangle

```java
// Use getBody to get the VJCanvas from the shadow class
VJCanvas canvasBody = (VJCanvas) gui.canvasArea.getBody();

// Get the graphic context of VJCanvas
Graphics g = canvasBody.getGraphics();

// Use getBody to get the Label from the shadow class
Label labelBody = (Label) gui.labelArea.getBody();

// Set the text depends on where the mouse is clicked
labelBody.setText("Click occurred at ("+evt.x+"), ("+evt.y+").");

// Clear the drawing area
canvasBody.update(g);

// Draw a rectangle of size 10x10 at where the mouse clicks
g.fillRect(evt.x-5, evt.y-5, 10, 10);
```
Using Image

- Support for using images is spread across the `java.applet`, `java.awt` and `java.awt.image` packages.
- The AWT makes it easy to load images in either of two formats: GIF and JPEG.
- To load an image, `getImage` is called.
- In fact, `getImage` returns immediately without actually loading the image.
- The actual loading doesn’t start until the first time the program tries to draw the image.
A. Using *getImage* ()

- The Applet class supplies two *getImage* methods:
  - `getImage(URL url)`
  - `getImage(URL url, String name)`
- Only **applets** can use the Applet *getImage* methods
- The Applet *getImage* methods don’t work until the applet has a **full context**
- For this reason, one should call *getImage* from a method such as *init* when the applet context has been formed
- Some examples of *getImage*:
  ```java
  Image image1 = getImage(getCodeBase(), "imageFile.gif");
  Image image2 = getImage(imageURL);
  ```
B. Using `drawImage()`

- Images are displayed when `drawImage` is called.
- In fact the image is loaded only when `drawImage` is called. Hence the first call to `drawImage` sometimes cannot display the image.

- `Graphics` class declares the following `drawImage` methods:
  - `drawImage (Image img, int x, int y, ImageObserver observer)`
  - `drawImage (Image img, int x, int y, int width, int height, ImageObserver observer)`
  - `drawImage (Image img, int x, int y, Color bgcolor, ImageObserver observer)`
  - `drawImage (Image img, int x, int y, int width, int height, Color bgcolor, ImageObserver observer)`
• All methods return \textit{true} if the image has been \textit{completely loaded and completely drawn}; otherwise, return \textit{false}

• An \textit{image observer} needs to be provided in these methods

• An image observer is an object that implements the \texttt{ImageObserver} interface

• It is \textit{notified} whenever new information about the image becomes available

• Most \texttt{components} implement the \texttt{ImageObserver} interface and hence can be an \textit{image observer}
Example:

- Draw the image `rocketShip.gif` by using `VJCanvas`.
- Add three buttons to display the original, the scale-up and cropped version of `rocketShip.gif`.
Step 1: Setting layout and Naming the components

- Add a project called `imageGUI`. Choose to use GUI Builder.
- From the panel created, delete until six cells remain.
- From the component palette, add the items canvas and buttons to the cells.
- Name the components as `canvasArea`, `smallButton`, `bigButton`, and `cropButton`, respectively.
- Label the text of the buttons to be Small, Big, and Crop.
- Add operation codes for the buttons to be
  
  ```java
  group.smallButtonCallback()  // for Small button
  group.bigButtonCallback()    // for Big button
  group.cropButtonCallback()   // for Crop button
  ```
Step 2: Add code for initialization

- Import the following class and packages
  
  \texttt{java.applet.Applet, sunsoft.jws.visual.rt.awt.*}
  
  \texttt{and java.awt.image.*}

- Add the following code for \texttt{initGroup()}

```java
Image image;  // the buffer to store the image
int size = 0;  // size=0,1,2 small, big, crop
protected void initGroup() {
    // Everything is done when the group is first created
    Applet ap = this.getApplet();  // get the current applet
    image = ap.getImage(ap.getCodeBase(), "rocketship.gif");  // get the image from where the applet located
}
```
Step 3: Add code to implement the button’s callback

```java
public void smallButtonCallback() {
    VJCanvas canvasBody = (VJCanvas) gui.canvasArea.getBody();
    Graphics g = canvasBody.getGraphics();
    size = 0; // small size
    canvasBody.repaint();
}

public void bigButtonCallback() {
    VJCanvas canvasBody = (VJCanvas) gui.canvasArea.getBody();
    Graphics g = canvasBody.getGraphics();
    size = 1; // big size
    canvasBody.repaint();
}
```

When `repaint` is called, `update` of `canvasBody` will be called to clear the VJCanvas. `Update` and `paint` events will be generated by the VJCanvas.
Step 4: Add code to response to the paint event

```java
public boolean handleEvent(Message msg, Event evt) {
    switch (evt.id) { // response if paint event received
        case VJCanvas.PAINT_EVENT:
            if (msg.target == gui.canvasArea) {
                // create VJCanvas object and graphics object
                VJCanvas canvasBody =
                    (VJCanvas) gui.canvasArea.getBody();
                Graphics g = canvasBody.getGraphics();
                if (size == 0) { // small size
                    g.drawImage(image, 0, 0, canvasBody);
                } else if (size == 1) { // big size
                    g.drawImage(image, 90, 0, 300, 62, canvasBody);
                } // the image is resized to 300x62
            }
            return super.handleEvent(msg, evt);
    } // Let the superclass handle the event if we don't
}
```
C. Behind the scene

• When the `getImage` is called, an **image producer** is created to produce the raw data for the **Image** object

• **Image producer** - an object that adheres to the **ImageProducer** Interface

• The image producer provides data to an **image consumer** - an object that adheres to the **ImageConsumer** interface
• The AWT handles the **production and consumption** of image data.
• **Users need not be aware** of the operation behind the scene normally.
• If any modifications, other than scaling (such as rotation), is needed to the image data, an **image filter** would be implemented.
• At that time, users need to **implement the image production and consumption themselves**.
To use an existing filter, the following codes are often used

```java
Image sourceImage, resultImage;
... // Initialize sourceImage by calling getImage ...

// to create an filter object from SomeImageFilter
ImageFilter filter = new SomeImageFilter();

// to get the image producer of the current image
ImageProducer currentProducer = sourceImage.getSource();

// to get the new image producer for the modified image
ImageProducer producer = new FilteredImageSource(
    currentProducer, filter);

// From the new image producer, create the image
Image resultImage = createImage (producer);
```

When `drawImage` is called, the image consumer inside the AWT will response to it and get data from `resultImage`
Currently, there are not so many image filters built inside the java.awt.image package.

One of them is CropImageFilter filter, which produces an image consisting of a rectangular region of a larger image.

More image filter can be found in http://java.sun.com/docs/books/tutorial/2e/book.html.
Example:

- To implement the **Crop** button of the previous example
- If the **Crop** button is clicked, only the **upper left 50x50 pixels** of an image is displayed
Step 1: Add code to response to the Crop button

```java
Image resultImage; // declare to be a member variable
public void cropButtonCallback() {
    resultImage = image; // get VJCanvas & graphics body
    VJCanvas canvasBody=(VJCanvas)gui.canvasArea.getBody();
    Graphics g = canvasBody.getGraphics();
    // use the CropImageFilter to crop only the upper
    // left 50x50 pixels of the source image
    ImageFilter filter = new CropImageFilter(0,0,50,50);
    ImageProducer pro = image.getSource();
    ImageProducer producer = new FilteredImageSource(pro, filter);
    // create the image and call repaint
    resultImage = canvasBody.createImage(producer);
    size = 2;
    canvasBody.repaint();
}
```
public boolean handleEvent(Message msg, Event evt) {
    switch (evt.id) {
        case VJCanvas.PAINT_EVENT:
            if (msg.target == gui.canvasArea) {
                if (size == 0) {
                
                } else if (size == 2) {
                    g.drawImage(resultImage, 0, 0, canvasBody);
                    // if crop is clicked, draw the modified image
                }
            }
            return super.handleEvent(msg, evt);
    }
}