# Getting Started with the PictureExplorer Class

*Learn about a picture explorer class that allows you to determine the numeric color values for any pixel in a picture by placing a cursor on the pixel. The pixel position is controlled by clicking or dragging the mouse within the picture, clicking buttons, or typing coordinate values. You can also zoom in and out to view the pixel in more or less detail, and you can see the actual color of the pixel in a large square.*

**Published:** March 19, 2009
**By Richard G. Baldwin**

Java Programming Notes # 362



---

# Preface

## General

This lesson is the next in a series *(see Resources)* designed to teach you how to write Java programs to do things like:

- Remove *redeye* from a photographic image.
- Distort the human voice.
- Display one image inside another image.
- Do edge detection, blurring, and other filtering operations on images.
- Insert animated cartoon characters into videos of live humans.

If you have ever wondered how to do these things, you've come to the right place.

## What you have learned from earlier lessons

If you have studied the earlier lessons in this series, you have learned about **Turtle** objects and their ability to move around in a world or a picture and to draw lines as they are moving. You have learned all about the **World** class, the **Picture** class and its superclass named **SimplePicture**.

By learning about the constructors and methods of the **SimplePicture** class, you have learned that objects of the **Picture** class are useful for much more than simply serving as living quarters for turtles. They are also useful for manipulating images in interesting and complex ways.

## What you will learn in this lesson

Near the end of the previous lesson, I told you that there remained only one significant method of the **SimplePicture** class that I had not yet explained: the **explore** method.

**Pixel Editor Program**
See the lesson titled *A Pixel Editor Program in Java: Multimedia Programming with Java* in Resources for a non-trivial application of a **PictureExplorer** object.

### The explore method and the PictureExplorer class

The **explore** method consists of a single statement that creates an object of the **PictureExplorer** class. The PictureExplorer class is a large and complex class. From an educational viewpoint, the **PictureExplorer** class is a very significant class because it provides an event-driven graphical user interface *(GUI)*, which is an extremely important Java programming topic. It also incorporates objects of anonymous inner classes as listener objects, which is also a very important Java topic.

I will begin my explanation of the **PictureExplorer** class in this lesson.

### Source code listings

A complete listing of Ericson's **PictureExplorer** class is provided in Listing 13 near the end of the lesson. A complete listing of a very simple program named Java362a that I will use to illustrate the behavior of the **PictureExplorer** class is provided in Listing 14.

### Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

#### Figures



#### Listings



### Supplementary material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com.

## General background information

### A multimedia class library

In this series of lessons, I will present and explain many of the classes in a multimedia class library that was developed and released under a **Creative Commons Attribution 3.0 United States License** *(see Resources)* by Mark Guzdial and Barbara Ericson at Georgia Institute of Technology. In doing this, I will also present some interesting sample programs that use the library.

### Software installation and testing

I explained how to download, install, and test the multimedia class library in an earlier lesson titled *Multimedia Programming with Java, Getting Started (see Resources)*.

## Preview

In this and the lessons that follow, you will learn about the **PictureExplorer** class, which allows you to determine the numeric color values for any pixel in a picture by placing a cursor on the pixel. The pixel position is controlled by clicking or dragging the mouse within the picture, clicking buttons, or typing coordinate values. You can also zoom in and out to view the pixel in more or less detail and you can see the actual color of the pixel in a large square.

I will use a very simple sample program to illustrate the creation of an object of the **PictureExplorer** class.

The program named Java362a begins by creating a new **Picture** object with known dimensions using input from an image file in the current directory. Then it calls the **show** method on the object to produce the screen output shown in Figure 1.

**Figure 1. Screen output produced by the show method.**

## Call the explore method

After that, the program calls the **explore** method on the **Picture** object to produce the screen output shown in Figure 2.

**Figure 2. Screen output produced by the explore method.**


You learned all about the **show** method of the **SimplePicture** class in earlier lessons *(see Resources)*. We will be mainly interested in **explore** method and the **PictureExplorer** class in this and the next few lessons.

## The explore method

The **explore** method of the **SimplePicture** class contains a single statement, which instantiates an object of the **PictureExplorer** class. As soon as that object is instantiated, the GUI shown in Figure 2 appears on the screen.



## The purpose of the PictureExplorer class

The purpose of an object of the **PictureExplorer** class is to allow you to determine the numeric color values for any pixel in a picture by placing a cursor on the pixel. The pixel position is controlled by clicking or dragging the mouse

within the picture, clicking buttons, or typing coordinate values. You can also zoom in and out to view the pixel in more or less detail and you can see the actual color of the pixel in a large square.

More specifically, an object of the **PictureExplorer** class will:

- Display a **Picture** object in the format shown in Figure 2.
- Let you explore the picture by displaying the x, y, red, green, and blue values of the pixel at the cursor:
    - When you click a mouse button within the picture, or
    - When you press and hold a mouse button while moving the cursor within the picture.
- Let you zoom in or out, showing the individual pixels in more or less detail.
- Let you type in x and y coordinate values to see the color at that location.
- Let you adjust the location of the cursor one pixel at a time by clicking *previous* and *next* buttons to specify a new pair of x and y coordinate values.
- Let you view the color of the pixel at the cursor in a large colored square. *(See the nearly black square in Figure 2.)*

An object of the **PictureExplorer** class is a powerful tool for examining pixel colors in a picture.

**An excellent educational example**

As I mentioned earlier, and as indicated by the behavior described above, the **PictureExplorer** class is a large and complex class containing a significant amount of event-driven GUI code. It even instantiates event-listener objects from anonymous classes, which is a very important Java programming topic. Therefore, the **PictureExplorer** class is an excellent class to be studied for an understanding of the use of event-driven GUI programming in the real world.

## Reducing the confusion

The **PictureExplorer** class contains a private inner class named **PictureExplorerFocusTraversalPolicy**, which I will also explain at some point in this series of lessons. It also makes heavy use of a class named **ImageDisplay** from Ericson's library.

Because I will be switching back and forth among code fragments extracted from Ericson's **PictureExplorer** class, code fragments extracted from other classes in Ericson's library, and code fragments extracted from my sample programs, things can get confusing.

**Background color for the PictureExplorer class**

In an attempt to reduce the confusion, I will present code fragments extracted from Ericson's **PictureExplorer** class against the background color shown in Listing 1.

**Listing 1. Background color for the PictureExplorer class.**

```
I will present code fragments from the
PictureExplorer
class against this background color.
```

### Background color for other Ericson classes

Similarly, I will present code fragments extracted from other classes in Ericson's library against the background color shown in Listing 2.

**Listing 2. Background color for other Ericson classes.**

```
I will present code fragments from other
Ericson classes
against this background color.
```

### Background color for my sample programs

Finally, I will present code fragments extracted from my sample programs against the background color shown in Listing 3.

**Listing 3. Background color for the program named Java362a.**

```
I will present code fragments from my sample
programs
against this background color.
```

In the event that I need to distinguish among more than three classes in the same lesson, I will come up with a fourth color and explain its use at the time.

# Discussion and sample code

## The sample program named Java362a

The purpose of this simple program is to support an explanation of the **PictureExplorer** class.

Normally, I break programs down and explain them in fragments. However, this program is so short and so simple that the program is shown in its entirety in Listing 4. *(It is also provided in Listing 14 near the end of the lesson for easy reference.)*

**Listing 4. Source code for the program named Java362a.**

```
public class Main{
  public static void main(String[] args){
    //Construct a new 460x345 Picture object.
    Picture pix1 = new
Picture("ScaledBeach460x345.jpg");
    pix1.show();//display the picture in the
show format
    //Display the picture again in the explore
format.
    pix1.explore();
  }//end main method
}//end class Main
```

**Create a Picture object and display it with the show method**

A **Picture** object having dimensions of 450x345 pixels is created by reading an image file in the current directory. Then the **show** method is called on the **Picture** object producing the screen output shown in Figure 1.

Following that, the **explore** method is called on the object, producing the screen output shown in Figure 2.

As I mentioned earlier, the **explore** method simply creates a new object of the **PictureExplorer** class. The GUI shown in Figure 2 appears on the screen as soon as that object is created. Figure 3 shows another view of the GUI with the **Zoom** menu pulled down and the GUI resized to a smaller size with scrollbars showing.

## The big picture view of the GUI

Because things are going to become progressively more complicated, it will probably be worthwhile for us to step back and take a big-picture look at the GUI shown in Figure 3.

**A JFrame object**

The onscreen window that you see in Figure 3 is the visual manifestation of a **JFrame** object. Basically, a **JFrame** object consists of the following parts:

- A banner at the top containing some built-in control components *(three buttons on the right and a menu on the left)* and optionally containing a **String** title.
- A rectangular area under the banner that can contain one or more menu labels side-by-side. This area is collapsed if you elect not to provide menus.
- A content area underneath the menu area.
- A border around the outer edges.

**The menu**

A **JMenuBar** object has been placed in the menu area for the **JFrame** object shown in Figure 3. A single **JMenu** object has been added to the menu bar with a label of **Zoom**. Seven **JMenuItem** objects have been added to the **JMenu** object.

### The content area

The content area *(immediately below the menu area)* has a default **BorderLayout** object as the layout manager. This layout manager makes it possible to place one component in the CENTER and four additional components in the NORTH, SOUTH, EAST, and WEST locations. In this case, there is one component in the CENTER and one component in the NORTH location. There are no components in the EAST, SOUTH, and WEST locations. *(Keep in mind that each of the five allowable components can themselves contain other components.)*

### A JScrollPane object in the CENTER location

The component in the CENTER is an object of the **JScrollPane** class. Without getting into the details at this point, the scroll pane makes it possible to view and scroll an object of Ericson's **ImageDisplay** class. **ImageDisplay** is a subclass of the **JPanel** class with the ability to render and display an **Image** object.

You will learn more about the detailed structure of the component in the CENTER later, including the details of the *mouse* and *mouse motion* listener objects registered on the component.

### A JPanel object in the NORTH location

The component in the NORTH location of the **JFrame** object's content area is a **JPanel** object with the layout manager also set to **BorderLayout**. This **JPanel** object contains two smaller **JPanel** objects, one in its NORTH location and one in its SOUTH location. There are no components in the CENTER, EAST, or WEST locations of the **JPanel** object.

A **JPanel** object is also a container that can contain other components. However, there is no content pane associated with a **JPanel** object. Other components are added directly to the **JPanel** object.

### The locationPanel and the colorInfoPanel

The **JPanel** object in the NORTH location *(of the **JPanel** object in the NORTH location of the content pane)* is referred to in this class as the **locationPanel**. The construction of this panel is very complex with numerous components and numerous registered listener objects.

The **JPanel** object in the SOUTH location of that same panel is referred to as the **colorInfoPanel**. The construction of this panel is less complex than the construction of

the **locationPanel**. Among other things, this panel is completely passive with no registered listener objects.

### The locationPanel and the FlowLayout manager

The layout manager for the **locationPanel** is an object of the **FlowLayout** class. With this layout manager, you can add any number of components to the container component and they will position themselves in horizontal rows. If there are too many components to fit on one row, some will spill over to the next row. You can cause the components on the rows to be aligned to the left, right, or center.

### The population of the locationPanel

The **locationPanel** is primarily populated with the following components:

- A **Box** object (*I will explain this at the appropriate time.)*
- Some **JLabel** objects.
- Some **JTextField** objects.
- Some **ImageIcon** objects that are used to put the triangle images on the *next* and *previous* buttons on the left and right ends of the text fields.
- Some **JButton** objects that serve as the next and previous buttons.

There are lots of event handlers registered on various components in the **locationPanel**.

You will learn how all of the components are put together and how they behave later when we dig into the code for the **locationPanel** in detail.

### The colorInfoPanel

The **colorInfoPanel** is also a **JPanel** object, and the layout manager for the **colorInfoPanel** is also an object of the **FlowLayout** class.

As I mentioned earlier, the **colorInfoPanel** is much simpler than the **locationPanel** and is primarily populated with the following components:

- Some **JLabel** objects.
- Another **JPanel** object *(the small nearly black square in Figure 3)*.

There are no listener objects registered on components on the **colorInfoPanel**.

Once again, you will learn how these components are put together and how they behave later when we dig into the programming details for the **colorInfoPanel**.

## The PictureExplorer class

A complete listing of the **PictureExplorer** class is provided in Listing 13 near the end of the lesson. I will break the class down and explain it in fragments. The beginning of the class is shown in Listing 5.

**Listing 5. Beginning of the PictureExplorer class.**

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.image.*;
import javax.swing.border.*;

public class PictureExplorer implements
       MouseMotionListener, ActionListener,
MouseListener{
```

## The class implements three listener interfaces

The most significant thing about Listing 5 is that the class implements the following listener interfaces:

- MouseMotionListener
- ActionListener
- MouseListenener

This means, among other things, that an object of the class is a listener for a variety of event types fired by components that it owns. In other words, it is a listener on itself. This, in turn, means that the class must provide concrete definitions for all of the event-handler methods declared in the three interfaces listed above.

**A listener on itself**
If you are unfamiliar with the concept that an object of a class can be a listener on itself, see my earlier lesson titled *Component Events* in Resources.

## A quick look ahead

For example, a quick look ahead indicates that the object of the **PictureExplorer** class is registered to listen for action events fired by the seven different menu items on the *Zoom* menu. *(See the upper-left corner of Figure 3, for the Zoom menu.)* Also, the same object of the **PictureExplorer** class is registered to listen for *mouse* events and *mouse motion* events on the image display in the center of the **JFrame**.

## Anonymous classes

In addition, action listener objects instantiated from *anonymous classes* are registered to listen for events fired by the left and right arrow buttons on either side of the text fields and on the text fields themselves.

**Anonymous classes**
For more information on anonymous classes, see lessons 1636 through 1642 in Resources.

## Private instance variables of the PictureExplorer class

The **PictureExplorer** class declares a large number of private instance variables and initializes a couple of them. They are shown in Listing 6 for later reference.

**Listing 6. Private instance variables of the PictureExplorer class.**

```
 // current x and y index
 private int xIndex = 0;
 private int yIndex = 0;

 //Main gui variables
 private JFrame pictureFrame;
 private JScrollPane scrollPane;

 //information bar variables
 private JLabel xLabel;
 private JButton xPrevButton;
 private JButton yPrevButton;
 private JButton xNextButton;
 private JButton yNextButton;
 private JLabel yLabel;
 private JTextField xValue;
 private JTextField yValue;
 private JLabel rValue;
 private JLabel gValue;
 private JLabel bValue;
 private JLabel colorLabel;
 private JPanel colorPanel;

 // menu components
 private JMenuBar menuBar;
 private JMenu zoomMenu;
 private JMenuItem twentyFive;
 private JMenuItem fifty;
 private JMenuItem seventyFive;
 private JMenuItem hundred;
 private JMenuItem hundredFifty;
 private JMenuItem twoHundred;
 private JMenuItem fiveHundred;

 /** The picture being explored */
 private DigitalPicture picture;

 /** The image icon used to display the
picture */
 private ImageIcon scrollImageIcon;

 /** The image display */
 private ImageDisplay imageDisplay;

 /** the zoom factor (amount to zoom) */
 private double zoomFactor;
```

```
 /** the number system to use, 0 means
starting at 0,
  *  1 means starting at 1 */
 private int numberBase=0;
```

There's not much to be said about the instance variables at this point. We will be referring back to them as the explanation of the class progresses.

**The constructor**

The **PictureExplorer** class provides only one constructor and it is shown in its entirety in Listing 7.

**Listing 7. The constructor for the PictureExplorer class.**

```
 /**
  * Public constructor
  * @param picture the picture to explore
  */
 public PictureExplorer(DigitalPicture
picture){
   // set the fields
   this.picture=picture;
   zoomFactor=1;

   // create the window and set things up
   createWindow();
 }//end constructor
```

**Save the incoming parameter**

The constructor receives a reference to the **Picture** object that is to be displayed as the interface type **DigitalPicture**. *(Recall from an earlier lesson that the **Picture** class implements the **DigitalPicture** interface. See the earlier lesson titled The DigitalPicture Interface: Multimedia Programming with Java in Resources for more information.)*

The incoming reference is stored in the instance variable named **picture** that was declared in Listing 6.

**Set the zoom factor**

Then the constructor sets the value of the instance variable named **zoomFactor** to 1. This instance variable is used at startup to cause the picture to be displayed at its natural size when the GUI first appears on the screen. *(The user can modify this later using the Zoom menu in the upper-left corner of Figure 3.)*

**Create the GUI window**

Finally, the constructor in Listing 7 calls the **createWindow** method to begin the overall construction process. When the **createWindow** method returns, the constructor returns the new **PictureExplorer** object's reference to the **explore** method of the **SimplePicture** class. However, the **explore** method doesn't save the reference in a named reference variable. Therefore, the **PictureExplorer** object is an anonymous object that will remain on the screen until the user clicks the X-button in the upper-right corner of Figure 3.

**The createWindow method**

The createWindow method that is called by the constructor is shown in its entirety in Listing 8.

**Listing 8. The createWindow method.**

```
 /**
  * Creates the JFrame and sets everything up
  */
 private void createWindow(){
   // create the picture frame and initialize
it
   createAndInitPictureFrame();

   // set up the menu bar
   setUpMenuBar();

   //create the information panel
   createInfoPanel();

   //creates the scrollpane for the picture
   createAndInitScrollingImage();

   // show the picture in the frame at the
size it needs
   // to be
   pictureFrame.pack();
   pictureFrame.setVisible(true);
 }//end createWindow method
```

## A sequence of method calls

As you can see, the code in the **createWindow** method consists of:

- A sequence of four calls to other methods to construct various parts of the **PictureExplorer** object.
- A call to the **pack** method to set the **JFrame** to the correct size.
- A call to the **setVisible** method to cause the **JFrame** object to become visible on the screen.

I will explain each of those methods in sequence.

### The createAndInitPictureFrame method

You have seen the term **PictureFrame** used in earlier lessons in this series *(see Resources)*. However, although the general intent is the same, the implementation of the picture frame in this class is different from the implementation that you learned about in conjunction with the **show** method *(see Resources)*.

As you learned in an earlier lesson, the **SimplePicture** class has a private instance variable named **pictureFrame**, which is of type **PictureFrame**. This instance variable holds a reference to an object instantiated from Ericson's class named **PictureFrame**.

The **PictureExplorer** class also has a private instance variable named **pictureFrame** *(see Listing 6)*, but it is of type **JFrame**. It holds a reference to an object of the **JFrame** class *(not the **PictureFrame** class)*.

### The JFrame object

That **JFrame** object is instantiated and its reference is assigned to the instance variable named **pictureFrame** at the beginning of the method shown in Listing 9.

**Listing 9. The createAndInitPictureFrame method.**

```
 /**
  * Method to create and initialize the picture
frame
  */
 private void createAndInitPictureFrame(){
   pictureFrame = new JFrame(); // create the
JFrame
   //allow the user to resize it
   pictureFrame.setResizable(true);
    // use border layout
   pictureFrame.getContentPane().setLayout(
                                       new
BorderLayout());
    // when close stop
   pictureFrame.setDefaultCloseOperation(

JFrame.DISPOSE_ON_CLOSE);
   pictureFrame.setTitle(picture.getTitle());
   PictureExplorerFocusTraversalPolicy newPolicy
=
                new
PictureExplorerFocusTraversalPolicy();

pictureFrame.setFocusTraversalPolicy(newPolicy);

 }//end createAndInitPictureFrame method
```

### Swing components...

If you are familiar with the use of Swing components such as **JFrame** objects, you probably won't find anything in Listing 9 that you don't understand *(with the possible exception of the focus traversal material)*.

## Why setResizable?

Having created the **JFrame** object, Listing 9 calls the **setResizable** method on the object to make it possible for the user to resize the GUI by dragging the edges.

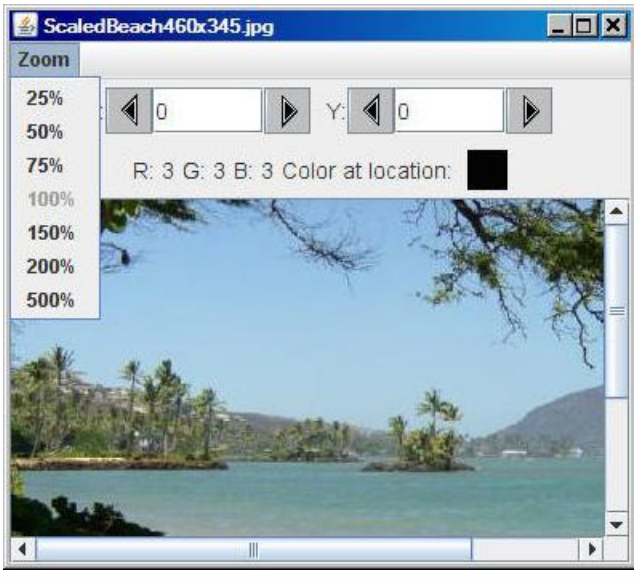
I'm not sure why Ericson included this statement, since a **JFrame** object is resizable by default. *(Perhaps she just wanted to make the code more self-documenting.)*

It is beneficial for the GUI to be resizable. If you zoom up to show the pixels in more detail, the overall size of the image increases. You can then manually resize the GUI to make it possible to view the entire larger image without the use of scroll bars *(assuming that your screen is large enough to accommodate the larger image.)*.

## Resized PictureExplorer GUI with Zoom menu exposed

Figure 3 shows the result of resizing the GUI to make it smaller and pulling down the Zoom menu to expose the seven zoom options.

**Figure 3. Resized PictureExplorer GUI with Zoom menu exposed.**




## Set the layout manager

Then Listing 9 sets the layout manager on the **ContentPane** to **BorderLayout**. Again, I'm not sure why because **BorderLayout** is the default for the **ContentPane**. In any event, **BorderLayout** works nicely because the image can be displayed in the

**The getContentPane method**
If you are unfamiliar with the *content pane* concept, see my earlier lesson titled *Swing from A to Z, Some Simple Components* in Resources.

CENTER position and a panel containing the text fields, navigation buttons, etc., can be placed in the NORTH position as shown in Figure 3.

## Set the default close operation

In case you are not familiar with this procedure, see the lesson titled *Swing from A to Z: Analyzing Swing Components, Part 3, Construction* in Resources.

Calling the **setDefaultCloseOperation** method and passing an acceptable numeric constant as a parameter specifies the required behavior when the user clicks the X-button in the upper-right corner of Figure 2.

According to the Sun documentation, the behavior specified by the constant named **DISPOSE_ON_CLOSE** is as follows:

> *"Releases all of the native screen resources used by this Window, its subcomponents, and all of its owned children. That is, the resources for these Components will be destroyed, any memory they consume will be returned to the OS, and they will be marked as undisplayable."*

In short, this causes the GUI to go away, releasing all supporting resources in the process. *(This is not the default close operation, which is to simply hide the **JFrame**.)*

## Set the title

After setting the close operation, Listing 9 gets the **String** title belonging to the picture that was received as a parameter and sets that as the title for the **JFrame**.

## Set the focus traversal policy

To make a long and complicated story short, instantiating a new object of the **PictureExplorerFocusTraversalPolicy** *(which is a private inner class of the **PictureExplorer** class)* and passing its reference to the **setFocusTraversalPolicy** method specifies how the focus moves from one component to the next when the user presses the tab key.

This is a fairly complex topic, and I am going to defer an explanation until later when I explain the private inner class named **PictureExplorerFocusTraversalPolicy**.

When the **setFocusTraversalPolicy** method returns, the **createAndInitPictureFrame** method shown in Listing 9 will terminate, returning control to the **createWindow** method shown in Listing 8.

## Setting up the menu bar

The next method call in Listing 8 is a call to the **setUpMenuBar** method. As you might conclude from the name, this method sets up the Zoom menu shown in Figure 3.

Creating menus on a Java GUI isn't conceptually difficult, but it can require a great deal of tedious work.

Let's start by taking another look at the instance variables shown in Listing 6. Near the middle of that listing, you will find nine instance variables under a comment that reads **menu components**.

The last seven of those nine instance variables with names like **twentyFive** and **fiveHundred** are all of type **JMenuItem**. They will be used to represent the seven menu items shown in the Zoom menu in the upper-left corner of Figure 3. Later on, an action listener will be registered on each menu item to provide the required behavior when the user selects a menu item.

The two instance variables in Listing 6 named **zoomMenu** and **menuBar** will be used to construct the menu and attach it to the **JFrame** object as shown in Figure 3.

**The setUpMenuBar method**

The **setUpMenuBar** method, which is called by the **createWindow** method in Listing 8, begins in Listing 10.

**Listing 10. Beginning of the setUpMenuBar method.**

```
 /**
  * Method to create the menu bar, menus, and
menu items
  */
 private void setUpMenuBar(){
   //create menu
   menuBar = new JMenuBar();
   zoomMenu = new JMenu("Zoom");
   twentyFive = new JMenuItem("25%");
   fifty = new JMenuItem("50%");
   seventyFive = new JMenuItem("75%");
   hundred = new JMenuItem("100%");
   hundred.setEnabled(false);
   hundredFifty = new JMenuItem("150%");
   twoHundred = new JMenuItem("200%");
   fiveHundred = new JMenuItem("500%");
```

## A disabled menu item

Take another look at Figure 3 and you will see that the menu item labeled **100%** is disabled. This is because that is the default zoom factor at startup and I didn't select any other zoom factor from the menu before capturing the screen shot shown in Figure 3. The menu item is disabled because it is already in effect.

We will see later that when we select other zoom factors from the menu, the selected zoom factor will become disabled as soon as it takes effect.

## Instantiate a new JMenuItem object for 100%

There is a statement near the middle of the code in Listing 10 that instantiates a new **JMenuItem** object with a label of **100%** and assigns that object's reference to the instance variable named **hundred**. *(I colored that statement and the one following in red in Listing 10 to make it easy for you to find. Of course there is no color associated with Java source code.)*

Immediately below the assignment of the object's reference to the instance variable named **hundred**, there is a call to the **setEnabled** method on that reference, passing *false* as a parameter. This is one way that menu items *(and some other GUI components as well)* are enabled and disabled in Java. *(For another approach, see the lesson titled Understanding Action Objects in Java in Resources.)*

The remaining eight statements in Listing 10 simply instantiate new objects *(of the correct type with the correct labels)* and assign the object's references to the corresponding instance variables. *(Note that a **JMenuBar** object doesn't have a label.)*

## Register an action listener on the menu items

Remember that I told you earlier that the structure of this program is such that the object of the **PictureExplorer** class serves as a listener object on certain components that belong to the object.

Listing 11 calls the **addActionListener** method seven times in succession to register the **PictureExplorer** object as an action listener on each of the menu items. *(If you are unfamiliar with the registration of listener objects on event sources, see the lesson titled Event Handling in JDK 1.1, A First Look, Delegation Event Model in Resources.)*

**Listing 11. Register an action listener on the menu items.**

```
    // add the action listeners
    twentyFive.addActionListener(this);
    fifty.addActionListener(this);
    seventyFive.addActionListener(this);
    hundred.addActionListener(this);
    hundredFifty.addActionListener(this);
    twoHundred.addActionListener(this);
    fiveHundred.addActionListener(this);
```

## Only one of many ways

There are many ways to structure code for an event-driven GUI in Java, each of which has advantages and disadvantages depending on the application. The fact that the

same listener object *(this object)* is being registered on all of the menu items means that the **actionPerformed** method of the **PictureExplorer** class, *(which will be defined later)*, must identify which menu item was selected by the user and then take the required action for that particular menu item.

## Complete the construction of the menu

The required code for completing the construction of the menu and attaching it to the **JFrame** object is shown in Listing 12.

**Listing 12. Complete the construction of the menu.**

```
   // add the menu items to the menus
   zoomMenu.add(twentyFive);
   zoomMenu.add(fifty);
   zoomMenu.add(seventyFive);
   zoomMenu.add(hundred);
   zoomMenu.add(hundredFifty);
   zoomMenu.add(twoHundred);
   zoomMenu.add(fiveHundred);

   menuBar.add(zoomMenu);

   // set the menu bar to this menu
   pictureFrame.setJMenuBar(menuBar);
}// end setUpMenuBar method
```

## What is a JMenu object?

The instance variable named **zoomMenu** contains a reference to an object of the **JMenu** class *(see Listing 10)*. Briefly, Sun describes such an object as:

> *"An implementation of a menu -- a popup window containing JMenuItems that is displayed when the user selects an item on the JMenuBar. In addition to JMenuItems, a JMenu can also contain JSeparators.*
>
> *In essence, a menu is a button with an associated JPopupMenu. When the "button" is pressed, the JPopupMenu appears. If the "button" is on the JMenuBar, the menu is a top-level window. If the "button" is another menu item, then the JPopupMenu is a "pull-right" menu."*

Stated differently, a **JMenu** object is a container that can contain multiple **JMenuItem** and **JSeparator** objects as shown in Figure 3. *(There are no **JSeparator** objects in Figure 3.)*

## JMenuItem objects fire action events

**JSeparator** objects are mainly used for cosmetic purposes to separate the menu items into groups and they do not fire events. However, the **JMenuItem** objects are active sources of action events. When the user selects a **JMenuItem** object, it will fire an action event.

### If an action listener has been registered...

If an action listener object has been registered on the menu item *(as in Listing 11)*, the firing of the action event will cause the method named **actionPerformed** belonging to the listener object to be called. The behavior of the **actionPerformed** method will be the response to the event.

### If a tree falls in the woods and there is no one to hear it...

The event is fired when the user selects the menu item regardless of whether or not there are any registered listeners. If there are no registered listeners, there simply is no programmatic response to the event.

### Construct the menu

The first seven statements in Listing 12 add the **JMenuItem** objects to the **JMenu** object, producing the menu that you see in Figure 3.

### An object of the JMenuBar class

The instance variable named **menuBar** contains a reference to an object of the **JMenuBar** class. Briefly, Sun has this to say about an object of the **JMenuBar** class:

> *"An implementation of a menu bar. You add **JMenu** objects to the menu bar to construct a menu. When the user selects a **JMenu** object, its associated **JPopupMenu** is displayed, allowing the user to select one of the **JMenuItem** objects on it."*

Stated differently, a **JMenuBar** object is a container, which can contain multiple **JMenu** objects. However, in this class, we are placing only one **JMenu** object *(labeled Zoom)* in the container. We accomplish that by calling the **add** method on the **JMenuBar** object and passing the **JMenu** object's reference as a parameter *to the **add** method (see Listing 12)*.

### The behavior of the JMenu object

In Figure 3, the component with the label **Zoom** is the **JMenu** object and it has been added to a horizontal **JMenuBar** object at this point. *(JMenuBar objects don't have labels, but the one in Figure 3 can be seen if you look closely enough.)*

When the user selects one of the **JMenu** objects *(in this case, there is only one)*, the menu opens to expose the **JMenuItem** objects without any help from the programmer.

### Attach the JMenuBar object to the JFrame object

Finally, the code in Listing 12 attaches the **JMenuBar** object to the **JFrame** object by calling the **setJMenuBar** method on the reference to the **JFrame** object and passing the **JMenuBar** object's reference as a parameter.

### Significantly different from adding components

Note that this is significantly different from adding GUI components to a **JFrame** object. When you add GUI components to a **JFrame** object, you must add them to the **JFrame** object's content pane, *(which is a topic that is far too complex to explain in this lesson)*. You cannot add components directly to the **JFrame** object.

However, when you call the **setJMenuBar** method to attach the menu bar to the **JFrame**, you call the method directly on the reference to the **JFrame** object and the content pane is not involved.

### When the setJMenuBar method returns...

When the call to the **setJMenuBar** method returns in Listing 12, the **setUpMenuBar** method, *(which began in Listing 10)*, terminates and returns control to the **createWindow** method in Listing 8.

At this point, the menu is ready for use with one major exception. The required behavior associated with the selection of each menu item hasn't been established yet.

The required behavior for each menu item will be established later through the definition of the **actionPerformed** method of the **PictureExplorer** class.

## Run the program

### It's time to take a break

We have a long way to go before we can fully understand the **PictureExplorer** class. However, you have been working hard if you have made it to this point. It's time to take a break, drink some coffee, eat a doughnut *(or maybe some tofu if you prefer that)* and let what you have learned so far sink in.

### Experiment

Although the sample program doesn't amount to much, I encourage you to copy the code from Listing 14, compile the code, and execute it using an image file of your

choice.  Experiment with the code, making changes, and observing the results of your changes.  Make certain that you can explain why your changes behave as they do.

Also experiment with the GUI that is produced when the program calls the **explore** method on your **Picture** object.

# Summary

The purpose of an object of the **PictureExplorer** class is to allow you to determine the numeric color values for any pixel in a picture by placing a cursor on the pixel.  The pixel position is controlled by clicking or dragging the mouse within the picture, clicking buttons, or typing coordinate values.  You can also zoom in and out to view the pixel in more or less detail and you can see the actual color of the pixel in a large square.

So far, we have been concentrating on the construction of the graphical user interface that appears when you call the **explore** method on a **Picture** object.

You have learned how the GUI is constructed from a big-picture viewpoint.

You have learned that the GUI window is the visual manifestation of a **JFrame** object, and you have learned how the **JFrame** object is configured.

You have learned how the Zoom menu is constructed and how an action listener object is registered on the items in that menu.

We have yet to get into the behavior that is programmed into the various event handlers that are registered on components in the GUI.  We will get to that later.

# What's next?

The next lesson will begin with an explanation of the method named **createInfoPanel**, which is called by the **createWindow** method in Listing 8 during the construction of the GUI.

The **createInfoPanel** method is probably the most challenging part of the entire **PictureExplorer** class *(with the possible exception of the focus-traversal material)*.

After that, we will continue with the sequence of methods that are called in the method named **createWindow**, which is at the core of constructing the GUI.

# Resources

- Creative Commons Attribution 3.0 United States License
- Media Computation book in Java - numerous downloads available
- Introduction to Computing and Programming with Java: A Multimedia Approach

- DrJava download site
- DrJava, the JavaPLT group at Rice University
- DrJava Open Source License
- The Essence of OOP using Java, The this and super Keywords
- Threads of Control
- Painting in AWT and Swing
- Wikipedia Turtle Graphics
- IsA or HasA
- Vector Cad-Cam XI Lathe Tutorial
- Classification of 3D to 2D projections
- Color model from Wikipedia
- Light and color:  an introduction by Norman Koren
- Color Principles - Hue, Saturation, and Value
- 200 Implementing the Model-View-Controller Paradigm using Observer and Observable
- 300 Java 2D Graphics, Nested Top-Level Classes and Interfaces
- 302 Java 2D Graphics, The Point2D Class
- 304 Java 2D Graphics, The Graphics2D Class
- 306 Java 2D Graphics, Simple Affine Transforms
- 308 Java 2D Graphics, The Shape Interface, Part 1
- 310 Java 2D Graphics, The Shape Interface, Part 2
- 312 Java 2D Graphics, Solid Color Fill
- 314 Java 2D Graphics, Gradient Color Fill
- 316 Java 2D Graphics, Texture Fill
- 318 Java 2D Graphics, The Stroke Interface
- 320 Java 2D Graphics, The Composite Interface and Transparency
- 322 Java 2D Graphics, The Composite Interface, GradientPaint, and Transparency
- 324 Java 2D Graphics, The Color Constructors and Transparency
- 400 Processing Image Pixels using Java, Getting Started
  402 Processing Image Pixels using Java, Creating a Spotlight
  404 Processing Image Pixels Using Java: Controlling Contrast and Brightness
  406 Processing Image Pixels, Color Intensity, Color Filtering, and Color Inversion
  408 Processing Image Pixels, Performing Convolution on Images
  410 Processing Image Pixels, Understanding Image Convolution in Java
  412 Processing Image Pixels, Applying Image Convolution in Java, Part 1
  414 Processing Image Pixels, Applying Image Convolution in Java, Part 2
  416 Processing Image Pixels, An Improved Image-Processing Framework in Java
  418 Processing Image Pixels, Creating Visible Watermarks in Java
  450 A Framework for Experimenting with Java 2D Image-Processing Filters
  452 Using the Java 2D LookupOp Filter Class to Process Images
  454 Using the Java 2D AffineTransformOp Filter Class to Process Images
  456 Using the Java 2D LookupOp Filter Class to Scramble and Unscramble Images
  458 Using the Java 2D BandCombineOp Filter Class to Process Images

460 Using the Java 2D ConvolveOp Filter Class to Process Images
462 Using the Java 2D ColorConvertOp and RescaleOp Filter Classes to Process Images

- 506 JavaBeans, Introspection
- 2100 Understanding Properties in Java and C#
- 2300 Generics in J2SE, Getting Started
- 340 Multimedia Programming with Java, Getting Started
- 342 Getting Started with the Turtle Class: Multimedia Programming with Java
- 344 Continuing with the SimpleTurtle Class: Multimedia Programming with Java
- 346 Wrapping Up the SimpleTurtle Class: Multimedia Programming with Java
- 348 The Pen and PathSegment Classes: Multimedia Programming with Java
- 349 A Pixel Editor Program in Java: Multimedia Programming with Java
- 350 3D Displays, Color Distance, and Edge Detection
- 351 A Slider-Controlled Softening Program for Digital Photos
- 352 Adding Animated Movement to Your Java Application
- 353 A Slider-Controlled Sharpening Program for Digital Photos
- 354 The DigitalPicture Interface
- 355 The HSB Color Model
- 356 The show Method and the PictureFrame Class
- 357 An HSB Color-Editing Program for Digital Photos
- 358 Applying Affine Transforms to Picture Objects
- 359 Creating a lasso for editing digital photos in Java
- 360 Wrapping Up the SimplePicture Class
- 361 A Temperature and Tint Editing Program for Digital Photos

# Complete program listings

Complete listings of the programs discussed in this lesson are shown in Listing 13 and Listing 14 below.

**Listing 13. Source code for Ericson's PictureExplorer class.**

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.image.*;
import javax.swing.border.*;
/**
 * Displays a picture and lets you explore the
picture by
 * displaying the x, y, red, green, and blue values
of the
 * pixel at the cursor when you click a mouse
button or
 * press and hold a mouse button while moving the
cursor.
 * It also lets you zoom in or out.  You can also
type in
```

```
 * a x and y value to see the color at that
location.
 *
 * Originally created for the Jython Environment
for
 * Students (JES).
 * Modified to work with DrJava by Barbara Ericson
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Keith McDermottt, gte047w@cc.gatech.edu
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class PictureExplorer implements
       MouseMotionListener, ActionListener,
MouseListener{

 // current x and y index
 private int xIndex = 0;
 private int yIndex = 0;

 //Main gui variables
 private JFrame pictureFrame;
 private JScrollPane scrollPane;

 //information bar variables
 private JLabel xLabel;
 private JButton xPrevButton;
 private JButton yPrevButton;
 private JButton xNextButton;
 private JButton yNextButton;
 private JLabel yLabel;
 private JTextField xValue;
 private JTextField yValue;
 private JLabel rValue;
 private JLabel gValue;
 private JLabel bValue;
 private JLabel colorLabel;
 private JPanel colorPanel;

 // menu components
 private JMenuBar menuBar;
 private JMenu zoomMenu;
 private JMenuItem twentyFive;
 private JMenuItem fifty;
 private JMenuItem seventyFive;
 private JMenuItem hundred;
 private JMenuItem hundredFifty;
 private JMenuItem twoHundred;
 private JMenuItem fiveHundred;

 /** The picture being explored */
 private DigitalPicture picture;

 /** The image icon used to display the picture */
 private ImageIcon scrollImageIcon;
```

```
 /** The image display */
 private ImageDisplay imageDisplay;

 /** the zoom factor (amount to zoom) */
 private double zoomFactor;

 /** the number system to use, 0 means starting at
0,
  *  1 means starting at 1 */
 private int numberBase=0;

 /**
  * Public constructor
  * @param picture the picture to explore
  */
 public PictureExplorer(DigitalPicture picture)
 {
   // set the fields
   this.picture=picture;
   zoomFactor=1;

   // create the window and set things up
   createWindow();
 }

 /**
  * Changes the number system to start at one
  */
 public void changeToBaseOne()
 {
   numberBase=1;
 }

 /**
  * Set the title of the frame
  *@param title the title to use in the JFrame
  */
 public void setTitle(String title)
 {
   pictureFrame.setTitle(title);
 }

 /**
  * Method to create and initialize the picture
frame
  */
 private void createAndInitPictureFrame()
 {
   pictureFrame = new JFrame(); // create the
JFrame
   //allow the user to resize it
   pictureFrame.setResizable(true);
    // use border layout
   pictureFrame.getContentPane().setLayout(
                                       new
BorderLayout());
```

```
    // when close stop
   pictureFrame.setDefaultCloseOperation(

JFrame.DISPOSE_ON_CLOSE);
   pictureFrame.setTitle(picture.getTitle());
   PictureExplorerFocusTraversalPolicy newPolicy =
                new
PictureExplorerFocusTraversalPolicy();
   pictureFrame.setFocusTraversalPolicy(newPolicy);

 }

 /**
  * Method to create the menu bar, menus, and menu
items
  */
 private void setUpMenuBar()
 {
   //create menu
   menuBar = new JMenuBar();
   zoomMenu = new JMenu("Zoom");
   twentyFive = new JMenuItem("25%");
   fifty = new JMenuItem("50%");
   seventyFive = new JMenuItem("75%");
   hundred = new JMenuItem("100%");
   hundred.setEnabled(false);
   hundredFifty = new JMenuItem("150%");
   twoHundred = new JMenuItem("200%");
   fiveHundred = new JMenuItem("500%");

   // add the action listeners
   twentyFive.addActionListener(this);
   fifty.addActionListener(this);
   seventyFive.addActionListener(this);
   hundred.addActionListener(this);
   hundredFifty.addActionListener(this);
   twoHundred.addActionListener(this);
   fiveHundred.addActionListener(this);

   // add the menu items to the menus
   zoomMenu.add(twentyFive);
   zoomMenu.add(fifty);
   zoomMenu.add(seventyFive);
   zoomMenu.add(hundred);
   zoomMenu.add(hundredFifty);
   zoomMenu.add(twoHundred);
   zoomMenu.add(fiveHundred);
   menuBar.add(zoomMenu);

   // set the menu bar to this menu
   pictureFrame.setJMenuBar(menuBar);
 }

 /**
  * Create and initialize the scrolling image
  */
```

```
 private void createAndInitScrollingImage()
 {
   scrollPane = new JScrollPane();

   BufferedImage bimg = picture.getBufferedImage();
   imageDisplay = new ImageDisplay(bimg);
   imageDisplay.addMouseMotionListener(this);
   imageDisplay.addMouseListener(this);
   imageDisplay.setToolTipText("Click a mouse
button on "
                + "a pixel to see the pixel
information");
   scrollPane.setViewportView(imageDisplay);
   pictureFrame.getContentPane().add(
                          scrollPane,
BorderLayout.CENTER);
 }

 /**
  * Creates the JFrame and sets everything up
  */
 private void createWindow()
 {
   // create the picture frame and initialize it
   createAndInitPictureFrame();

   // set up the menu bar
   setUpMenuBar();

   //create the information panel
   createInfoPanel();

   //creates the scrollpane for the picture
   createAndInitScrollingImage();

   // show the picture in the frame at the size it
needs
   // to be
   pictureFrame.pack();
   pictureFrame.setVisible(true);
 }

 /**
  * Method to set up the next and previous buttons
for the
  * pixel location information
  */
 private void setUpNextAndPreviousButtons()
 {
   // create the image icons for the buttons
   Icon prevIcon = new ImageIcon(

SoundExplorer.class.getResource("leftArrow.gif"),
                                          "previous
index");
   Icon nextIcon = new ImageIcon(
```

```
SoundExplorer.class.getResource("rightArrow.gif"),
                                                "next
index");
   // create the arrow buttons
   xPrevButton = new JButton(prevIcon);
   xNextButton = new JButton(nextIcon);
   yPrevButton = new JButton(prevIcon);
   yNextButton = new JButton(nextIcon);

   // set the tool tip text
   xNextButton.setToolTipText(
                        "Click to go to the next x
value");
   xPrevButton.setToolTipText(
                    "Click to go to the previous x
value");
   yNextButton.setToolTipText(
                        "Click to go to the next y
value");
   yPrevButton.setToolTipText(
                    "Click to go to the previous y
value");

   // set the sizes of the buttons
   int prevWidth = prevIcon.getIconWidth() + 2;
   int nextWidth = nextIcon.getIconWidth() + 2;
   int prevHeight = prevIcon.getIconHeight() + 2;
   int nextHeight = nextIcon.getIconHeight() + 2;
   Dimension prevDimension =
                       new
Dimension(prevWidth,prevHeight);
   Dimension nextDimension =
                      new Dimension(nextWidth,
nextHeight);
   xPrevButton.setPreferredSize(prevDimension);
   yPrevButton.setPreferredSize(prevDimension);
   xNextButton.setPreferredSize(nextDimension);
   yNextButton.setPreferredSize(nextDimension);

   // handle previous x button press
   xPrevButton.addActionListener(new
ActionListener() {
     public void actionPerformed(ActionEvent evt) {
       xIndex--;
       if (xIndex < 0)
         xIndex = 0;
       displayPixelInformation(xIndex,yIndex);
     }
   });

   // handle previous y button press
   yPrevButton.addActionListener(new
ActionListener() {
     public void actionPerformed(ActionEvent evt) {
       yIndex--;
```

```
       if (yIndex < 0)
         yIndex = 0;
       displayPixelInformation(xIndex,yIndex);
     }
   });

   // handle next x button press
   xNextButton.addActionListener(new
ActionListener() {
     public void actionPerformed(ActionEvent evt) {
       xIndex++;
       if (xIndex >= picture.getWidth())
         xIndex = picture.getWidth() - 1;
       displayPixelInformation(xIndex,yIndex);
     }
   });

   // handle next y button press
   yNextButton.addActionListener(new
ActionListener() {
     public void actionPerformed(ActionEvent evt) {
       yIndex++;
       if (yIndex >= picture.getHeight())
         yIndex = picture.getHeight() - 1;
       displayPixelInformation(xIndex,yIndex);
     }
   });
 }

 /**
  * Create the pixel location panel
  * @param labelFont the font for the labels
  * @return the location panel
  */
 public JPanel createLocationPanel(Font labelFont)
{

   // create a location panel
   JPanel locationPanel = new JPanel();
   locationPanel.setLayout(new FlowLayout());
   Box hBox = Box.createHorizontalBox();

   // create the labels
   xLabel = new JLabel("X:");
   yLabel = new JLabel("Y:");

   // create the text fields
   xValue = new JTextField(
                Integer.toString(xIndex +
numberBase),6);
   xValue.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
       displayPixelInformation(

xValue.getText(),yValue.getText());
     }
```

```
   });
   yValue = new JTextField(
                  Integer.toString(yIndex +
numberBase),6);
   yValue.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
      displayPixelInformation(

xValue.getText(),yValue.getText());
     }
   });

   // set up the next and previous buttons
   setUpNextAndPreviousButtons();

   // set up the font for the labels
   xLabel.setFont(labelFont);
   yLabel.setFont(labelFont);
   xValue.setFont(labelFont);
   yValue.setFont(labelFont);

   // add the items to the vertical box and the box
to
   // the panel
   hBox.add(Box.createHorizontalGlue());
   hBox.add(xLabel);
   hBox.add(xPrevButton);
   hBox.add(xValue);
   hBox.add(xNextButton);
   hBox.add(Box.createHorizontalStrut(10));
   hBox.add(yLabel);
   hBox.add(yPrevButton);
   hBox.add(yValue);
   hBox.add(yNextButton);
   locationPanel.add(hBox);
   hBox.add(Box.createHorizontalGlue());

   return locationPanel;
 }

 /**
  * Create the color information panel
  * @param labelFont the font to use for labels
  * @return the color information panel
  */
 private JPanel createColorInfoPanel(Font
labelFont)
 {
   // create a color info panel
   JPanel colorInfoPanel = new JPanel();
   colorInfoPanel.setLayout(new FlowLayout());

   // get the pixel at the x and y
   Pixel pixel = new Pixel(picture,xIndex,yIndex);

   // create the labels
```

```
   rValue = new JLabel("R: " + pixel.getRed());
   gValue = new JLabel("G: " + pixel.getGreen());
   bValue = new JLabel("B: " + pixel.getBlue());

   // create the sample color panel and label
   colorLabel = new JLabel("Color at location: ");
   colorPanel = new JPanel();
   colorPanel.setBorder(new
LineBorder(Color.black,1));

   // set the color sample to the pixel color
   colorPanel.setBackground(pixel.getColor());

   // set the font
   rValue.setFont(labelFont);
   gValue.setFont(labelFont);
   bValue.setFont(labelFont);
   colorLabel.setFont(labelFont);
   colorPanel.setPreferredSize(new
Dimension(25,25));

   // add items to the color information panel
   colorInfoPanel.add(rValue);
   colorInfoPanel.add(gValue);
   colorInfoPanel.add(bValue);
   colorInfoPanel.add(colorLabel);
   colorInfoPanel.add(colorPanel);

   return colorInfoPanel;
 }

 /**
  * Creates the North JPanel with all the pixel
location
  * and color information
  */
 private void createInfoPanel()
 {
   // create the info panel and set the layout
   JPanel infoPanel = new JPanel();
   infoPanel.setLayout(new BorderLayout());

   // create the font
   Font largerFont =
                   new
Font(infoPanel.getFont().getName(),

infoPanel.getFont().getStyle(),14);

   // create the pixel location panel
   JPanel locationPanel =
createLocationPanel(largerFont);

   // create the color informaiton panel
   JPanel colorInfoPanel =
```

```
createColorInfoPanel(largerFont);

   // add the panels to the info panel
   infoPanel.add(BorderLayout.NORTH,locationPanel);

infoPanel.add(BorderLayout.SOUTH,colorInfoPanel);

   // add the info panel
   pictureFrame.getContentPane().add(

BorderLayout.NORTH,infoPanel);
 }

 /**
  * Method to check that the current position is in
the
  * viewing area and if not scroll to center the
current
  * position if possible
  */
 public void checkScroll()
 {
   // get the x and y position in pixels
   int xPos = (int) (xIndex * zoomFactor);
   int yPos = (int) (yIndex * zoomFactor);

   // only do this if the image is larger than
normal
   if (zoomFactor > 1) {

     // get the rectangle that defines the current
view
     JViewport viewport = scrollPane.getViewport();
     Rectangle rect = viewport.getViewRect();
     int rectMinX = (int) rect.getX();
     int rectWidth = (int) rect.getWidth();
     int rectMaxX = rectMinX + rectWidth - 1;
     int rectMinY = (int) rect.getY();
     int rectHeight = (int) rect.getHeight();
     int rectMaxY = rectMinY + rectHeight - 1;

     // get the maximum possible x and y index
     int maxIndexX =
(int)(picture.getWidth()*zoomFactor)
                           - rectWidth - 1;
     int maxIndexY =
(int)(picture.getHeight()*zoomFactor)
                           - rectHeight - 1;

     // calculate how to position the current
position in
     // the middle of the viewing area
     int viewX = xPos - (int) (rectWidth / 2);
     int viewY = yPos - (int) (rectHeight / 2);

     // reposition the viewX and viewY if outside
```

```
allowed
     // values
     if (viewX < 0)
       viewX = 0;
     else if (viewX > maxIndexX)
       viewX = maxIndexX;
     if (viewY < 0)
       viewY = 0;
     else if (viewY > maxIndexY)
       viewY = maxIndexY;

     // move the viewport upper left point
     viewport.scrollRectToVisible(
         new
Rectangle(viewX,viewY,rectWidth,rectHeight));
   }
 }

 /**
  * Zooms in the on picture by scaling the image.
  * It is extremely memory intensive.
  * @param factor the amount to zoom by
  */
 public void zoom(double factor)
 {
   // save the current zoom factor
   zoomFactor = factor;

   // calculate the new width and height and get an
image
   // that size
   int width = (int)
(picture.getWidth()*zoomFactor);
   int height = (int)
(picture.getHeight()*zoomFactor);
   BufferedImage bimg = picture.getBufferedImage();

   // set the scroll image icon to the new image

imageDisplay.setImage(bimg.getScaledInstance(width,
                           height,
Image.SCALE DEFAULT));
   imageDisplay.setCurrentX((int) (xIndex *
zoomFactor));
   imageDisplay.setCurrentY((int) (yIndex *
zoomFactor));
   imageDisplay.revalidate();
   checkScroll();  // check if need to reposition
scroll
 }

 /**
  * Repaints the image on the scrollpane.
  */
 public void repaint()
 {
```

```
   pictureFrame.repaint();
 }

 //*****************************************//
 //               Event Listeners           //
 //*****************************************//

 /**
  * Called when the mouse is dragged (button held
down and
  * moved)
  * @param e the mouse event
  */
 public void mouseDragged(MouseEvent e)
 {
   displayPixelInformation(e);
 }

 /**
  * Method to check if the given x and y are in the
  * picture
  * @param x the horiztonal value
  * @param y the vertical value
  * @return true if the x and y are in the picture
and
  * false otherwise
  */
 private boolean isLocationInPicture(int x, int y)
 {
   boolean result = false; // the default is false
   if (x >= 0 && x < picture.getWidth() &&
       y >= 0 && y < picture.getHeight())
     result = true;

   return result;
 }

 /**
  * Method to display the pixel information from
the
  * passed x and y but also converts x and y from
strings
  * @param xString the x value as a string from the
user
  * @param yString the y value as a string from the
user
  */
 public void displayPixelInformation(
                           String xString, String
yString)
 {
   int x = -1;
   int y = -1;
   try {
     x = Integer.parseInt(xString);
     x = x - numberBase;
```

```
     y = Integer.parseInt(yString);
     y = y - numberBase;
   } catch (Exception ex) {
   }

   if (x >= 0 && y >= 0) {
     displayPixelInformation(x,y);
   }
 }

 /**
  * Method to display pixel information for the
passed x
  * and y
  * @param pictureX the x value in the picture
  * @param pictureY the y value in the picture
  */
 private void displayPixelInformation(
                                int pictureX, int
pictureY)
 {
   // check that this x and y is in range
   if (isLocationInPicture(pictureX, pictureY))
   {
     // save the current x and y index
     xIndex = pictureX;
     yIndex = pictureY;

     // get the pixel at the x and y
     Pixel pixel = new
Pixel(picture,xIndex,yIndex);

     // set the values based on the pixel
     xValue.setText(Integer.toString(
                                    xIndex  +
numberBase));
     yValue.setText(Integer.toString(
                                     yIndex +
numberBase));
     rValue.setText("R: " + pixel.getRed());
     gValue.setText("G: " + pixel.getGreen());
     bValue.setText("B: " + pixel.getBlue());
     colorPanel.setBackground(new
Color(pixel.getRed(),

pixel.getGreen(),

pixel.getBlue()));

   }
   else
   {
     clearInformation();
   }

   // notify the image display of the current x and
```

```
y
   imageDisplay.setCurrentX((int) (xIndex *
zoomFactor));
   imageDisplay.setCurrentY((int) (yIndex *
zoomFactor));
 }

 /**
  * Method to display pixel information based on a
mouse
  * event
  * @param e a mouse event
  */
 private void displayPixelInformation(MouseEvent e)
 {

   // get the cursor x and y
   int cursorX = e.getX();
   int cursorY = e.getY();

   // get the x and y in the original (not scaled
image)
   int pictureX = (int)(cursorX/zoomFactor +
numberBase);
   int pictureY = (int)(cursorY/zoomFactor +
numberBase);

   // display the information for this x and y
   displayPixelInformation(pictureX,pictureY);

 }

 /**
  * Method to clear the labels and current color
and
  * reset the  current index to -1
  */
 private void clearInformation()
 {
   xValue.setText("N/A");
   yValue.setText("N/A");
   rValue.setText("R: N/A");
   gValue.setText("G: N/A");
   bValue.setText("B: N/A");
   colorPanel.setBackground(Color.black);
   xIndex = -1;
   yIndex = -1;
 }

 /**
  * Method called when the mouse is moved with no
buttons
  * down
  * @param e the mouse event
  */
 public void mouseMoved(MouseEvent e)
```

```
 {}

 /**
  * Method called when the mouse is clicked
  * @param e the mouse event
  */
 public void mouseClicked(MouseEvent e)
 {
   displayPixelInformation(e);
 }

 /**
  * Method called when the mouse button is pushed
down
  * @param e the mouse event
  */
 public void mousePressed(MouseEvent e)
 {
   displayPixelInformation(e);
 }

 /**
  * Method called when the mouse button is released
  * @param e the mouse event
  */
 public void mouseReleased(MouseEvent e)
 {
 }

 /**
  * Method called when the component is entered
(mouse
  * moves over it)
  * @param e the mouse event
  */
 public void mouseEntered(MouseEvent e)
 {
 }

 /**
  * Method called when the mouse moves over the
component
  * @param e the mouse event
  */
 public void mouseExited(MouseEvent e)
 {
 }

 /**
  * Method to enable all menu commands
  */
 private void enableZoomItems()
 {
   twentyFive.setEnabled(true);
   fifty.setEnabled(true);
   seventyFive.setEnabled(true);
```

```
  hundred.setEnabled(true);
  hundredFifty.setEnabled(true);
  twoHundred.setEnabled(true);
  fiveHundred.setEnabled(true);
}

/**
 * Controls the zoom menu bar
 *
 * @param a the ActionEvent
 */
public void actionPerformed(ActionEvent a)
{

  if(a.getActionCommand().equals("Update"))
  {
    this.repaint();
  }

  if(a.getActionCommand().equals("25%"))
  {
    this.zoom(.25);
    enableZoomItems();
    twentyFive.setEnabled(false);
  }

  if(a.getActionCommand().equals("50%"))
  {
    this.zoom(.50);
    enableZoomItems();
    fifty.setEnabled(false);
  }

  if(a.getActionCommand().equals("75%"))
  {
    this.zoom(.75);
    enableZoomItems();
    seventyFive.setEnabled(false);
  }

  if(a.getActionCommand().equals("100%"))
  {
    this.zoom(1.0);
    enableZoomItems();
    hundred.setEnabled(false);
  }

  if(a.getActionCommand().equals("150%"))
  {
    this.zoom(1.5);
    enableZoomItems();
    hundredFifty.setEnabled(false);
  }

  if(a.getActionCommand().equals("200%"))
  {
```

```
     this.zoom(2.0);
     enableZoomItems();
     twoHundred.setEnabled(false);
   }

   if(a.getActionCommand().equals("500%"))
   {
     this.zoom(5.0);
     enableZoomItems();
     fiveHundred.setEnabled(false);
   }
 }

 /**
  * Test Main.  It will ask you to pick a file and
then
  * show it
  */
 public static void main( String args[])
 {
   Picture p = new
Picture(FileChooser.pickAFile());
   PictureExplorer test = new PictureExplorer(p);

 }

 /**
  * Class for establishing the focus for the
textfields
  */
 private class PictureExplorerFocusTraversalPolicy
                 extends FocusTraversalPolicy {

        /**
         * Method to get the next component for
focus
         */
        public Component getComponentAfter(
                                  Container
focusCycleRoot,
                                  Component
aComponent) {
            if (aComponent.equals(xValue))
              return yValue;
            else
              return xValue;
        }

        /**
         * Method to get the previous component for
focus
         */
         public Component getComponentBefore(
                                  Container
focusCycleRoot,
                                  Component
```

```
aComponent) {
            if (aComponent.equals(xValue))
              return yValue;
            else
              return xValue;
         }

         public Component getDefaultComponent(
                                Container
focusCycleRoot) {
            return xValue;
        }

        public Component getLastComponent(
                                Container
focusCycleRoot) {
            return yValue;
        }

        public Component getFirstComponent(
                                Container
focusCycleRoot) {
            return xValue;
        }
    }//end PictureExplorerFocusTraversalPolicy
inner class

}//end PictureExplorer class
```

**Listing 14. Source code for the program named Java362a.**

```
/*Program Java362a
Copyright R.G.Baldwin 2009

The purpose of this program is to support an explanation
of the PictureExplorer class.

A Picture object having dimensions of 450x345 pixels is
created. The the show method and the explore method are
called on the object to produce two different screen
displays of the picture.

The explore method simply creates a new object of the
PictureExplorer class.

Tested using Windows Vista Premium Home edition and
Ericso's multimedia library.
**************************************************/

public class Main{
  public static void main(String[] args){
    //Construct a new 460x345 Picture object.
```

```
    Picture pix1 = new Picture("ScaledBeach460x345.jpg");
    pix1.show();//display the picture in the show format
    //Display the picture again in the explore format.
    pix1.explore();
  }//end main method
}//end class Main
```

---

# Copyright



# About the author

**Richard Baldwin** *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is object-oriented programming using Java and other OOP languages.*

*Richard has participated in numerous consulting projects and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which have gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.*

*In addition to his programming expertise, Richard has many years of practical experience in Digital Signal Processing (DSP). His first job after he earned his Bachelor's degree was doing DSP in the Seismic Research Department of Texas Instruments. (TI is still a world leader in DSP.) In the following years, he applied his programming and DSP expertise to other interesting areas including sonar and underwater acoustics.*

*Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.*

*Baldwin@DickBaldwin.com*

-end-