An Introduction to Java Programming

Building a SlideShow Application

Acrobat version of this tutorial (1.9 MB)

Tutorial Contents

Back to top

An Introduction to Java Programming

Table of Contents Next Section

Overview

In this tutorial, we will be building a Java application which will display a series of images in succession, similar to a traditional slideshow. We will create custom image buttons and menu items that can be used to control the image display sequence. Other menu items will provide additional functionality such as image selection and optional preferences.

This tutorial is aimed at a broad audience, from beginning programmers who have little or no Java programming experience, to experienced programmers who are interested in learning Java. Experience with other programming languages is not required, but is useful because certain elementary programming concepts are assumed. Familiarity with object-oriented programming concepts and familiarity with the C programming language would be very helpful in order to make full use of the information presented in this tutorial.

Back to top

Table of Contents Next Section

An Introduction to Java Programming

Previous Section | Table of Contents | Next Section

What You Will Need For This Tutorial

There are several basic things that you will need in order to complete this tutorial:

- A PowerMacintosh Computer with 64 megabytes of RAM (96 recommended) running MacOS 8.1 or later (8.6 recommended) and 20 megabytes of free hard drive space;
- Macintosh Runtime for Java 2.1.1 or later;
- MRJ SDK 2.1 or later;
- A Java development environment. We recommend Metrowerks' Code Warrior. We will be using CodeWarrior 5 throughout this tutorial. More information is available on Metrowerks' web site;
- Stuffit Expander 5.0 or later to decompress the source code, and files associated with this tutorial. Stuffit Expander is freely available from Aladdin Systems' web site; and
- Tutorial sources and files (availiable from the Apple ftp Site)

This tutorial includes source files, preconfigured project files, resources, and text clippings that allow you to follow along with the instructions with a minimum of hassle. You will need these files in order to follow the steps outlined in these pages. You may download the tutorial files by following this link. If you do not yet have these files, please download them before proceeding to the next section.

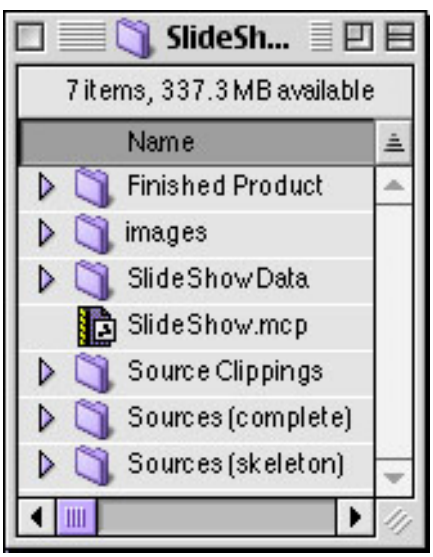
Back to top

Previous Section | Table of Contents | Next Section

An Introduction to Java Programming

Previous Section | Table of Contents | Next Section

Setting Up the Project



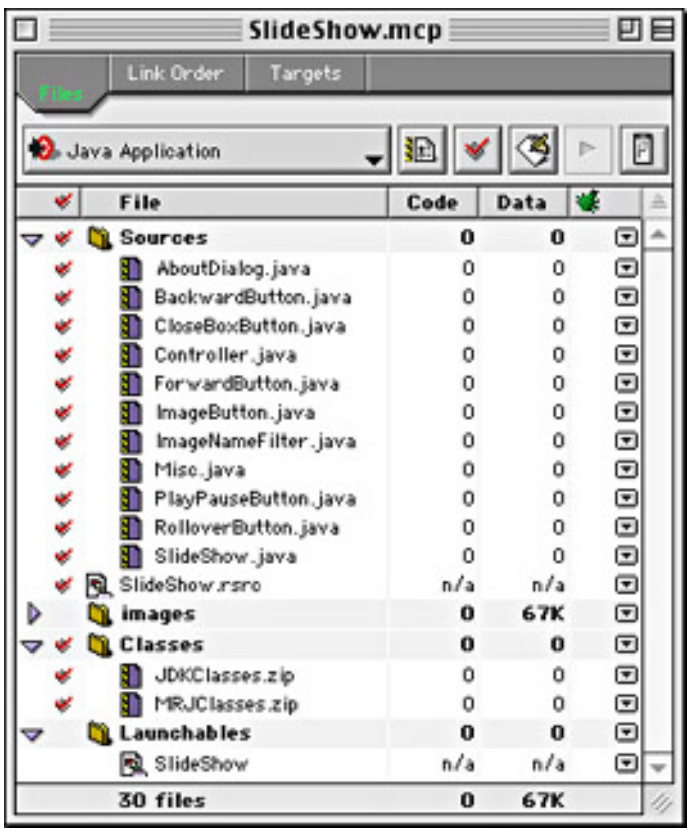
This tutorial uses a unique system that allows you to learn the concepts presented in this lesson without struggling with the frustration of coding errors caused by mistakes in typing or formatting.

As the picture (right) shows, the sources are organized logically in several folders:

- **Finished Product** - This folder contains the **.jar** file (**Java ARchive** file where the application classes and resources reside) and a pre-built version of the application. You may want to play around with the application a little to familiarize yourself with its operation.
- **images** - This folder contains all of the image resources (button images) used by the application.
- **SlideShowData** - This folder is used by CodeWarrior to store project information and temporary files. If this folder does not yet exist, it will be created the first time you compile your project, or change your project in some way.
- **SlideShow.mcp** - The Metrowerks CodeWarrior project file used by this tutorial. The project file contains information about build settings, as well as aliases to the files used to build the application.
- **Source Clippings** - This folder contains a number of sub-folders which contain text clippings (or code snippets) that will be inserted in the source code to complete methods. We will discuss how these files are used in more detail shortly.

- **Sources (complete)** - These are finished versions of each source file. These are provided for reference. If you run into compile errors, you may compare your version of the source files with these versions.
- **Sources (skeleton)** - This folder contains all of the "skeleton" source files for the application. The skeleton file contains method declarations and instructions and will be "fleshed out" into a completed version of the source file as we go through the tutorial. We will discuss this procedure shortly.

Now let's open the project in CodeWarrior and examine it in detail. If you have CodeWarrior 5, you may double-click directly on the project file "SlideShow.mcp". If you have an older version of CodeWarrior, you will need to use "SlideShow(CW4).mcp" instead.

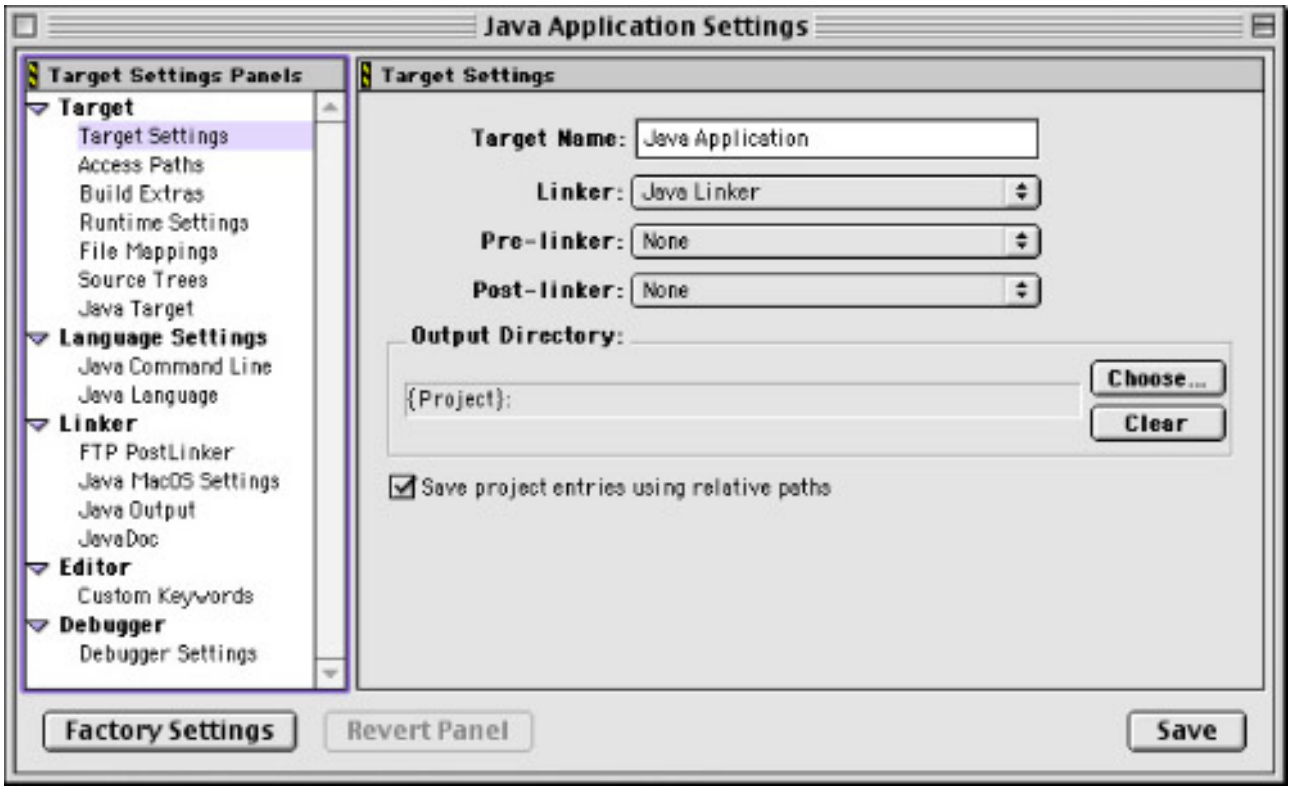


When you open the project, your project should look something like the picture (left). We have organized the project so that all of the sources are contained in a group called **Sources**.

All of the image resources are in a group called **images**, and library files are in a group called **Classes.**

Before we start examining the source code, we will examine the project settings. Although this project is preconfigured for your convenience, we will examine the pertinent settings that would need to be configured if you were writing a Java application from scratch.

To bring up the project settings dialog, either click on the project settings button:  or click on the Targets tab at the top of the window and then double-click on the line that reads **Java Application**.

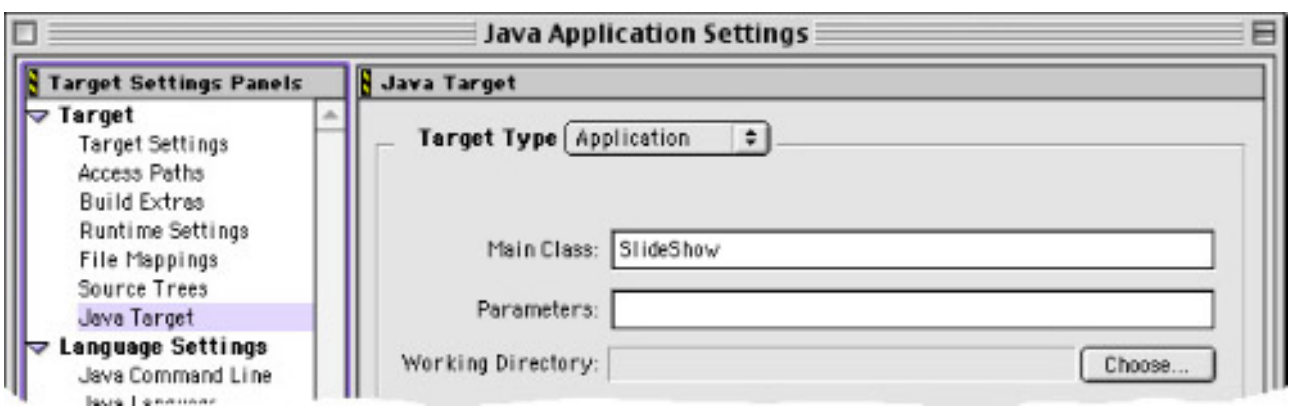


The settings dialog should look like the picture above. If it does not, click on the **Target Settings** item in the left panel. Go to the edit field labeled **Target Name**, and change the text to “SlideShow”. This specifies the name of the output file.

Make sure the **Linker** popup field reads “Java Linker”. CodeWarrior supports many different development languages, and Java is not the default, so we need to make sure that we are using the Java tools to build and link our project.

The **Pre-linker** and **Post-linker** popup menus should be set to “none”.

Now click on the **Java Target** item in the left pane. Your dialog should now look like this:



The **Target Type** popup menu has three possible values. “Library”, “Applet”, and “Application”. Since our project is a stand-alone program, we choose “Application”. If we wanted our program to live in an HTML file inside of a browser, then we would choose “Applet”. We would choose “Library” if we wanted to make a file that contained some Java routines that we wanted to call from another source code base.

Make sure that the **Main Class** text field contains the value “SlideShow”. This specifies that the `main()` routine of the application is in the class `SlideShow` (contained in SlideShow.java).

Next, click on the **Java Output** item in the left panel. Your dialog should look like this:

Make sure that “Jar File” is selected from the **Output Type** popup menu. Make sure that the **Name** field contains the entry “SlideShow.jar”. The jar format is a compressed format (if you check the Compress checkbox) similar to the .zip file format. It is a storage or archive format that accommodates a virtual hierarchical file structure that may contain both class files and resource files. Since our application has both of these file types, we use this format so that we can place all of our class files and resources in a single file on disk.

Now that we have completed these settings, click the **Save** button and close the window by clicking on the close box in the upper left hand corner of the window.

Our application uses classes from the MRJToolkit, and also classes from Sun’s built in Java Class Libraries. These library files are in our project in the classes folder: MRJClasses.zip and JDKClasses.zip. If you are starting a project from scratch, or if you do not have them in your project, you will need to add these files manually. To do so, select **Add Files** from the **Project** menu. In the standard file dialog, navigate to your MRJClasses folder (in the MRJ Libraries folder in the Extensions Folder of your active System Folder) and add JDKClasses.zip and MRJClasses.zip to your project. This step is vital. If you do not add these classes, your project will not compile.

We can now start looking at how the files are organized. (You may need to click on the **Files** tab to get back to your list of files).

Back to top

Previous Section | Table of Contents | Next Section

An Introduction to Java Programming

Previous Section | Table of Contents | Next Section

Architecture of the SlideShow Application

The SlideShow project contains 11 source files:

1. **AboutDialog.java** - contains the code for creating and displaying the application about box, visible when the user selects **About SlideShow** from the Apple Menu.
2. **BackwardButton.java** - based on `RolloverButton`, this file contains code for behavior unique to the backwards button.
3. **CloseBoxButton.java** - based on `RolloverButton`, this file contains code for behavior unique to the close box in the controller floating palette.
4. **Controller.java** - contains the code for creating, displaying, and handling events associated with the controller floating palette and its associate buttons, the forward button, backward button, play/pause button, and the close box button.
5. **ForwardButton.java** - based on `RolloverButton`, this file contains code for behavior unique to the forward button.
6. **ImageButton.java** - the base class for all of the button objects, this source implements common button behavior such as the ability to load and display images in the button.
7. **ImageNameFilter.java** - this source file contains code for filtering non-image files from the open file dialog.
8. **Misc.java** - contains miscellaneous routines for loading images.
9. **PlayPauseButton.java** - based on `RolloverButton`, this file contains code for behavior unique to the play/pause button.
10. **RolloverButton.java** - based on `ImageButton`, this file contains code for extending the `ImageButton` class to handle multiple image states in response to user interaction.
11. **SlideShow.java** - the main application class, this file contains code for displaying the slideshow window, creating and maintaining menu items, opening image files, and responding to user interaction.

As you can see from this brief synopsis of the source files involved, there is quite a bit of

functionality in such a “simple” application. In order to make this tutorial easier to follow and understand, we have broken the implementation of these classes into separate HTML files. Each HTML file contains a series of steps which explains the code that will be added to the source skeleton in order to implement all of the necessary functionality provided by the class.

Each source file in the project has a folder in the **Source Clippings** folder. For example, the first file, AboutDialog.java, has a corresponding folder called **AboutDialog**. As the picture left illustrates, this folder contains a number of text clippings. These clippings will be dragged into the source file at predetermined locations in order to “flesh out” a specific method or add additional code.

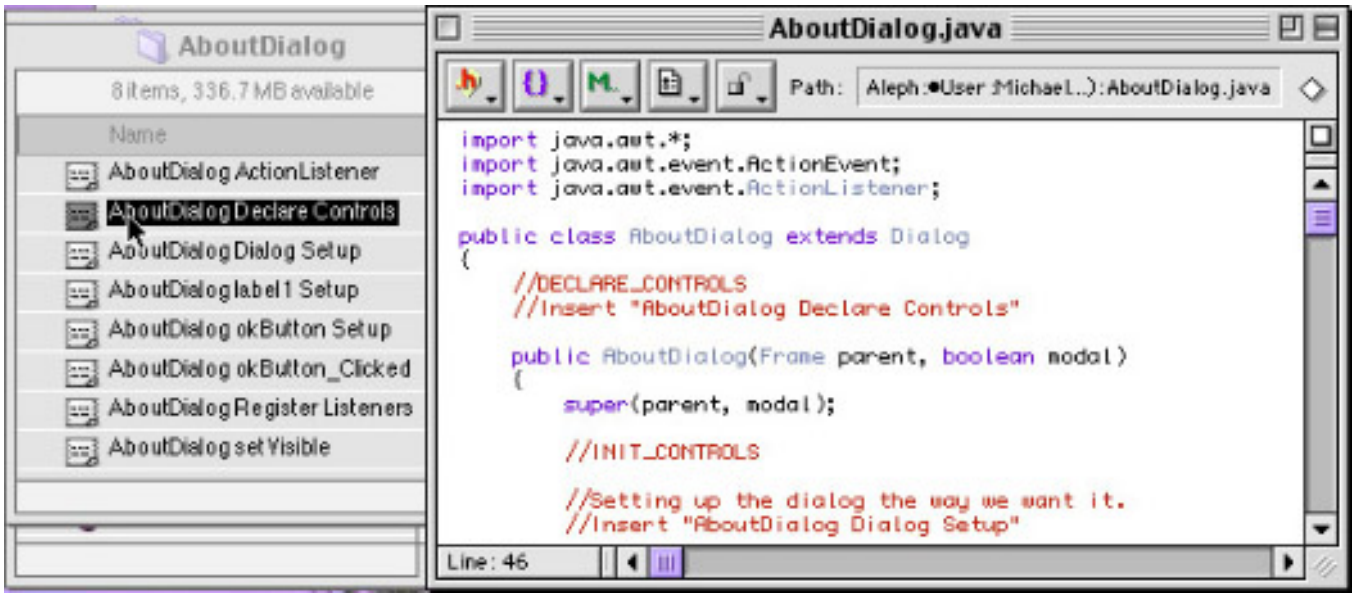
Each clipping is named in a manner that summarizes the functionality of that particular code snippet. For example, **AboutDialog Setup** indicates that the code snippet is used to setup the dialog. For clarity, all snippets will start with the name of the source file they belong to.

Throughout this tutorial, we will be specific about which source clipping should be used, and where it should be placed in the code. When there is a section of code near an area of code that needs an added text clipping, we will use the following format throughout the tutorial:

```
/**
 * This method does something
 */
 void foo ( int x )
 {
    // comment that tells the user which clipping to insert
    // i.e., insert myClass foo
```

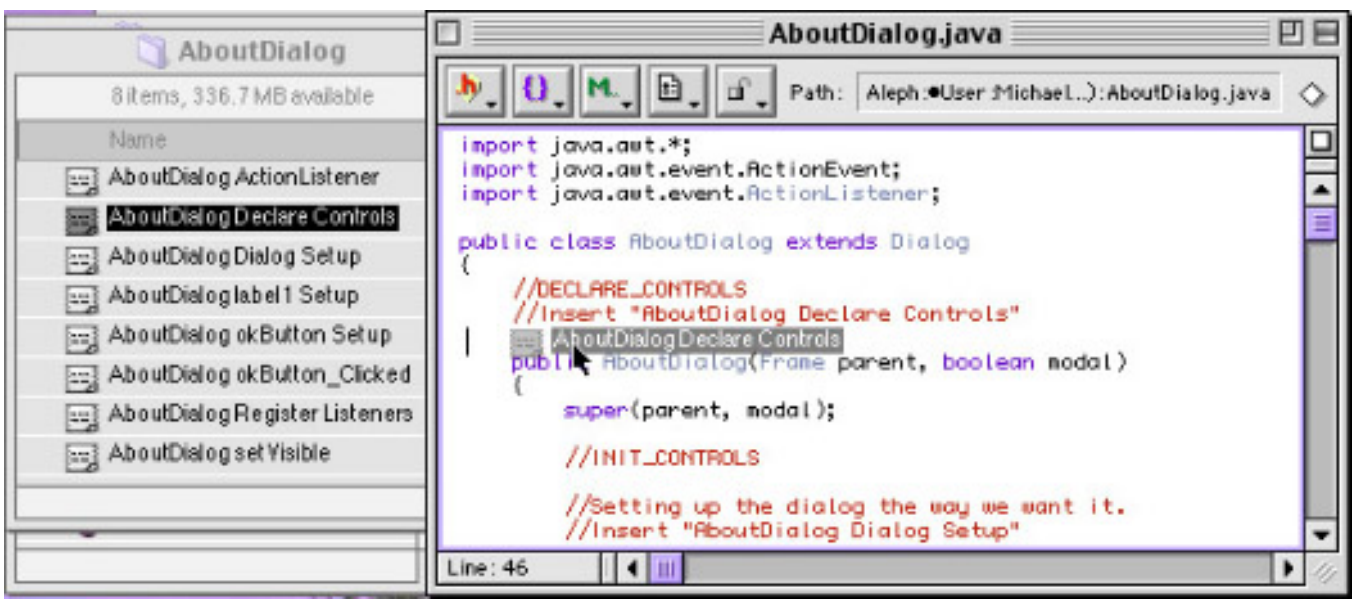
Note that the top area is in a light blue gray color. This region contains the code preceding the area where the clipping will be inserted.

The next area is a light yellow color. This shows the comment in the source that indicates the clipping to be used. The specific clipping should be inserted on the line **immediately following** this comment.



We recommend that you arrange your source window and the clipping window in the Finder so that you can see both simultaneously. This will facilitate dragging. See the picture above for an example.

With the source window as the front most window, click on the clipping to be dragged into the source file, and drag the file to the source window.



You will see an I-Beam cursor indicating where the clipping will be placed (see picture above). Make sure that you place the clipping precisely. Poor placement may result in compile errors. Frequently, there will be a blank line immediately following the comment where the clipping goes. Be careful to place the clipping before any trailing closing brace character “}”.

In the tutorial file, a section will show the source after a successful drag operation. Make sure that your source matches this block.

```
/**
 * This method does something
 */
 void foo ( int x )
 {
    // comment that tells the user which clipping to insert
    // ie, insert myClass foo
    System.out.println( "x: " + x );
}
```

The top section (in light blue) is shows the contents of the skeleton file. The darker blue area shows the contents of the newly added text clipping. This color scheme makes it easy to see what code preexists in the skeleton, and what code is added by the clipping.

Now that we have described the process of creating the complete source file using the skeleton file and the clipping, let's start building the project!

Back to top

Previous Section Table of Contents Next Section

The Apple Store | Hot News | Products | Design & Publishing | Developer | About Apple | Support | Education | Where to Buy

Search Shortcut

Search Tips | Site Map Extended Index

The Apple Store | Hot News | About Apple | Products | Support
Design & Publishing | Education | Developer | Where to Buy | Home

Contact Us - Developer Site Map

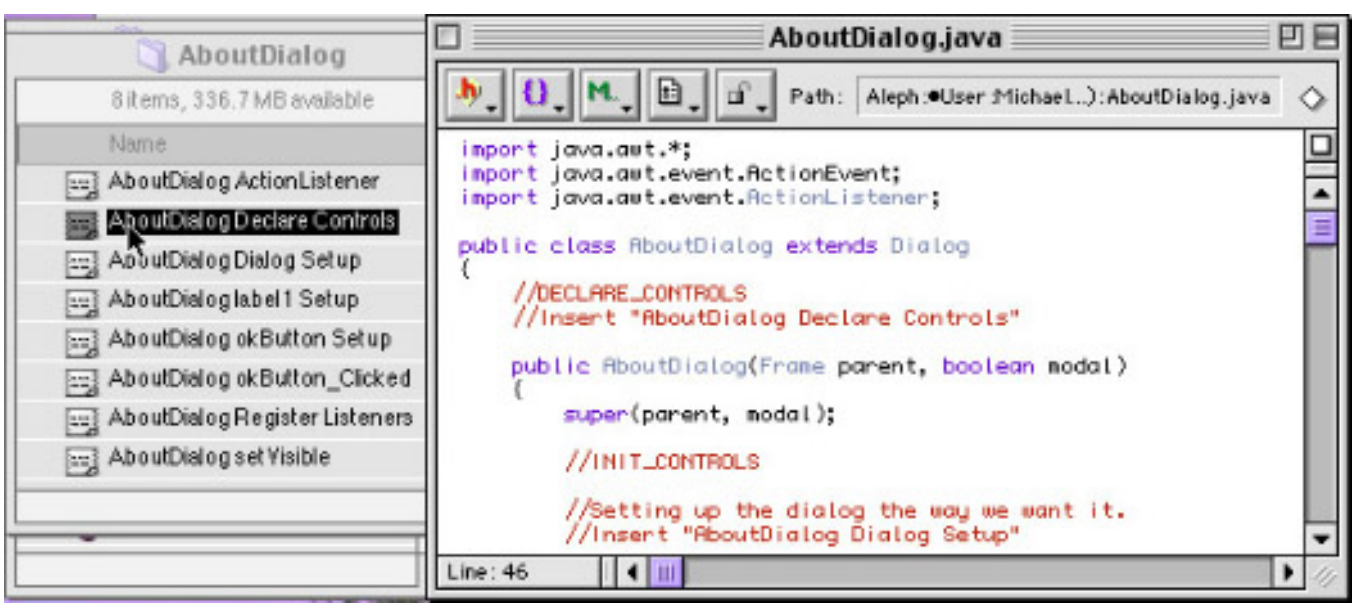
An Introduction to Java Programming

Previous Section | Table of Contents | Next Section

Step 1 - Building the About Box

The `AboutBox` is a very simple class that presents the user with information about the application. Before we get started, locate the **AboutDialog** folder in the **Source Clippings** folder. Open the **AboutDialog** folder, and position it so that the entire contents are visible when you are in CodeWarrior. You may wish to close other Finder windows to avoid confusion.

Now open the **AboutDialog.java** skeleton file by double-clicking on the **AboutDialog.java** item in the project window of CodeWarrior. Your layout should look something like the image below:



Now you are ready to start the source modifications in the section Building the About Dialog.

Once you complete these steps, close the source file and clipping folder before proceeding to the next section, Building the ImageButton.

Back to top

Previous Section Table of Contents Next Section

Building the About Dialog

File: AboutDialog.java

Contents

Overview

This file creates a dialog which is made visible when the user selects the **About SlideShow…** item from the Apple Menu. This class is a subclass of java.awt.Dialog, and registers a listener to dismiss the dialog when the OK button is pressed.

This file has two methods. The first is a constructor which specifies the dialog size, position, creates the OK button and the label, and other properties. The second is the `setVisible( )` method which is called to change the state of the dialog's visibility.

Steps to Follow

Step 1 - Declare the dialog controls

At the top of the file, we import packages we will use in this file (in this case, for simplicity, we import the entire java.awt package, and a couple classes we need for event handling), and declare our about dialog class.

Importing packages and classes allows us to abbreviate class names later on in the file. For instance, since we imported the entire java.awt package, when we make reference to classes in that package we do not need to specify the fully qualified package name. Instead, we can simply refer to the class by its immediate name. Thus, when we declare a `java.awt.Label` object, we only need to specify `label` as the class name. One might ask why not import all the packages all the time so anything that might be needed would be available. Importing a lot of files slows down the compiler since it needs to search for each class referred to in a large list. So then, why not import each class needed explicitly? This tends to make the top of the file unsightly and unnecessarily complex. Deciding when to import an entire package versus a collection of classes from a package is a judgement call. A good rule of thumb is if you are importing four or more classes from one package, go ahead and import the package instead.

```
import java.awt.*;
Import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class AboutDialog extends Dialog
{
    //DECLARE_CONTROLS
    //Insert "AboutDialog Declare Controls"
```

Locate the AboutDialog Declare Controls clipping in the AboutDialog folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
import java.awt.*;
Import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class AboutDialog extends Dialog
{
    //DECLARE_CONTROLS
    //Insert "AboutDialog Declare Controls"
```

```
    Label label1;
    Button okButton;
```

We have now declared two variables. The `label1` variable represents a `Label` component, and the `okButton` variable represents a `Button` component.

Back to top

Step 2 - Setting up the dialog

We now define the constructor for the `AboutDialog` class. This constructor takes two parameters, a `Frame` object which is the creator of the dialog, and a Boolean which specifies whether the dialog is modal or not. We pass these parameters off to the superclass (java.awt.Dialog) constructor so that we can take advantage of the default behavior of the dialog class.

```
public AboutDialog(Frame parent, Boolean modal)
{
    super(parent, modal);
    //INIT_CONTROLS
    //Setting up the dialog the way we want it.
    //Insert "AboutDialog Dialog Setup"
```

Now we are ready to set up the dialog. Locate the AboutDialog Dialog Setup clipping in the AboutDialog folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public AboutDialog(Frame parent, Boolean modal)
{
    super(parent, modal);
    //INIT_CONTROLS

    //Setting up the dialog the way we want it.
    //Insert "AboutDialog Dialog Setup"
    GridBagLayout gridBagLayout;
    gridBagLayout = new GridBagLayout( );
    setLayout(gridBagLayout);
    setVisible(false);
    setSize(277,100);
    setBackground(new Color(15724527));
    setTitle("About...");
    setResizable(false);
```

The new dialog setup code creates a new `GridBagLayout` layout manager. A layout manager is a class that is responsible for the placement of objects in a container. `GridBagLayout` is one of the most flexible layout managers, but its flexibility comes at the price of complexity. For the purpose of this tutorial, we will not be examining `GridBagLayout` in detail. Please

see the JavaSoft web site for a tutorial on GridBagLayout.

Once the layout manager is created, `setVisible(false)` is called to ensure the dialog is not initially visible. The dialog is set to the required size, a light gray background color is specified, the title is specified, and the dialog is made non-resizable, as a matter of personal preference.

Back to top

Step 3 - Setting up the label and placing it in the layout

Now that we have specified the basic properties of the dialog, we are ready to create the label and define its characteristics.

```
    setTitle("About...");
    setResizable(false);
    //Setting up label1 and placing it in the layout
    //Insert "AboutDialog label1 Setup"
```

Locate the AboutDialog label1 Setup clipping in the AboutDialog folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
    setTitle("About...");
    setResizable(false);

    //Setting up label1 and placing it in the layout
    //Insert "AboutDialog label1 Setup"
    label1 = new Label("This is my cool SlideShow
                        Application!",Label.CENTER);
    GridBagConstraints gbc;
    gbc = new GridBagConstraints( );
    gbc.gridx = 1;
    gbc.gridy = 1;
    gbc.fill = GridBagConstraints.NONE;
    gbc.insets = new Insets(0,0,0,0);
    ((GridBagLayout)getLayout( )).setConstraints(label1, gbc);
    add(label1);
```

The first step is to create a new `java.awt.Label` object and assign it to the `label1` variable we previously declared. We pass the `Label` constructor the text to display and specify “Label.CENTER” as the horizontal alignment. This will cause the label to be drawn centered within its bounds.

We now set up the GridBagConstraints and add the label to the dialog.

Back to top

Step 4 - Setting up the “OK” button and placing it in the layout

The next item to be added is the `okButton`.

```
    ((GridBagLayout)getLayout( )).setConstraints(label1, gbc);
    add(label1);
    //Setting up okButton and placing it in the layout
    //Insert "AboutDialog okButton Setup"
```

Locate the AboutDialog okButton Setup clipping in the AboutDialog folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
    ((GridBagLayout)getLayout( )).setConstraints(label1, gbc);
    add(label1);

    //Setting up okButton and placing it in the layout
    //Insert "AboutDialog okButton Setup"
    okButton = new Button( );
okButton.setLabel("OK");
    gbc = new GridBagConstraints( );
    gbc.gridx = 1;
    gbc.gridy = 2;
    gbc.fill = GridBagConstraints.NONE;
    gbc.insets = new Insets(0,0,0,0);
    ((GridBagLayout)getLayout( )).setConstraints(okButton, gbc);
    add(okButton);
```

The first step is to create a new instance of class java.awt.Button and assign it to our `okButton` variable we previously declared. We set the label of the button to "OK", and set up the GridBagConstraints. Lastly, we add the button to the dialog.

Back to top

Step 5 - Responding to button clicks from the okButton

Now that we have an OK button, we need to create a method that will respond to the button press and hide the `AboutDialog`. Skip down in the source file past the `setVisible( )` method.

```
Public void setVisible(Boolean b)
{
    //Place the dialog in the Macintosh Alert Position
    //Insert "AboutDialog setVisible"
}

//Innerclass for handling ActionEvents
//Insert "AboutDialog ActionListener"
//Respond to button clicked ActionEvents from the okButton
//Insert "AboutDialog okButton_Clicked"
```

Locate the AboutDialog okButton_Clicked clipping in the AboutDialog folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public void setVisible(Boolean b)
{
    //Place the dialog in the Macintosh Alert Position
    //Insert "AboutDialog setVisible"
}

//Innerclass for handling ActionEvents
//Insert "AboutDialog ActionListener"

/Respond to button clicked ActionEvents from the okButton
//Insert "AboutDialog okButton_Clicked"
void okButton_Clicked(ActionEvent event)
{
    setVisible(false);
}
```

Here we are creating a method that takes an action event parameter and does not return anything. The `ActionEvent` will be broadcast from the button when the button is clicked. This method hides the dialog by calling `setVisible( )` with `false` as the parameter.

Back to top

Step 6 - Creating an inner class to handle action events

We have an `okButton_Clicked( )` method that knows how to behave appropriately when the “OK” Button is clicked. Now we need a mechanism that responds to the button press and calls our method. When the Button is pressed, it generates an `ActionEvent`. We need to create an inner class which will listen for this `ActionEvent` and call our `okButton_Clicked( )` method to hide the dialog. Go back up in the source file to the comment immediately following the `setVisible( )` method.

```
Public void setVisible(Boolean b)
{
    //Place the dialog in the Macintosh Alert Position
    //Insert "AboutDialog setVisible"
}
//Inner class for handling ActionEvents
//Insert "AboutDialog ActionListener"
```

Locate the AboutDialog ActionListener clipping in the AboutDialog folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public void setVisible(Boolean b)
{
    //Place the dialog in the Macintosh Alert Position
    //Insert "AboutDialog setVisible"
}

//Innerclass for handling ActionEvents
//Insert "AboutDialog ActionListener"
class Action implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        okButton_Clicked(event);
    }
}
```

This code may seem confusing at first, but it is really quite straightforward. We want to respond to the `ActionEvent` broadcast by the `okButton` object. Hence we create an inner class called `Action` which implements the `ActionListener` interface. The `ActionListener` interface defines a single `actionPerformed` method which we implement in our class. By implementing this method, we can respond to action performed events. Our `actionPerformed` method simply calls our `okButton_Clicked( )` method and passes the received event as the parameter.

In a nutshell, the Button keeps a list of Listeners who have registered with the Button that they wish to be notified when an `actionPerformed` event occurs. When an `actionPerformed` event occurs, the Button traverses its list of Listeners and notifies each one in turn that the event occurred. It subsequently calls the `actionPerformed` method of each listener with a new `ActionEvent` describing the details of the event.

For more information on event handling in JDK 1.1, see JavaSoft's Event Handling Tutorial.

Back to top

Step 7 - Registering our action listener

We have created an inner class that responds to `ActionEvents` by calling our `okButton_Clicked( )` method. Now we need to hook up our handler to the `okButton`.

Go up to the end of the code block we added in Step 4.

```
gbc.fill = GridBagConstraints.NONE;
gbc.insets = new Insets(0,0,0,0);
((GridBagLayout)getLayout( )).setConstraints(okButton, gbc);
add(okButton);
//REGISTER_LISTENERS
//Registering our ActionListener with the okButton
//Insert "AboutDialog Register Listeners"
```

Locate the AboutDialog Register Listeners clipping in the AboutDialog folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
gbc.fill = GridBagConstraints.NONE;
gbc.insets = new Insets(0,0,0,0);
((GridBagLayout)getLayout( )).setConstraints(okButton, gbc);
add(okButton);

//REGISTER_LISTENERS
//Registering our ActionListener with the okButton
//Insert "AboutDialog Register Listeners"
Action lAction = new Action( );
okButton.addActionListener(lAction);
```

Registering our `ActionListener` is fairly straightforward. We create an instance of our inner class, and call `addActionListener( )` from our button with our `Action` object as a parameter. Basically, we are telling the button that we have a class (our Action class) that is interested in receiving notification when `ActionEvents` occur. When the `okButton` is clicked, it checks its list of registered listeners, and sends the `Action` object an `ActionEvent`. The `Action` object processes the ActionEvent and calls `okButton_clicked( )` which hides the window.

Back to top

Step 8 - Implementing setVisible( )

There is one task remaining that we need to accomplish for this class. We want to override `setVisible( )` so that we can add centering behavior to our `AboutBox`.

```
/**
 * Shows or hides the component depending on the Boolean flag b.
 * @param b  if true, show the component; otherwise, hide the
 * component.
 * @see java.awt.Component#isVisible
 */
public void setVisible(Boolean b)
{
    //Place the dialog in the Macintosh Alert Position
    //Insert "AboutDialog setVisible"
}
```

Locate the AboutDialog setVisible clipping in the AboutDialog folder and drag it directly below the last line of code shown above in orange. Make sure that it precedes the closing brace of the function. Your code should now look like this:

```
/**
 * Shows or hides the component depending on the Boolean flag b.
 * @param b  if true, show the component; otherwise, hide the
 *    component.
 * @See java.awt.Component#isVisible
 */
public void setVisible(Boolean b)
{
    //Place the dialog in the Macintosh Alert Position
    //Insert "AboutDialog setVisible"
    if(b)
    {
        Dimension bounds  =
             Toolkit.getDefaultToolkit( ).getScreenSize( );
        Dimension abounds = getSize( );
        setLocation((bounds.width - abounds.width) / 2,
             (bounds.height - abounds.height) / 3);
    }
    super.setVisible(b);
}
```

This code snippet uses basic math to determine the center of the screen. It is within an `if( )` statement because we only want to do our computation if we are in the process of becoming visible. The first thing we do is get the bounds (height and width) of the screen. We do this via a utility class called the Toolkit. This class is part of the standard AWT.

Once we have the screen bounds, we get the size of the dialog and move the dialog so that it is centered horizontally, and placed at 1/3 of the screen height.

This completes the source modifications for About.java.

Back to top

Summary

There are several important concepts to be learned from this source file. We learned how to declare and initialize controls that appear in a dialog. We were introduced to event management in Java and learned how to respond to a button click. We also took a cursory look at layout components in a window, and learned how to register our event handlers. It is surprising how much we learned just from a simple About box.

Now we are ready to return to our main tutorial file where we will prepare our project for the next step, Building the ImageButton.

An Introduction to Java Programming

Previous Section | Table of Contents | Next Section

Step 2 - Building the ImageButton

The `ImageButton` class is the first of several classes that implement the button behavior used in all of our controls. This is the base class that contains basic behavior, such as the ability to load and display images.

If you have not already done so, close the AboutDialog window in the Finder, and open the **ImageButton** folder in the **Source Clippings** folder. You may need to resize and reposition the window so that all of the clippings are visible. Now open the **ImageButton.java** skeleton file by double-clicking on the corresponding file in the project window of CodeWarrior. You may need to rearrange your window so that you can see the **ImageButton** clipping folder in the Finder.

Now you are ready to start the source modifications in the section Building the Image Button.

Once you complete these steps, close the source file and clipping folder before proceeding to Step 3.

Back to top

Step 3 - Building the RolloverButton

The `RolloverButton` class extends the `ImageButton` class to provide multiple-state information within the button.

As we have done before, close the **ImageButton** folder and open the **RolloverButton** Source Clipping folder. Open the **RolloverButton.java** from the Project window before proceeding to the next set of steps in the section Building the RolloverButton.

Back to top

Step 4 - Building the Forward Button

The `ForwardButton` class extends the `RolloverButton` class. It customizes the behavior in that class in order to specify a unique set of images to be used for its display.

Before proceeding to the steps for the Forward Button, close any open source files, and open the **ForwardButton.java** source file and the **ForwardButton** Source Clipping folder in the Finder. Once again, you may need to resize or reposition your windows to make optimal use of your screen real estate. Once this preparation is completed, proceed to the steps in the section Building the Forward Button.

Back to top

Step 5 - Building the Backward Button

The `BackwardButton` class is very similar to the `ForwardButton` class, except that we specify a different series of image files.

Once again, close any open source files and open the **BackwardButton.java** skeleton file. Open the **BackwardButton** Source Clipping folder in the Finder.

After completing this step, proceed to the steps in the section Building the Backward Button.

Back to top

Step 6 - Building the Play/Pause Button

While related to the `BackwardButton` and `ForwardButton`, and also derived from `RolloverButton`, the `PlayPauseButton` class is slightly more complex. Since it is a two-state toggle button, it has some additional functionality to facilitate handing this additional state.

Before proceeding to the steps for the Play/Pause Button, close any open source files, and open the **PlayPauseButton.java** source file and the **PlayPauseButton** Source Clipping folder in the Finder. Once again, you may need to resize or reposition your windows to make optimal use of your screen real estate. Once this preparation is completed, proceed to the steps in the section Building the Play/Pause Button.

Back to top

Previous Section Table of Contents Next Section

Building the Image Button

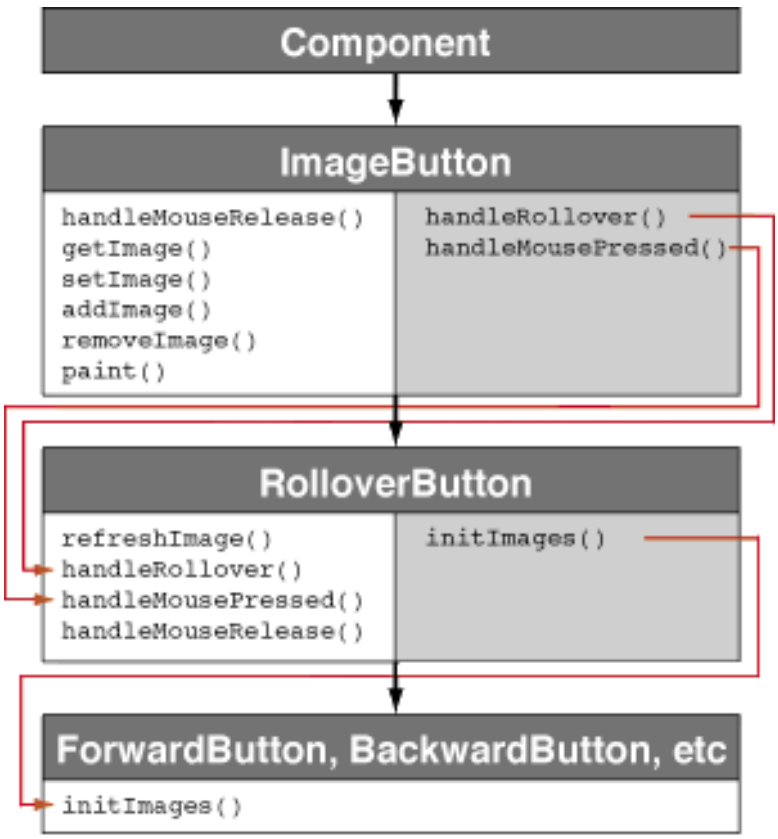
File: ImageButton.java

Contents

Overview

The `ImageButton` is the base class that provides core functionality for all of the buttons used in the controller.

The `ImageButton` class is derived from java.awt.Component (see diagram right).



It implements several methods that provide basic functionality such as retrieving an image, setting an image, removing an image and painting itself. It also responds to `MouseRelease` messages.

This class declares two abstract methods, `handleRollover( )` and `handleMousePressed( )`, which are implemented in the derived-class `RolloverButton`.

The `RolloverButton` class is responsible for swapping images when the button is clicked, and when the mouse is hovering over the button. For more information on this class, see Building the Rollover Button.

There is a third and final tier which consists of three classes that derive from `RolloverButton`: `ForwardButton`, `BackwardButton`, and `PlayPauseButton`. These classes are extremely simple and implement a single method that specifies which images to use for the button state. These classes are explained in more detail later.

This hierarchy allows us to better group related functionality together with common behaviors in the base class and more specific behaviors in the derived classes. This allows for a much cleaner and coherent API, and demonstrates the power of object-oriented programming.

Steps to Follow

Back to top

Step 1 - Declaring the Data Members

The `ImageButton` is an abstract class. That means that it cannot be directly instantiated. It specifies an interface of methods that derived classes must override in order to implement its functionality.

We start by importing the necessary packages, the awt package, the event package, and java.util.Hashtable.

The class is declared as a public, abstract class which derives from java.awt.Component.

```
import java.awt.*;
Import java.awt.event.*;
Import java.util.Hashtable;

public abstract class ImageButton extends Component
{
    //Declare data members
    //Insert "ImageButton data members"
```

Locate the ImageButton data members clipping in the ImageButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
import java.awt.*;
Import java.awt.event.*;
Import java.util.Hashtable;

public abstract class ImageButton extends Component
{
    //Declare data members
    //Insert "ImageButton data members"
    protected Hashtable imageHash;
    protected Image image;
    protected String imageName;
    protected Boolean isMouseDown = false;
    protected Boolean isMouseInside = false;
    protected String actionCommand;
    protected ActionListener actionListener = null;
```

We declare all of the data members as protected. This is because we do not want them to be accessible except by derived classes. The first data member is `imageHash`. We will use the hashtable to keep track of the button images. We chose to use a hashtable because we wanted to be able to store an arbitrary number of images and retrieve them by name. It is the responsibility of the derived class to swap images based on state or user interaction.

The second member, `image`, referrers to the currently displayed image. The variable `imageName` is a String representing the name of this image.

Next, we have some state information about the mouse. The members `isMouseDown` and `isMouseInside` allow us to keep track of where the mouse is located so that we can perform rollover actions correctly. These variables will primarily be used by the derived classes.

The final two members, `actionCommand` and `actionListener`, are used for responding to user interaction. We will examine this in more detail in Step 3 and Step 10.

Back to top

Step 2 - Handling Mouse Events

The main function of a button is to respond to user interaction such as a mouse press. In order to respond correctly to the mouse, we need to write an inner class for handling mouse events.

Scroll down to the very bottom of the source file where it reads:

```
public void paint(Graphics g)
{
    //Let the super class draw, then handle drawing the current image.
    //Insert "ImageButton paint"
}
//Inner class for handing mouse events.
//Insert "ImageButton Mouse Handling"
```

Locate the ImageButton Mouse Handling clipping in the ImageButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public void paint(Graphics g)
{
    //Let the super class draw, then handle drawing the current image.
    //Insert "ImageButton paint"
}

//Inner class for handing mouse events.
//Insert "ImageButton Mouse Handling"
class Mouse extends MouseAdapter
{
    public void mouseExited(MouseEvent event)
    {
        ImageButton_MouseExited(event);
    }

    public void mouseEntered(MouseEvent event)
    {
        ImageButton_MouseEntered(event);
    }

    public void mouseReleased(MouseEvent event)
    {
        ImageButton_MouseReleased(event);
    }

    public void mousePressed(MouseEvent event)
    {
        ImageButton_MousePressed(event);
    }
}

protected void ImageButton_MousePressed(MouseEvent event)
```

```
{
    isMouseDown = true;
    handleMousePressed( );
}

protected void ImageButton_MouseReleased(MouseEvent event)
{
    isMouseDown = false;
    handleMouseRelease(isMouseInside);
}

protected void ImageButton_MouseEntered(MouseEvent event)
{
    isMouseInside = true;
    handleRollover(isMouseInside, isMouseDown);
}

protected void ImageButton_MouseExited(MouseEvent event)
{
    isMouseInside = false;
    handleRollover(isMouseInside, isMouseDown);
}
```

Wow! That's a lot of code. Don't worry. It is pretty straightforward. Let's look at it in more manageable sections.

First we declare a subclass of `MouseAdapter` called `Mouse`.

```
class Mouse extends MouseAdapter {
```

`MouseAdapter`, in the java.awt.event package, is an abstract class that is provided as a convenience for easily creating listeners. Here, we override the class and implement the methods we are interested in: `mouseEntered`, `mouseExited`, `mousePressed`, and `mouseReleased`. These methods will be called when a certain type of `MouseEvent` occurs. When the user moves the mouse over the `ImageButton`, the `mouseEntered( )` routine will be called. When the user moves the mouse outside of the `ImageButton`, the `mouseExited( )` routine will be called. Similarly, `mousePressed( )` and `mouseReleased( )` are called when the mouse button is pressed and when the mouse button is released, respectively.

```
public void mouseExited(MouseEvent event)
{
    ImageButton_MouseExited(event);
}
```

Each of these methods is defined in a similar fashion. The event that is received is passed off to a subordinate function. This is done for convenience. It makes the secondary method easier to override since it is not located inside the inner class.

The `ImageButton_MousePressed( )` method is very simple:

```
protected void ImageButton_MousePressed(MouseEvent event)
{
    isMouseDown = true;
    handleMousePressed( );
}
```

It sets the isMouseDown data member to true and calls `handleMousePressed( )`. Remember that `handleMousePressed( )` is defined in this class as an abstract method and is overridden in RolloverButton. Thus, when the mouse button is pressed, it calls the method in the `RolloverButton` that provides the implementation. As a result, we handle the event in our low-level class, but respond to the user in our derived class.

`ImageButton_MouseReleased( )`, `ImageButton_MouseEntered( )`, and `ImageButton_MouseExited( )` are very similar. They pass state information to the derived class via the abstract methods that are defined in the derived class. `ImageButton_MouseReleased( )` is an exception in that it calls `handleMouseRelease( )`, which is the only non-abstract mouse handling routine. We will look at this method in more detail in Step 4.

Now it is time to go back up to the top of the file and look at the constructor where we register our listener we just created.

Next page

Previous document

Step 3 - Registering the Action Listener

Now that we have methods that can respond to mouse events, we need to register our listener with the `ImageButton` class. This is done in the constructor.

```
public ImageButton( )
{
    //REGISTER_LISTENERS
    //Insert "ImageButton register listener
```

Locate the ImageButton register listener clipping in the ImageButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public ImageButton( )
{
    //REGISTER_LISTENERS
    //Insert "ImageButton register listener
    Mouse aMouse = new Mouse( );
    this.addMouseListener(aMouse);
```

First, we create a new instance of our `Mouse` inner class we defined in Step 2. Secondly, we register the `Mouse` class as a listener for the `ImageButton`. Now, when the user performs any mouse movement relating to the `ImageButton`, the `Mouse` class will be called to respond to the generated event.

To complete our constructor, we have some additional initialization to perform:

```
    Mouse aMouse = new Mouse( );
    this.addMouseListener(aMouse);
    //Initialize state information
    //Insert "ImageButton init state"
```

Locate the ImageButton init state clipping in the ImageButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
    Mouse aMouse = new Mouse( );
    this.addMouseListener(aMouse);

    //Initialize state information
    //Insert "ImageButton init state"
    imageHash = new Hashtable( );
    actionCommand = "ImageButton Action";
```

We allocate a new hashtable to contain the button images, and then we initialize our action command string. The action command string will allow objects which receive the action event from our button to determine the source of the message.

Back to top

Step 4 - Handling MouseReleased Messages

We have defined our inner class that handles mouse events and registers that class as a `mouseListener` for the button. Now it is time to start implementing the methods.

```
/**
 * Gets called when the mouse button is pressed on this button.
 * @param isMouseInside, if true, the mouse is located inside
 * the button area, if false the mouse is outside the button
 * area.
 */
protected void handleMouseRelease(Boolean isMouseInside)
{
    //Handle firing an ActionEvent to our listeners if the
    //mouse was released inside the button.
    //Insert "ImageButton handleMouseReleased"
```

As you can see from the JavaDoc, the `handleMouseRelease( )` method gets called when the user presses the mouse button on this button and then releases it. We explored the mechanism for propagating this message in Step 2. We take a Boolean parameter that lets us know if the mouse was inside the button when it was released.

Locate the ImageButton handleMouseReleased clipping in the ImageButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Gets called when the mouse button is pressed on this button.
 * @param isMouseInside, if true, the mouse is located inside
 * the button area, if false the mouse is outside the button
 * area.
 */
 protected void handleMouseRelease(Boolean isMouseInside)
 {
     //Handle firing an ActionEvent to our listeners if the
     //mouse was released inside the button.
     //Insert "ImageButton handleMouseReleased"
     if (isMouseInside)
          fireActionEvent( );
}
```

We check to see if the mouse was still inside the button when it was released. The Boolean `isMouseInside` is passed to us by `ImageButton_MouseReleased( )` from Step 2. If the mouse is not inside, we don't do anything. Otherwise, we call `fireActionEvent( )`, which creates a new action event and notifies any registered listeners of the event. We will talk about this function in more detail in Step 10. For now, it is only important to know that this function will notify other slideshow components that the button has been pressed so that they have a chance to respond to this action.

Back to top

Step 5 - Implementing addImage( )

Skipping down past the abstract declarations of `handleRollover( )` and `handleMousePressed( )`, which are implemented in `RolloverButton`, we come to the declaration of `addImage`:

```
/**
 * Adds an image to the button.
 * @param imagePath, the location of the image resource to use.
 * This path is relative to the location of this class file.
 * @param imageName, the name used to identify the image for
 * later use in this button.
 * @see #removeImage
 */
 public void addImage(String imagePath, String imageName)
 {
    //Handle storing the information in our internal data
    //structure.
    //Insert "ImageButton addImage"
```

`Addimage` is used to add an image to the button's list of usable images. It takes an `imagePath` as a string which is a location and name of the image file to use relative to the application resources, and a string that specifies the name of the image. This is not the filename. It is used to internally refer to that particular image.

Locate the ImageButton addImage clipping in the ImageButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Adds an image to the button.
 * @param imagePath, the location of the image resource to use.
 * This path is relative to the location of this class file.
 * @param imageName, the name used to identify the image for
 * later use in this button.
 * @see #removeImage
 */
 public void addImage(String imagePath, String imageName)
 {
    //Handle storing the information in our internal data
    //structure.
    //Insert "ImageButton addImage"
    if (imageName != null && !imageName.equals(""))
    {
        Image newImage = Misc.loadImage(imagePath, this, true);
        if (newImage != null)
        {
            imageHash.put(imageName, newImage);
        }
     }
}
```

This method checks the `imageName` to make sure that it is neither null, nor empty. Since we are going to store the image in a hashtable and use the name as a key, the name must not be null and it must be non-empty. If the `imageName` does not meet these criteria, we exit the function (drop out of the `if` statement). Otherwise, we load the image using a supplementary routine from the `Misc` class and store it in a temporary variable. The `Misc` class has a single routine that loads images and does error handling. Its function is outside the scope of this tutorial, but we felt it was important to include a reasonably robust mechanism for loading resources that you may use in your own projects.

If the image was loaded successfully (i.e., the image loaded is not null), we add the item to our hashtable, using the image name as the key and the image as the data. What is a hashtable? A hashtable is a data structure that allows you to store data in several storage slots retrievable by a key. The key is used to determine which slot the item is stored in. It is a very fast and efficient storage mechanism which is built-in to java.

Now that we have a mechanism for adding images to our pool of button images, we need to be able to remove them.

Back to top

Step 6 - Implementing removeImage( )

The `removeImage` function can be used to remove unwanted images from the button image pool, or for cleanup purposes.

```
/**
 * Removes an image from the button
 * @param imageName, the identifying name of the image to remove.
 * @see #addImage
 */
 public void removeImage(String imageName)
 {
     //Handle removing the image from our internal data
     //structure.
     //Insert "ImageButton removeImage"
```

This method only takes a string as a parameter. It takes the `imageName`, looks it up in the hashtable, and deletes the item if it is found.

Locate the ImageButton removeImage clipping in the ImageButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Removes an image from the button
 * @param imageName, the identifying name of the image to remove.
 * @see #addImage
 */
 public void removeImage(String imageName)
 {
     //Handle removing the image from our internal data
     //structure.
     //Insert "ImageButton removeImage"
     if (imageName != null && !imageName.equals(""))
     {
         imageHash.remove(imageName);
     }
}
```

The body of this method is fairly simple. We check to see if the name passed to the function is non-empty and non-null, and then call `remove` from the hashtable with the image name as the parameter. Now it's time to look at `setImage( )`.

Back to top

Step 7 - Implementing setImage( )

The routine `setImage( )` is used to change the image displayed in the button to a specific image that has been added to the collection of button images.

```
/**
 * Sets the image for the button to use as its current image.
 * @param imageName, the identifying name of the image to use.
 */
 public void setImage(String imageName)
 {
     //Handle locating the image in our internal data structure,
     //setting it as the current image, and repainting the
     //button.
     //Insert "ImageButton setImage"
```

Locate the ImageButton setImage clipping in the ImageButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
//**
 * Sets the image for the button to use as its current image.
 * @param imageName, the identifying name of the image to use.
 */
 public void setImage(String imageName)
 {
     //Handle locating the image in our internal data structure,
     //setting it as the current image, and repainting the
     //button.
     //Insert "ImageButton setImage"
     if (imageName != null && !imageName.equals(""))
     {
         Image temp = (Image)imageHash.get(imageName);
         if (temp != null)
         {
              image = temp;
              this.imageName = imageName;
              repaint( );
         }
     }
}
```

`SetImage( )` seems a little more difficult on the surface than `removeImage( )`, but it is really not. We check to make sure that the image name is neither null nor empty, and then retrieve the current image from the hashtable, storing it in the temporary variable temp. After checking to make sure that the retrieved image is not null, we set our image data member to the retrieved image. At first glance, this may seem strange. Why are we using a temporary variable in the first place? Why couldn't we write:

```
image = (Image)imageHash.get(imageName);
```

and then check to see if image is null? Well then if the image we were loading did not exist, we would have no idea what the image variable previously contained, and our current image would be null. This would be a bad idea. So we retrieve the image into a temporary variable, and then if it is valid, set the current image variable to the temporary. Then we store the image name:

```
this.imageName = imageName;
```

What's up with the `this.imageName`? Well, you may note that the parameter of this routine is called `imageName`. Since we want to set the value of the `ImageButton` data member `imageName` to the local routine parameter imageName, we use `this.imageName` to specify class scope for the variable instead of local scope.

Last but not least, we call `repaint( )`, a java.awt.Component method that redraws the image button and displays our new image. Whew! Now it's time for the trivial `getImage( )` method.

Back to top

Step 8 - Implementing getImage( )

This method quite simply returns the name of the current image.

```
/**
 * Gets the name of the image currently in use.
 * @return The identifying name of the image being used.
 */
 public String getImage( )
 {
     //Return the current image name.
     //Insert "ImageButton getImage"
```

Locate the ImageButton getImage clipping in the ImageButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Gets the name of the image currently in use.
 * @return The identifying name of the image being used.
 */
 public String getImage( )
 {
     //Return the current image name.
     //Insert "ImageButton getImage"

     return imageName;
}
```

It really doesn't get much easier than this. We simply return our current image name stored in the image button data member `imageName`. Next is the very similar function `getImageObject( )`.

Back to top

Step 9 - Implementing getImageObject( )

This method returns the actual image object associated with the current button image, not just the name.

```
/**
 * Gets the actual Image Object which is currently being used.
 * @return The java.awt.Image currently in use.
 */
 public Image getImageObject( )
 {
     //Return the current image object.
     //Insert "ImageButton getImageObject"
```

Locate the ImageButton getImageObject clipping in the ImageButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Gets the actual Image Object which is currently being used.
 * @return The java.awt.Image currently in use.
 */
 public Image getImageObject( )
 {
     //Return the current image object.
     //Insert "ImageButton getImageObject"

     return image;
```

```
}
```

This should come as no surprise. We simply return our current image stored in our image data member of `ImageButton`. Now that we can add, remove, set and get button images, it is time to implement some routines for responding to button actions.

Back to top

Step 10 - Handling Action Events

As we recall from Step 2 and Step 3, there is a very specific chain of events that occur when the user clicks on the button. The first thing that happens is our `MouseHandler` inner class gets called along with the appropriate `MouseEvent`. In the case of a mouse click, our `mousePressed( )` routine gets called followed by `mouseReleased( )`. If the mouse is still inside of the button when it is released, we call `fireActionEvent( )`. This sends messages to other components (that are registered as listeners for the button) to notify them that the button was activated.

```
public Image getImageObject( )
{
    //Return the current image object.
    //Insert "ImageButton getImageObject"
    return image;
}
//Routines for handling ActionListener management.
//Insert "ImageButton Action Management"
```

Let's look at the mechanism for action management. Locate the ImageButton Action Management clipping in the ImageButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
//Routines for handling ActionListener management.
//Insert "ImageButton Action Management"
```

```
/**
 * Sets the command name of the action event fired by this
 * button.
 * @param command The name of the action event command fired
 * by this button
 */
 public void setActionCommand(String command)
 {
     actionCommand = command;
 }

/**
 * Returns the command name of the action event fired by this
 * button.
 * @return the action command name
 */
 public String getActionCommand( )
 {
     return actionCommand;
 }

/**
 * Adds the specified action listener to receive action events
 * from this button.
 * @param l the action listener
 */
 public void addActionListener(ActionListener l)
 {
     actionListener = AWTEventMulticaster.add(actionListener, l);
 }

/**
 * Removes the specified action listener so it no longer receives
 * action events from this button.
 * @param l the action listener
 */
 public void removeActionListener(ActionListener l)
 {
     actionListener = AWTEventMulticaster.remove(
                                 actionListener, l);
 }

/**
 * Fire an action event to the listeners.
 */
 protected void fireActionEvent( )
 {
```

```
     if (actionListener != null)
         actionListener.actionPerformed(new ActionEvent(this,
               ActionEvent.ACTION_PERFORMED, actionCommand));
 }
```

These methods encapsulate a mechanism for broadcasting notification that our button was pressed. This notification takes place in the form of an action event. Let's look at these functions one at a time.

```
public void setActionCommand(String command)
{
    actionCommand = command;
}
```

When an `ActionEvent` is sent, it contains a string called an **action command**. This command gives the receiver additional information about what the command is. This routine is used to define the current action command to be sent out by the button. The code simply caches the action command to our data member.

```
public String getActionCommand( )
{
    return actionCommand;
}
```

This routine retrieves the current action command by returning the contents of our `actionCommand` data member.

```
public void addActionListener(ActionListener l)
{
    actionListener = AWTEventMulticaster.add(actionListener, l);
}
```

This routine allows `Listener` objects interested in receiving `ActionEvents` from this button to register themselves with the button.

```
public void removeActionListener(ActionListener l)
{
    actionListener = AWTEventMulticaster.remove( actionListener, l);
}
```

This allows previously interested Listeners to tell the button they no longer need to be notified when an `ActionEvent` is generated by this button.

```
protected void fireActionEvent( )
{
    if (actionListener != null)
        actionListener.actionPerformed(new
        ActionEvent(this,
        ActionEvent.ACTION_PERFORMED, actionCommand));
}
```

This calls the `actionPerformed` method of all the registered listeners with a new action event describing the details of the event, effectively broadcasting the action event to all interested Listeners.

Now it's time to implement `getPreferredSize( )`.

Back to top

Step 11 - Implementing getPreferredSize( )

Because our button selects images from an image pool, we don't know at design time how big to make the button. Thus, we implement a `getPreferredSize` method. This method will be called by the layout manager of our container in order to calculate the button size. We need to return a size based on the size of the image we are using.

```
/**
 * Returns the preferred size of this component.
 * @see #getMinimumSize
 * @see LayoutManager
 */
 public Dimension getPreferredSize( )
 {
     //If the current image is not null, then return the size of
     //the image.
     //If it is null, defer to the super class.
     //Insert "ImageButton getPreferredSize"
```

We are overriding the `getPreferredSize( )` method from java.awt.Component. It returns a `Dimension` object which specifies the preferred height and width of our button. Locate the ImageButton getPreferredSize clipping in the ImageButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Returns the preferred size of this component.
 * @see #getMinimumSize
 * @see LayoutManager
 */
 public Dimension getPreferredSize( )
 {
     //If the current image is not null, then return the size of
     //the image. If it is null, defer to the super class.
     //Insert "ImageButton getPreferredSize"
     if (image != null)
         return new Dimension(image.getWidth(this),
                    image.getHeight(this));

     return super.getPreferredSize( );
}
```

We want to return the size of our current image as the preferred size of the button. The first thing we do is check to see if the image is null. If it is, we call `getPreferredSize( )` from our superclass so that we can use the default component behavior. Otherwise, we return a new `Dimension` object that we create using the height and width of our image object.

We are almost finished with this class. The only thing that remains is drawing our button. This is done in the `paint` method.

Back to top

Step 12 - Implementing paint( )

`Paint( )` is the routine that gets called to draw our object on the screen.

```
/**
 * Paints the component. This method is called when the contents
 * of the component should be painted in response to the
 * component first being shown or damage needing repair. The
 * clip rectangle in the Graphics parameter will be set to the
 * area which needs to be painted.
 * @param g the specified Graphics window
 * @see #update
 */
 public void paint(Graphics g)
 {
    //Let the super class draw, then handle drawing the current
    //image.
    //Insert "ImageButton paint"
```

As you can see from the JavaDoc, the `paint( )` method is called when the contents of the component needs to be drawn due to invalidation of the component or a request for an update. The Graphics parameter `g` is the graphics context the object needs to be drawn in. Locate the ImageButton paint clipping in the ImageButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Paints the component. This method is called when the contents
 * of the component should be painted in response to the
 * component first being shown or damage needing repair. The
 * clip rectangle in the Graphics parameter will be set to the
 * area which needs to be painted.
 * @param g the specified Graphics window
 * @see #update
 */
 public void paint(Graphics g)
 {
    //Let the super class draw, then handle drawing the current
    //image.
    //Insert "ImageButton paint"
    super.paint(g);

    if (image != null)
        g.drawImage(image, 0, 0, this);
}
```

First, we call the `paint` method of our base class to insure that any preparatory imaging occurs. Then we check to see if the image is `null`. If it is not, we draw the current image starting at location 0, 0. This means that we draw the image so that the top left corner is 0 pixels from the top of the button bounds, 0 pixels from the left of the button bounds, and we use the default image dimensions. That's all there is to it!

Back to top

Summary

In review, we set up our class to be derived from `Component`. This allows us to inherit some basic functionality such as being able to draw to the screen, having a bounds, etc. We set up an interface that derived classes will implement to do things like respond to action events. We set up a `MouseListener` and registered it with our button so that we can respond to mouse events such as `MousePressed`, `MouseReleased`, `MouseEntered`, and `MouseExited`. We wrote an inner class to send action events so that our derived classes can respond appropriately to user interaction, and we laid some groundwork for our derived classes such as several image routines for getting, setting, adding and removing images. We wrote a `preferredSize` method so we can tell layout managers how big we want to be, and we added a paint method so that we could draw ourselves.

That may seem like a lot of work, but a lot of it is to simplify the creation of our derived classes which for the most part are much more simple than this class. We have implemented the core functionality for our button, and the road is now much easier from here.

Now we are ready to go back to our main tutorial file and prepare for the next step, Building the Rollover button.

Previous Page

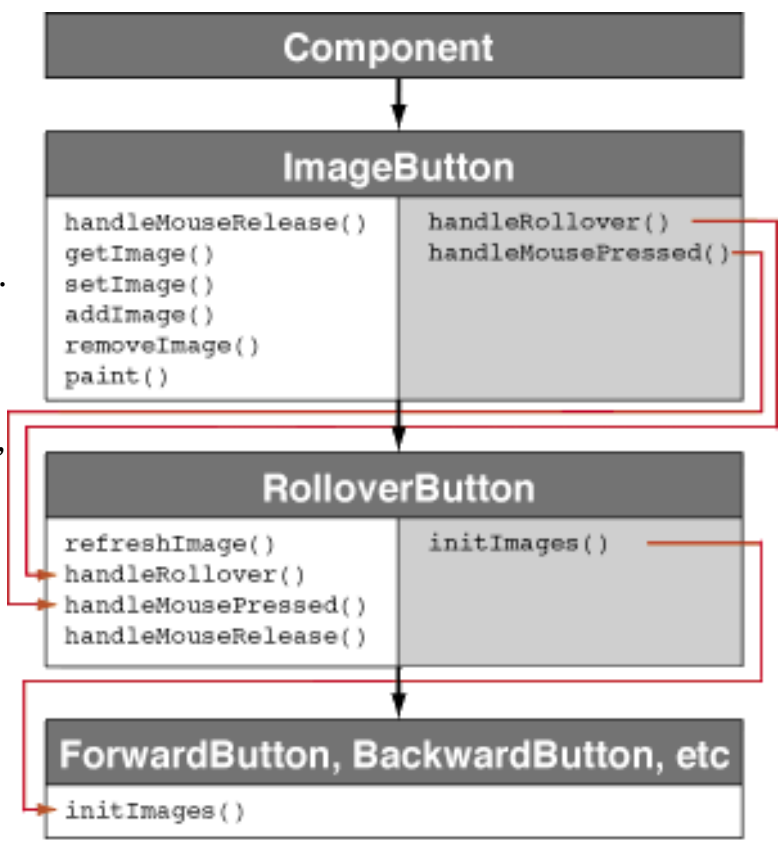
Building the Rollover Button

File: RolloverButton.java

Contents

Overview

The `RolloverButton` is the second tier of a series of classes that encapsulates the functionality of buttons for the slide show controller. As the image below demonstrates, this class is derived from `ImageButton`.



While the `ImageButton` class contains basic functionality such as `MouseEvent` handling and methods to handle images and paint the component (see Building the Image Button), it defines several abstract methods that are implemented in this class. These methods are `handleRollover( )` and `handleMousePressed( )`. This class implements these methods in order to provide rollover functionality; i.e., when the user hovers over a button, the image changes. When the user clicks on the button, the image changes to a depressed button state. The state returns to normal when the user leaves the button.

This class also defines a single abstract function called `initImages( )` which must be implemented in the derived classes `ForwardButton`, `BackwardButton`, and `PlayPauseButton`.

Back to top

Steps to Follow

Step 1 - Declaring the data members

The class `RolloverButton` is an abstract class. Like the `ImageButton` class, this means that it cannot be directly instantiated. Only derived classes that implement the `initImages( )` method which is declared as abstract (more on this later) may be instantiated. We are extending `ImageButton` in order to take advantage of all of the basic image and event handling behavior we implemented in that class.

You may notice that there are no import statements at the beginning of the class. That is because we require no additional imports other than the implicit **java.lang.*** package. Our class knows about the `ImageButton` class because these two classes are in the same package.

```
public abstract class RolloverButton extends ImageButton
{
    //Declare data members
    //Insert "RolloverButton data members"
```

Locate the RolloverButton data members clipping in the RolloverButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public abstract class RolloverButton extends ImageButton
{
    //Declare data members
    //Insert "RolloverButton data members"
    protected String upImage;
    protected String downImage;
    protected String rolloverImage;
```

We declare three data members, all of which are strings. These are the names of the images to be used for the various states. The first, `upImage` is the default image to use when the user is outside the bounds of the button and the button is not depressed. The second, `downImage` is used when the user has clicked the mouse on the button and has not yet released the button. Lastly, the `rolloverImage` is the name of the image to use when the user is hovering over the button with the mouse cursor, but the button has not yet been pressed.

Now that we have our data members, it is time to look at the constructor.

Back to top

Step 2 - Initializing the Rollover Button

We initialize the button in the constructor.

```
Public RolloverButton( )
{
    //Initialize the state of the button
    //Insert "RolloverButton init state"
```

Locate the RolloverButton init state clipping in the RolloverButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public RolloverButton( )
{
    //Initialize the state of the button
    //Insert "RolloverButton init state"
    upImage = "up";
    downImage = "down";
    rolloverImage = "rollover";
    initImages( );
    setImage(upImage);
}
```

We assign the three data members identifiers that we will be using to refer to the individual images. For example, we associate the string “up” with the variable `upImage`. The string “up” is what will be used as the key in the hashtable for the image to be used when the button is in its up state.

Next we call our `initImages( )` method. Again, this is an abstract method and is not defined in this class. Subclasses must override this method and specify the actual images to be used.

Finally, we call `setImage( )` using the upImage as the key. If no image is specified, nothing will happen. We recall from Step 7 in ImageButton that we check to see if an image is loaded. If "up" was not found in our hashtable, it will be null, and thus `setImage( )` won’t do anything. Now it is time to look at `refreshImages( )`.

Back to top

Step 3 - Implementing refreshImage( )

When we need to update the state of the button, `refreshImage( )` is used. It checks the current button state and loads the correct image to display.

```
/**
 * Sub classes need to define this to handle initializing their
 * images, and state information.
 */
 protected abstract void initImages( );

/**
 * Sets the button to be in the correct configuration for the
 * current state.
 */
 Public void refreshImage( )
 {
```

```
    //Handle determining the current state, and reacting
    //appropriately
    //Insert "RolloverButton refreshImage"
```

After the abstract declaration of `initImages( )` which we previously discussed, we reach `refreshImage( )`. This method is only called from our derived class `PlayPauseButton`, but it could be useful to any future derived classes that might need this functionality, which is why we have chosen to place it in this class rather than `PlayPauseButton`. Locate the RolloverButton refreshImage clipping in the RolloverButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Sub classes need to define this to handle initializing their
 * images, and state information.
 */
 Protected abstract void initImages( );

/**
 * Sets the button to be in the correct configuration for the
 * current state.
 */
 Public void refreshImage( )
 {
    //Handle determining the current state, and reacting
    //appropriately
    //Insert "RolloverButton refreshImage"
    if (isMouseInside)
    {
        if (isMouseDown)
        {
            setImage(downImage);
        }
        else
        {
            setImage(rolloverImage);
        }
    }
    else
    {
        setImage(upImage);
    }
}
```

This is fairly self explanatory. We check to see if the mouse is inside the button (recall that the Boolean `isMouseInside` is a data member from our base class, `ImageButton`) and then

check to see if the mouse is down (`isMouseDown` is also from `ImageButton`). If the mouse is down and inside our button, we set the image to our down image. If the mouse is inside the button, but not down, we set the button image to the rollover image. If the mouse is not inside our button, we set the image to the `upImage`.

Here is a logic table for our rollover button:

	Mouse Inside	Mouse Outside
Button Up	rolloverImage	upImage
Button Down	downImage	upImage

Now that we have our rollover behavior specified, it is time to define `handleMousePressed( )`.

Back to top

Step 4 - Implementing handleMousePressed( )

As we recall from ImageButton, when we get a `MouseEvent` of the type `MousePressed`, we set some internal flags and then call the abstract method `handleMousePressed( )`. Here is where we implement that abstract method to respond to mouse presses.

```
/**
 * Gets called when the mouse button is pressed on this button.
 */
 Protected void handleMousePressed( )
 {
     //Set the image to the appropriate image for a mouse press.
     //Insert "RolloverButton mousePressed"
```

Locate the RolloverButton mousePressed clipping in the RolloverButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Gets called when the mouse button is pressed on this button.
 */
 Protected void handleMousePressed( )
 {
     //Set the image to the appropriate image for a mouse press.
     //Insert "RolloverButton mousePressed"
     setImage(downImage);
}
```

When the button is pressed, we set the current image to the `downImage`. Pretty easy! You are beginning to see how easy our underlying architecture is making the definition of this class. Adding extra functionality is quite straightforward.

Now it's time for `handleMouseReleased( )`.

Back to top

Step 5 - Implementing handleMouseReleased( )

The `handleMouseReleased( )` method is called when the mouse is released over the button. It takes two Boolean parameters; the first indicates whether the mouse is inside the button, and the second indicates whether the mouse was pressed inside the button before this method was called.

```
/**
 * Gets called when the mouse button is released on this button.
 * @param isMouseInside, if true, the mouse is located inside
 * the button area, if false the mouse is outside the button.
 * @param wasMouseDown, if true the mouse was down inside this
 * button before this method was called.
 */
 Protected void handleMouseRelease(Boolean isMouseInside,
                                   Boolean wasMouseDown)
 {
     //Set the image to the appropriate image for a mouse
     //release, and calls the super classes version to include
     //inherited functionality.
     //Insert "RolloverButton mouseReleased"
```

Locate the RolloverButton mouseReleased clipping in the RolloverButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Gets called when the mouse button is released on this button.
 * @param isMouseInside, if true, the mouse is located inside
 * the button area, if false the mouse is outside the button.
 * @param wasMouseDown, if true the mouse was down inside this
 * button before this method was called.
 */
 Protected void handleMouseRelease(Boolean isMouseInside,
                                   Boolean wasMouseDown)
 {
     //Set the image to the appropriate image for a mouse
     //release, and calls the super classes version to include
     //inherited functionality.
     //Insert "RolloverButton mouseReleased"
     if (isMouseInside)
     {
         setImage(rolloverImage);
     }
     super.handleMouseRelease(isMouseInside);
}
```

If the user is inside the button we call `setImage( )` with the rollover image. We then call our superclass `handleMouseRelease( )` method to inherit default button release behavior. Regardless of the location of the mouse, we still want the superclass to execute its code.

Last but not least is the function `handleRollover( )`.

Back to top

Step 6 - Implementing handleRollover( )

The last method in this file is `handleRollover( )`. It is used to determine which image to used based on the state information passed into the routine. It looks very similar to `refresh( )` but uses parameterized information instead of stored state information.

```
/**
 * Gets called when the mouse crosses into or out of the button
 * area.
 * @param isMouseInside, is true if the mouse is in the button
 * area, false if it is outside.
 * @param isMouseDown, is true if the mouse button is pressed,
 * false if it is not.
 */
 Protected void handleRollover(Boolean isMouseInside,
                               Boolean isMouseDown)
 {
       //Handle determining the current state, and reacting
       //appropriately
       //Insert "RolloverButton handleRollover"
```

Locate the RolloverButton handleRollover clipping in the RolloverButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Gets called when the mouse crosses into or out of the button
 * area.
 * @param isMouseInside, is true if the mouse is in the button
 * area, false if it is outside.
 * @param isMouseDown, is true if the mouse button is pressed,
 * false if it is not.
 */
 Protected void handleRollover(Boolean isMouseInside,
                               Boolean isMouseDown)
 {
       //Handle determining the current state, and reacting
       //appropriately
       //Insert "RolloverButton handleRollover"
```

```
        if (isMouseInside)
        {
            if (isMouseDown)
            {
                setImage(downImage);
            }
            else
            {
                setImage(rolloverImage);
            }
        }
        else
        {
            setImage(upImage);
        }
}
```

This code should look quite familiar. If the mouse is inside the button and down, we see the image to `downImage`. If it is inside, but not down, we set the image to `rolloverImage`. If the mouse is not inside, set the image to `upImage`.

It happens that the logic for this method turns out to be the same for the refresh method, but this does not necessarily have to be the case. So in order to keep the generality which makes for robust classes, we have chosen not to combine these two methods.

Back to top

Summary

That completes the work we have to do on this file. As you can see, implementing the `RolloverButton` was far easier than `ImageButton`. That is because we are taking advantage of the basic behaviors of `ImageButton` and adding only the functionality necessary to give rollover behavior to our button. We implemented two methods that were declared as abstract from `ImageButton`, `handleRollover( )`, and `handleMousePressed( )` as well as some additional methods for refreshing the state, and handling mouse released messages.

Now it's time to complete the final tier of our button classes, `ForwardButton`, `BackwardButton`, and `PlayPauseButton`. Click here to return to the main tutorial document.

Building the Forward Button

File: ForwardButton.java

Contents

Overview

The `ForwardButton` class is a subclass of `RolloverButton`. It specifies a series of images that represent the appearance of a "forward" or "next" control.

The image on the right shows the various images used by this button. This class implements a single method, `initImages( )` which is declared as abstract in `RolloverButton`.

Steps to Follow

Step 1 - Implementing initImages( )

This class does not import any packages. It uses only the default package **java.lang.*** and classes in its default package. This class is derived from `RolloverButton` which we examined earlier.

```
public class ForwardButton extends RolloverButton
{
    protected void initImages( )
    {
        //Initialize images for the ForwardButton
        //Insert "ForwardButton initImages"
```

We have only a single method that was defined as an abstract method in

RolloverButton. This method specifies the images to be used for this button.

Locate the ForwardButton initImages clipping in the ForwardButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public class ForwardButton extends RolloverButton
{
    protected void initImages( )
    {
 //Initialize images for the ForwardButton
        //Insert "ForwardButton initImages"
        addImage("images/FFF.jpg", upImage);
        addImage("images/FFFa.jpg", downImage);
        addImage("images/FFFb.jpg", rolloverImage);
    }
}
```

To implement this method, all we need to specify the images to be used, and the identifying string. Now we can really see the benefits of our architecture!

Summary

The forward button class is extremely simple. We are benefiting from our pyramid-based component architecture where we place basic functionality into large base classes and then refine behavior in successive classes.

In the next step, we will implement the BackwardButton. Click here to return to the main tutorial file.

Building the Backward Button

File: BackwardButton.java

Contents

Overview

Like its sibling, the `ForwardButton` class, `BackwardButton` is a subclass of `RolloverButton`. It specifies a series of images that represent the appearance of a "backward" or "previous" control.

The image on the right shows the various images used by this button. This class implements a single method, `initImages( )` which is declared as abstract in `RolloverButton`.

Steps to Follow

Step 1 - Implementing initImages( )

This class does not import any packages. It uses only the default package **java.lang.*** and classes in its default package. This class is derived from RolloverButton which we examined earlier.

```
public class BackwardButton extends RolloverButton
{
    protected void initImages( )
    {
        //Initialize images for the BackwardButton
        //Insert "BackwardButton initImages"
```

We have only a single method that was defined as an abstract method in RolloverButton. This method specifies the images to be used for this button. Locate the BackwardButton initImages clipping in the BackwardButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public class BackwardButton extends RolloverButton
{
    protected void initImages( )
    {
 //Initialize images for the BackwardButton
        //Insert "BackwardButton initImages"
        addImage("images/RWW.jpg", upImage);
        addImage("images/RWWa.jpg", downImage);
        addImage("images/RWWb.jpg", rolloverImage);
    }
}
```

This method looks nearly identical to the implementation of the `ForwardButton`. That's because we are doing basically the same thing. The only difference is that we are specifying a different set of images.

Back to top

Summary

This class is very similar to ForwardButton.java. Due to our architecture, this class is fairly trivial. The next (and final) step in our series of button classes is to implement the `PlayPauseButton`. To return to main tutorial file, click here.

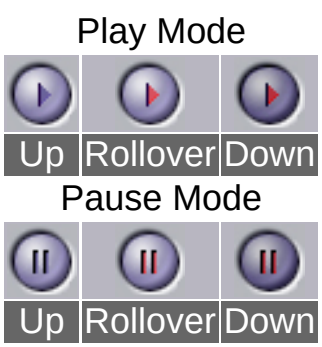
Building the Play/Pause Button

File: PlayPauseButton.java

Contents

Overview

Like its siblings `ForwardButton` and `BackwardButton`, the `PlayPauseButton` class is a subclass of `RolloverButton`. It is a little special, however, because it is a toggle button that switches between a "play" series of images and a "pause" series of images.



The image on the right shows the various images used by this button. This class implements the method, `initImages( )`, which is declared as abstract in `RolloverButton`. Additionally, it keeps track of its own state information and provides two accessor routines, `getState( )` and `setState( )`.

Steps to Follow

Back to top

Step 1 - Declaring and Defining Constants

This class does not import any packages. It uses only the default package **java.lang.*** and classes in its default package. This class is derived from RolloverButton which we examined earlier.

```
public class PlayPauseButton extends RolloverButton
{
    //Declare and define constants
    //Insert "PlayPauseButton Constants"
```

Locate the PlayPauseButton Constants clipping in the ForwardButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public class PlayPauseButton extends RolloverButton
{
    //Declare and define constants
    //Insert "PlayPauseButton Constants"
    public static final String PLAY_UP_IMAGE =  "play up";
    public static final String PLAY_DOWN_IMAGE ="play down";
    public static final String PLAY_ROLLOVER_IMAGE =
                                                "play rollover";
    public static final String PAUSE_UP_IMAGE = "pause up";
    public static final String PAUSE_DOWN_IMAGE="pause down";
    public static final String PAUSE_ROLLOVER_IMAGE =
                                                "pause rollover";

    public static final int PLAY_STATE  = 0;
    public static final int PAUSE_STATE = 1;
```

We are declaring many string constants. A majority of these are to be used for identifiers for the button images as they are placed in the hashtable of button images. The last two integers are constants that define the two possible button states for our toggle button, PLAY_STATE, and PAUSE_STATE.

Now that we have these constants, let's see how they are used in `initImages( )`.

Back to top

Step 2 - Implementing initImages( )

Like the other RolloverButton derivatives, the `initImages( )` method of the `PlayPauseButton` class is used to specify the images to be used for the various button states. This method is slightly different because we have six states instead of three since we are a toggle button.

```
Public void initImages( )
{
    //Initialize images and set the state for the PlayPauseButton
    //Insert "PlayPauseButton initImages"
```

Locate the PlayPauseButton initImages clipping in the ForwardButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public void initImages( )
{
    //Initialize images and set the state for the PlayPauseButton
    //Insert "PlayPauseButton initImages"
    addImage("images/play.jpg", PLAY_UP_IMAGE);
    addImage("images/playa.jpg", PLAY_DOWN_IMAGE);
    addImage("images/playb.jpg", PLAY_ROLLOVER_IMAGE);
    addImage("images/pause.jpg", PAUSE_UP_IMAGE);
    addImage("images/pausea.jpg", PAUSE_DOWN_IMAGE);
    addImage("images/pauseb.jpg", PAUSE_ROLLOVER_IMAGE);

    setState(PLAY_STATE);
}
```

The implementation for this method is pretty straightforward. We add the six images to our hashtable using `addImage( )` and passing our constant identifiers for the hash key string. We then call `setState( )` to set the initial state to the play mode. Let's look at what `setState( )` does.

Back to top

Step 3 - Implementing setState( )

The set state method toggles the state of the button to either PLAY_STATE or PAUSE_STATE.

```
/**
 * Sets the state of the PlayPauseButton.
 * @param the state to use.
 * @see #PLAY_STATE
 * @see #PAUSE_STATE
 */
public void setState(int state)
{
    //Handle switching states
    //Insert "PlayPauseButton setState"
```

`SetState( )` takes and integer state parameter. Locate the PlayPauseButton setState clipping in the ForwardButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Sets the state of the PlayPauseButton.
 * @param the state to use.
 * @See #PLAY_STATE
 * @see #PAUSE_STATE
 */
public void setState(int state)
{
    //Handle switching states
    //Insert "PlayPauseButton setState"
    switch (state)
    {
        case PLAY_STATE:
            upImage = PLAY_UP_IMAGE;
            downImage = PLAY_DOWN_IMAGE;
            rolloverImage = PLAY_ROLLOVER_IMAGE;
            setActionCommand("" + PLAY_STATE);
            this.state = state;
            refreshImage( );
            break;
        case PAUSE_STATE:
            upImage = PAUSE_UP_IMAGE;
            downImage = PAUSE_DOWN_IMAGE;
            rolloverImage = PAUSE_ROLLOVER_IMAGE;
            setActionCommand("" + PAUSE_STATE);
            this.state = state;
            refreshImage( );
            break;
     }
}
```

Our `setState( )` method does a switch on the state that is passed in. If the state is play, we set the `upImage`, `downImage`, and `rolloverImage` to the appropriate image strings. Then we set the action command to the play state. This allows listeners to recognize whether the event is a play command or a pause command. We change our internal state variable (`state` is a private data member that we will declare in Step 5), and `call refreshImage( )` from `RolloverButton` to update the visual appearance of our button.

The PAUSE_STATE case is very similar. We set cache the appropriate image strings in our data members for the images, set our action command member, store the current state, and refresh our button.

Now that we have a `setState( )`, let's look at `getState( )`.

Back to top

Step 4 - Implementing getState( )

Our state accessor routine, `getState( )` is fairly trivial. All it has to do is return the current state as either PLAY_STATE or PAUSE_STATE.

```
/**
 * Gets the state of the PlayPauseButton.
 * @return the state currently in use.
 * @See #PLAY_STATE
 * @see #PAUSE_STATE
 */
 public int getState( )
 {
     //Return the current state
     //Insert "PlayPauseButton getState"
```

Locate the PlayPauseButton getState clipping in the ForwardButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Gets the state of the PlayPauseButton.
 * @Return the state currently in use.
 * @See #PLAY_STATE
 * @see #PAUSE_STATE
 */
 public int getState( )
 {     //Return the current state
     //Insert "PlayPauseButton getState"
     return state;
}
```

All we do is return our stored state variable. Lastly, it is time to declare a single private variable.

Back to top

Step 5 - Declaring the State Variable

We have a single private variable that we need to declare.

```
Public int getState( )
{
     //Return the current state
     //Insert "PlayPauseButton getState"
     return state;
}
//Declare data members
//Insert "PlayPauseButton data members"
```

Locate the PlayPauseButton data members clipping in the ForwardButton folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public int getState( )
{
     //Return the current state
     //Insert "PlayPauseButton getState"
     return state;
}

//Declare data members
//Insert "PlayPauseButton data members"
protected int state;
```

This finishes off the `PlayPauseButton` class.

Back to top

Summary

In this section of the tutorial, we started by defining a base button class called `ImageButton`. We built a class on top of that called `RolloverButton` that defined specific behavior for changing the button images based on user interaction. We built two more buttons, `ForwardButton` and `BackwardButton` that defined a set of images to use for their display. Lastly, we built a `PlayPauseButton` class that toggled between two separate states while inheriting our image and rollover behavior.

In our next class, we will be working with is the `CloseBoxButton`. Click here to return to the main tutorial file.

An Introduction to Java Programming

Previous Section | Table of Contents | Next Section

Step 7 - Building the Close Box Button

The `CloseBoxButton` class extends the `RolloverButton` class. It implements a button that behaves like a simple rectangular close box in the upper left-hand corner of the controller floating palette.

As we have done before, close the **PlayPauseButton** folder and open the **CloseBoxButton** Source Clipping folder. Open the **CloseBoxButton.java** from the Project window before proceeding to the next set of steps in the section Building the Close Box Button.

Back to top

Step 8 - Building the Controller

The `controller` class is a floating palette that contains the controller buttons (forward, backward, and play/pause buttons), as well as the close box button.

Once again, close any open source files and open the **Controller.java** skeleton file. Open the **Controller Source Clipping** folder in the Finder.

After completing this step, proceed to the steps in the section Building the Controller.

Back to top

Step 9 - Building the Slide Show

The `SlideShow` class is the main application implementation file. It provides all of the user interface features, such as menus and windows typical of an application, and ties together all of our other classes into a cohesive unit.

Before proceeding to the steps for the `SlideShow`, close any open source files, and open the **SlideShow.java** source file and the **SlideShow Source Clipping** folder in the Finder. Once again, you may need to resize or reposition your windows to make optimal use of your screen real estate. Once this preparation is completed, proceed to the steps in the section Building the SlideShow Application.

Back to top

Step 10 - Building the Image Name Filter

The `ImageNameFilter` class is used by the `SlideShow` class to filter files displayed in the open file dialog box.

As we have done before, close the **SlideShow** folder and open the **ImageNameFilter Source Clipping** folder. Open the **ImageNameFilter.java** from the Project window before proceeding to the next steps in the section Building the Image Name Filter.

Back to top

Previous Section Table of Contents Next Section

Building the Close Box Button

File: CloseBoxButton.java

Contents

Overview

The `CloseBoxButton` is a special class that is derived from `RolloverButton`. Its function mimics a standard close box such as the one that you would find in the upper left-hand corner of a window. The `CloseBox` is used by the `Controller` class.

The image on the right shows the two images used by this button. Note that there is only an up state and a down state. There is no rollover state. This class implements a single method, `initImages( )`, which is declared as abstract in `RolloverButton`.

Steps to Follow

Step 1 - Implementing initImages( )

This class does not import any packages. It uses only the default package **java.lang.*** and classes in its default package. This class is derived from `RolloverButton`, which we examined earlier.

```
public class CloseBoxButton extends RolloverButton
{
    protected void initImages( )
    {
        //Initialize images for the CloseBoxButton
        //Insert "CloseBoxButton initImages"
```

We have only a single method that was defined as an abstract method in

RolloverButton. This method specifies the images to be used for this button.

Locate the CloseBoxButton initImages clipping in the `CloseBoxButton` folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public class CloseBoxButton extends RolloverButton
{
    protected void initImages( )
    {
 //Initialize images for the CloseBoxButton
        //Insert "CloseBoxButton initImages"
        addImage("images/closebox.jpg", upImage);
        addImage("images/closeboxa.jpg", downImage);
    }
}
```

Note that we only specify an up image and a down image. We do not specify a rollover image. Is this going to cause a problem? No, because the RolloverButton code we inherit is smart. When `handleRollover( )` is called, it will try to call `setImage(rolloverImage)`. There will be no entry in the hashTable for the `rolloverImage`, so it will return null. Our code in `ImageButton.setImage( )` knows not to change the current image if it is null. Thus, the button will not change its visual appearance if a rollover state occurs because there is no rollover image.

Back to top

Summary

The close box button is a very simple class. Like its fellow `RolloverButton`-derived classes, it overrides `initImages( )` but it specifies only two images. This is because we only use an up state and a down state and no intermediary states.

Now let's look at the next class, `Controller`, by returning to the main tutorial file here.

Building the Controller

File: Controller.java

Contents

Overview

The `Controller` class is a palette that provides a user interface for the slideshow. It is a floating palette that contains three buttons: a backward button, a play/pause button, and a forward button. It also has a close box that dismisses the window.

The `Controller` is designed to be a standalone component. It has a container that encloses the buttons and arranges them in a nice simple horizontal strip, and it listens for events from the buttons and rebroadcasts them. This allows clients to listen to the controller and receive messages, instead of dealing with each individual button separately.

The controller can be dismissed by clicking on the close box.

Steps to Follow

Back to top

Step 1 - Declaring the Controller Constants

The controller class imports several packages including the abstract window toolkit (or `awt`), as well as two packages from **java.awt.event** that are needed to handle events.

The class is derived from `window` since it is a floating window.

```
import java.awt.*;
Import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Controller extends Window
{
    //Declare constants
    //Insert "Controller Constants"
```

Locate the Controller Constants clipping in the Controller folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
import java.awt.*;
Import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Controller extends Window
{
    //Declare constants
    //Insert "Controller Constants"
    public static final String BACKWARD_COMMAND = "backward";
    public static final String FORWARD_COMMAND = "forward";
    public static final String PLAY_COMMAND = "play";
    public static final String PAUSE_COMMAND = "pause";

    protected static final String imagePath =
                                        "images/controlborder.jpg";
    protected static final int collapsedWidth = 9;
```

Most of this code is self explanatory. We create four strings that will be used as command names. When the user clicks on one of the buttons in the controller, we will broadcast a message with a command string that is appropriate for the action. For example, if the user clicks on the backwards button, we will broadcast a `BACKWARD_COMMAND` so that our listeners know how to respond.

We then declare a **String** that specifies a relative image path to the image that we will be using as the background of our controller. That image looks like this:

We will be placing the buttons on top of this image and also placing the close box button in the upper left-hand corner.

Our last constant is an integer that specifies how wide we want the window to be when it is collapsed. This number corresponds to the “thumb area” or gray region in the image.

Now that we have created the constants that we will be using in this class, let’s take a look at the data members.

Back to top

Step 2 - Declaring the Controller Data Members

We need to add several data members to our class to keep track of objects such as our buttons and background image.

```
    protected static final int collapsedWidth = 9;
    //Declare data members
    //Insert "Controller data members"
```

Locate the Controller data members clipping in the Controller folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
    protected static final int collapsedWidth = 9;

    //Declare data members
    //Insert "Controller data members"
    protected Image image;
    protected Frame frame;

    protected BackwardButton backwardButton1;
    protected PlayPauseButton playPauseButton1;
    protected ForwardButton forwardButton1;
    protected CloseBoxButton closeBoxButton1;
    protected DumbContainer container;

    protected ActionListener actionListener = null;
```

In order to understand what all of these data members are for, let's take a few moments to examine the taxonomy of this `Controller` class.

The controller is a window that is owned by a parent frame (the `SlideShow` class). The controller window has a container with a layout manager that encloses the three buttons—the forward button, the backward button, and the play/pause button. We also have a close button in the upper left-hand corner. This is not enclosed by the dumb container object.

Looking at the data members, we can see variables that store references to these objects. First, we have an image object that stores a reference to our background image.

Next, we have a reference to the frame that acts as our supervisor, followed by the four buttons on our controller. Finally, we have a dumb container object that arranges our buttons, and an action listener object that receives action events from the buttons in response to user actions. What is a `DumbContainer`? Well, if you scan down to the very end of the source file, you will see the following:

```
//Inner class so we can instantiate a simple lightweight container
//for use in laying out the controller properly.
class DumbContainer extends Container { }
```

You might think that this looks a bit strange. Why bother to override a class if we are not overriding any methods? Good question. The answer is that since `Container` is an abstract class, we cannot directly instantiate it. Therefore, we must subclass it if we want to instantiate it.

Now let's look at the constructor for our controller.

Back to top

Step 3 - Specifying the Controller's Parent Frame

The constructor for our `Controller` class takes a single parameter—the parent frame of our window.

```
/**
 * Creates a Controller object with the specified parent Frame.
 * @param the parent frame to associate the controller with.
 */
 public Controller(Frame parent)
 {
```

```
    //Setup parent info for this window
    //Insert "Controller parent info"
```

Locate the Controller parent info clipping in the Controller folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
* Creates a Controller object with the specified parent Frame.
 * @param the parent frame to associate the controller with.
 */
 Public Controller(Frame parent)
 {
    //Setup parent info for this window
    //Insert "Controller parent info"
    super(parent);
    frame = parent;
```

The first step in our constructor is to call the constructor of our superclass (`java.awt.Window`) and pass the `Frame`. This step performs common window initialization that we are inheriting from our base (`Window`) class. After initializing our superclass, we store the parent in our local data member for later use.

Our next step is to create and initialize the controls within our controller.

Back to top

Step 4 - Initializing Controls

We now have to create instances of the various objects hosted by our controller and initialize them.

```
    super(parent);
    frame = parent;
    //INIT_CONTROLS
    //Setup and layout objects in the controller
    //Insert "Controller init controls"
```

Locate the Controller init controls clipping in the Controller folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
    super(parent);
    frame = parent;

    //INIT_CONTROLS
    //Setup and layout objects in the controller
    //Insert "Controller init controls"
```

```
    setVisible(false);
    setLayout(null);
    setSize(90,30);

    closeBoxButton1 = new CloseBoxButton( );
    closeBoxButton1.setLocation(1,1);
    closeBoxButton1.setSize(closeBoxButton1.getPreferredSize( ));
    add(closeBoxButton1);

    container = new DumbContainer( );
    container.setLayout(new FlowLayout(FlowLayout.LEFT,0,0));
    container.setBounds(9, 1, 81, 28);

    backwardButton1 = new BackwardButton( );
    backwardButton1.setBounds(10,1,20,40);
    container.add(backwardButton1);

    playPauseButton1 = new PlayPauseButton( );
    playPauseButton1.setBounds(0,0,20,40);
    container.add(playPauseButton1);

    forwardButton1 = new ForwardButton( );
    forwardButton1.setBounds(0,0,20,40);
    container.add(forwardButton1);

    add(container);
```

There's a lot of code here, so let's break it down into smaller groups to aid our discussion.

```
setVisible(false);
setLayout(null);
setSize(90,30);
```

First, we make the controller non-visible. This is so the user doesn't see the buttons being added one at a time to the form. This would be sloppy. We want them only to see the end result. Next, we set the window layout to `null`. This is OK, because we want to use pixel positioning for the items in the form. The container will have its own layout that will nicely position the buttons. We set the size of the controller to 90 pixels wide and 30 pixels tall.

```
closeBoxButton1 = new CloseBoxButton( );
closeBoxButton1.setLocation(1,1); closeBoxButton1.setSize(
                        closeBoxButton1.getPreferredSize( ));
add(closeBoxButton1);
```

Now we create the close box and position it at 1, which is one pixel from the top left corner of the screen. We change the size of the box based on the preferred size (which we defined in `ImageButton` to be the size of the image used by the button). Once we size and position the close box, we add it to the window.

Before we add the buttons to the form, we create our instance of the `DumbContainer` class.

```
container = new DumbContainer( );
```

```
container.setLayout(new FlowLayout(FlowLayout.LEFT,0,0));
container.setBounds(9, 1, 81, 28);
```

We create the DumbContainer, and set the layout manager to FlowLayout. The flow layout manager positions its contained objects by placing them in a row, sized at their preferred size. If the horizontal space in the container is too small to put all the components in one row, `FlowLayout` uses multiple rows. Within each row, components are centered (the default), left-aligned, or right-aligned as specified when the `FlowLayout` is created. Here, we are specifying that the components are left-aligned. You can get more information on FlowLayout on JavaSoft's web site. Lastly, we position the container 9 pixels from the left edge of our window, 1 pixel from the top, and make it 81 pixels wide and 28 high.

Now that we have our dumb container in place, we can start adding buttons to it.

```
backwardButton1 = new BackwardButton( );
backwardButton1.setBounds(10,1,20,40);
container.add(backwardButton1);
```

We create a new backwards button instance and add it to our dumb container. Since our container is using left alignment, this item will be the leftmost object.

The forward button and play/pause button are made the same way.

```
playPauseButton1 = new PlayPauseButton( );
playPauseButton1.setBounds(0,0,20,40);
container.add(playPauseButton1);

forwardButton1 = new ForwardButton( );
forwardButton1.setBounds(0,0,20,40);
container.add(forwardButton1)
```

We create them, and add them to the container.

```
add(container);
```

Last, but not least, we add our dumb container to the window. Now we are going to skip down a bit and define our inner classes for handling action events.

Back to top

Step 5 - Handling Controller Actions

Skip down to nearly the bottom of the source file. You should see the following:

```
public Dimension getPreferredSize( )
{
    //If the current image is not null, then return the size of
    //the image. If it is null, defer to the super class.
    //Insert "Controller getPreferredSize"
}

//Inner class so we can instantiate a simple lightweight
//container for use in laying out the controller properly.
Class DumbContainer extends Container { }
```

```
//Inner class to handle action events
//Insert "Controller Action"
```

The controller needs to listen for action events from the buttons so that it can rebroadcast these messages. You may be thinking that this sounds a little inefficient. If a button sends a message to the controller saying, "Someone clicked the forward button. Pass it on," and then the controller sends a message "Yo, someone pressed my forward button," that extra step may seem redundant. Why can't a class listen directly to the individual buttons and skip the controller? Good question. What we are doing here is called object-oriented programming. We have created a widget called a controller that an object can listen to. The object doesn't need to know what kind of objects the controller contains and listen to those objects separately. The object can simply listen to the controller directly. Thus, if the interface between the controller and the buttons change, classes dependent on the controller still work correctly. This is one of the fundamentals that makes object-oriented languages like Java very nice.

Now let's implement the action handler for the controller. Locate the Controller Action clipping in the Controller folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public Dimension getPreferredSize( )
{
    //If the current image is not null, then return the size of
    //the image. If it is null, defer to the super class.
    //Insert "Controller getPreferredSize"
}

//Inner class so we can instantiate a simple lightweight
//container for use in laying out the controller properly.
Class DumbContainer extends Container { }

//Inner class to handle action events
//Insert "Controller Action"
class Action implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        Object object = event.getSource( );
        if (object == backwardButton1)
            backwardButton1_ActionPerformed(event);
        else if (object == forwardButton1)
            forwardButton1_ActionPerformed(event);
        else if (object == playPauseButton1)
            playPauseButton1_ActionPerformed(event);
        else if (object == closeBoxButton1)
            closeBoxButton1_ActionPerformed(event);
     }
}
```

Our inner class implements the `ActionListener` interface. This interface defines a single `actionPerformed` method which we implement in our class. By implementing this method, we can respond to events of type `ActionEvent`.

In our first line of this method, we create a temporary object reference and assign it to the source of the event

(the object that broadcast the action event). We do this so that we can determine which specific button the message came from.

Next, we have a series of if statements that call specific methods based on the event source object. We respond to these messages in external methods instead of doing the work inline in this class to make the code more flexible. If we call a method, we could override the behavior in the future. If the code is placed inline, it is difficult to change the behavior of a single object without overriding the entire class. We will look at the implementation of these methods in our next step.

Back to top

Step 6 - Handling Button Actions

We define a single method to respond to `ActionEvent` messages for each of the four buttons.

```
        else if (object == playPauseButton1)
            playPauseButton1_ActionPerformed(event);
        else if (object == closeBoxButton1)
            closeBoxButton1_ActionPerformed(event);
    }
}
//Routines to handle action events from the different buttons
//Insert "Controller button actions"
```

Locate the Controller button actions clipping in the Controller folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
        else if (object == playPauseButton1)
            playPauseButton1_ActionPerformed(event);
        else if (object == closeBoxButton1)
            closeBoxButton1_ActionPerformed(event);
    }
}

//Routines to handle action events from the different buttons
//Insert "Controller button actions"
void backwardButton1_ActionPerformed(ActionEvent event)
{
    fireActionEvent(BACKWARD_COMMAND);
}

void forwardButton1_ActionPerformed(ActionEvent event)
{
    fireActionEvent(FORWARD_COMMAND);
}

void playPauseButton1_ActionPerformed(ActionEvent event)
{
    String command = event.getActionCommand( );
    try
    {
        int state = Integer.valueOf(command).intValue( );
```

```
        switch (state)
        {
            case PlayPauseButton.PLAY_STATE:
                fireActionEvent(PLAY_COMMAND);
                playPauseButton1.setState(PlayPauseButton.PAUSE_STATE);
                break;
            case PlayPauseButton.PAUSE_STATE:
                fireActionEvent(PAUSE_COMMAND);
                playPauseButton1.setState(PlayPauseButton.PLAY_STATE);
                break;
         }
    }
    catch(NumberFormatException exc) { }
}

void closeBoxButton1_ActionPerformed(ActionEvent event)
{
    setVisible(false);
    dispose( );
}
```

The first two methods are very simple.

```
void backwardButton1_ActionPerformed(ActionEvent event)
{
    fireActionEvent(BACKWARD_COMMAND);
}

void forwardButton1_ActionPerformed(ActionEvent event)
{
    fireActionEvent(FORWARD_COMMAND);
}
```

For the backward button, we call our `fireActionEvent( )` method (implemented in Step 11) with the `BACKWARD_COMMAND` message. For the forward button, we call `fireActionEvent( )` with the `FORWARD_COMMAND` message.

The code for the play/pause button is more complex since it is a toggle button with two different functional modes.

```
void playPauseButton1_ActionPerformed(ActionEvent event)
{
    String command = event.getActionCommand( );
```

First, we get the action command string from the button so we can tell whether we are in the play state, or the pause state.

```
    try
    {
        int state = Integer.valueOf(command).intValue( );
```

Here we use the try keyword to start an exception handling block. We want to handle any `NumberFormatExceptions` that may be thrown by any of the following lines of code.

Next, we convert the action command into an integer so that we can use it in our switch statement. In Java, only integral types such as an int, or long can be used in a switch statement.

```
        switch (state)
        {
            case PlayPauseButton.PLAY_STATE:
                 fireActionEvent(PLAY_COMMAND);
                 playPauseButton1.setState(PlayPauseButton.PAUSE_STATE);
                 break;
            case PlayPauseButton.PAUSE_STATE:
                 fireActionEvent(PAUSE_COMMAND);
                 playPauseButton1.setState(PlayPauseButton.PLAY_STATE);
                 break;
         }
     }
```

If the state is for the play button, we fire an action event with the play command and toggle the button state by calling `PlayPauseButton.setState( )` with `Pause` as the argument. If the state is for the pause button, we do the converse and fire a pause command and toggle the button to the play state.

```
catch(NumberFormatException exc) { }
}
```

Lastly, we catch and ignore any `NumberFormatExceptions` which might have occurred.

Our close box action performed method is responsible for closing the controller.

```
void closeBoxButton1_ActionPerformed(ActionEvent event)
{
    setVisible(false);
    dispose( );
}
```

First, we hide the controller, and then we dispose of it. We hide it first so the user does not see the individual pieces of the window being destroyed.

Now that we have a class to handle action events, we need to return to our constructor and register our listeners.

Back to top

Step 7 - Registering our Event Listeners

Go up to the top of the file to our controller constructor.

```
    forwardButton1 = new ForwardButton( );
forwardButton1.setBounds(0,0,20,40);
    container.add(forwardButton1);

    add(container);
    //REGISTER_LISTENERS
    //Register our action listener with our buttons
    //Insert "Controller register listeners"
```

This is where we will register our listeners. Locate the Controller register listeners clipping in the Controller folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
    forwardButton1 = new ForwardButton( );
forwardButton1.setBounds(0,0,20,40);
    container.add(forwardButton1);

    add(container);

    //REGISTER_LISTENERS
    //Register our action listener with our buttons
    //Insert "Controller register listeners"
    Action aAction = new Action( );
    backwardButton1.addActionListener(aAction);
    forwardButton1.addActionListener(aAction);
    playPauseButton1.addActionListener(aAction);
    closeBoxButton1.addActionListener(aAction);
```

We will now register a listener for each of our buttons. You must register a listener with every object that you wish to receive events from. Since we want to know when each of our four buttons is pressed, we need to register a listener with each button.

Our first step is to create an instance of our inner class that is the listener. We then add this class as a listener to each control.

Now it is time to wrap up our constructor.

Back to top

Step 8 - Completing our Controller Constructor

We have added our buttons to the form, sized our controller, and registered listeners with each button. The only thing left is loading and positioning our background image.

```
    forwardButton1.addActionListener(aAction);
    playPauseButton1.addActionListener(aAction);
    closeBoxButton1.addActionListener(aAction);
    //Initialize state information.
    //Insert "Controller init state"
```

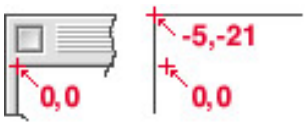
Locate the Controller init state clipping in the Controller folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
    forwardButton1.addActionListener(aAction);
    playPauseButton1.addActionListener(aAction);
    closeBoxButton1.addActionListener(aAction);
    //Initialize state information.
    //Insert "Controller init state"
    image = Misc.loadImage(imagePath, parent, true);

    setSize(getPreferredSize( ));
    //Work around a MRJ Bug.
    setLocation(-5,-21);
}
```

We use an external routine, `loadImage( )` to retrieve the background image from disk and store it to our image data member. As we previously mentioned, this routine mainly performs error checking and is provided to add code robustness. Its implementation is beyond the scope of this tutorial. Suffice it to say that it loads the background image.

Next, we set the size of the background image variable to the preferred size which is the physical dimensions of the picture. We then set the location to be 5 pixels to the left of our window and 21 pixels above. We do this, as the comment mentions, to avoid a placement bug in the MRJ that assumes that all windows have frames.

The image on the right illustrates this problem. Our window has no border whatsoever, as compared with the standard Macintosh window that has a 5 pixel left border and a 21 pixel top border. Unfortunately, the MRJ expects all windows to have this border. In our case, we need to compensate by drawing at -5, -21 which is the upper left corner of our borderless window.



Now it is time to implement the rest of the methods in our class. The first is an accessor for our playState data member, `setPlayState( )`.

Back to top

Step 9 - Implementing setPlayState( )

There is a standard data accessor routine style in the Java Beans specification. For routines that set the value of a data member, they must follow the format set `DataMember( )` where `DataMember` is the name of the member variable. We don't really have a data member called `playState` since the state information is contained in the play/pause button, but we have chosen to follow this specification as if we had a `playState` property. Thus, we have named our accessor `setPlayState( )`.

```
/**
 * Set the state of the Play/Pause button
 * @param if true, the button will be in the Play state;
 * If false it will be in the Pause state.
 * @see #isPlayState
 */
public void setPlayState(Boolean isPlay)
{
    //Handle setup for the appropriate state
    //Insert "Controller setPlayState"
```

Note that we have decided to use a Boolean for the play state. `True` means that we are in play mode and `false` means that we are in pause mode. Locate the Controller setPlayState clipping in the Controller folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Set the state of the Play/Pause button
 * @param if true, the button will be in the Play state;
 * If false it will be in the Pause state.
 * @See #isPlayState
 */
 public void setPlayState(Boolean isPlay)
 {
     //Handle setup for the appropriate state
     //Insert "Controller setPlayState"
     if (isPlay)
         playPauseButton1.setState(PlayPauseButton.PLAY_STATE);
     else
         playPauseButton1.setState(PlayPauseButton.PAUSE_STATE);
}
```

If the Boolean that is passed to our method is passed in as true, we want to switch to the play state which we do by calling `setState( )` from the `playPauseButton` with `PLAY_STATE` as the parameter. Conversely, if we get passed in a false, `setState( )` with `PAUSE_STATE` as the argument. Pretty straightforward.

Now we will write the routine that returns the current play state.

Back to top

Step 10 - Implementing isPlayState( )

Now we are going to implement the routine that is used to retrieve the current play state.

```
/**
 * Get the current state of the Play/Pause button
 * @return true if the button is in the Play state,
 * false if it is in the Pause state.
 * @See #setPlayState
 */
 public Boolean isPlayState( )
 {
     //Return the current state
     //Insert "Controller isPlayState"
```

The first thing that you will notice about this routine is that it is called `isPlayState( )` instead of `getPlayState( )` as you may have expected. That is because this accessor routine is a special case in that it returns a Boolean value. Naming conventions dictate that all accessors that return Boolean values should use a `isDataMember( )` naming style, while non-Boolean methods use `setDataMember( )` naming style.

Locate the Controller isPlayState clipping in the Controller folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Get the current state of the Play/Pause button
 * @return true if the button is in the Play state,
 * false if it is in the Pause state.
 * @See #setPlayState
 */
 public Boolean isPlayState( )
 {
     //Return the current state
     //Insert "Controller isPlayState"
     return playPauseButton1.getState( ) ==
               PlayPauseButton.PLAY_STATE;
}
```

The implementation of this method is pretty basic. We get the state of the play/pause button and return `true` if the state of the button is the same as `PLAY_STATE`, otherwise, we return false. Now it's time to implement `addActionListener( )`.

Back to top

Step 11 - Implementing Action Event Management Routines

Now it's time for the step we have all been waiting for- implementing `addActionListener( )`. This is very similar to what we did in the `ImageButton` source. The concepts are the same.

```
Public Boolean isPlayState( )
{
    //Return the current state
    //Insert "Controller isPlayState"
    return playPauseButton1.getState( ) ==
               PlayPauseButton.PLAY_STATE;
}
    //Routines for handling ActionListener management.
    //Insert "Controller Action Management"
```

Locate the Controller Action Management clipping in the Controller folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public Boolean isPlayState( )
{
    //Return the current state
    //Insert "Controller isPlayState"
    return playPauseButton1.getState( ) ==
               PlayPauseButton.PLAY_STATE;
}

    //Routines for handling ActionListener management.
    //Insert "Controller Action Management"
```

```
/**
 * Adds the specified action listener to receive action events.
 * @param l the action listener
 */
 public void addActionListener(ActionListener l)
 {
     actionListener = AWTEventMulticaster.add(actionListener, l);
 }

/**
 * Removes the specified action listener so it no longer receives
 * action events from this component.
 * @param l the action listener
 */
 public void removeActionListener(ActionListener l)
 {
     actionListener = AWTEventMulticaster.remove(actionListener, l);
 }

 /**
  * Fire an action event to the listeners.
  * @param command, the command String to send with the ActionEvent
  */
 protected void fireActionEvent(String command)
 {
     if (actionListener != null)
         actionListener.actionPerformed(new ActionEvent(this,
         ActionEvent.ACTION_PERFORMED, command));
 }
```

We have added code for three separate routines. Let's examine each separately.

```
Public void addActionListener(ActionListener l)
{
    actionListener = AWTEventMulticaster.add(actionListener, l);
}
```

The `addActionListener( )` is provided for the use of external classes that want to be able to listen to the controller and receive action events from it (in this case, it will be called by `SlideShow`). The AWTEventMulticaster is a convenience class which maintains a list of objects that want to receive notification from that object.

`AWTEventMulticasters` are somewhat tricky beasts. The `add( )` method takes the current multicaster and the listener to be added and returns a new `AWTEventMulticaster` with the listener added to its internal list. Thus, for the implementation of this routine, we call `add( )` with our current multicaster object that we have stored in our actionListener data member, and the listener to be added. We store the result in our actionListener data member (replacing the old one). Remove functions the same way:

```
public void removeActionListener(ActionListener l)
{
    actionListener = AWTEventMulticaster.remove(actionListener, l);
```

```
}
```

Thus, we use the same technique as `addActionListener( )` but we call the multicaster's `remove( )` method instead. The last item, `fireActionEvent:`

```
protected void fireActionEvent(String command)
{
    if (actionListener != null)
        actionListener.actionPerformed(new ActionEvent( this,
        ActionEvent.ACTION_PERFORMED, command));
}
```

The first step is the check the `actionListener` to make sure that it is not null. If this is null, we have no registered action listeners and, therefore, return from the function. Otherwise, we need to tell the multicaster to broadcast an action event to each of its registered listeners. To do so, we call the `actionPerformed( )` method of our `AWTEventMulticaster` and pass a new action event as the parameter. This action event takes three parameters: the originator of the action event (this), the type of event (`ACTION_PERFORMED`), and the string command. It is the responsibility of the multicaster to insure that this event is propagated correctly to each listener.

Now that we have our action handling routines in order, it's time to implement our `paint( )` method.

Back to top

Step 12 - Implementing paint( )

Our `paint` routine needs to draw the background as well as our buttons and close box.

```
Public void paint(Graphics g)
{
    //Handle painting the border image, and let the super class
    //paint the rest.
    //Insert "Controller paint"
```

Locate the Controller controller paint clipping in the Controller folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public void paint(Graphics g)
{
    //Handle painting the border image, and let the super class
    //paint the rest.
    //Insert "Controller paint"
    if (image != null)
        g.drawImage(image, 0, 0, this);
    super.paint(g);
}
```

As our comments indicate, we paint the background image ourselves by calling `drawImage( )` if the image is non-null. Then we call our superclass and let it do its painting which draws the buttons and window.

Next is the similar update method.

Back to top

Step 13 - Implementing update( )

The update method is called when a window or a portion of the screen needs to be refreshed.

```
Public void update(Graphics g)
{
    //Override update to simply call paint to reduce flicker.
    //Insert "Controller update"
```

Locate the Controller controller update clipping in the Controller folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public void update(Graphics g)
{
    //Override update to simply call paint to reduce flicker.
    //Insert "Controller update"
    paint(g);
}
```

As the comment suggests, we call `paint( )` from our update method. This reduces flicker because the default behavior of the update method of heavyweight components (such as a window) erases the content area before calling `paint`. We override this functionality to avoid erasing the window background. Since we have a background image, we can simply draw the background to clear the window.

Back to top

Step 14 - Implementing getPreferredSize( )

The `getPreferredSize( )` method returns the dimensions of the controller based on the size of the background image.

```
/**
 * Gets the size the controller should be to look its best
 * @return the dimensions the controller renders its self the best.
 */
 Public Dimension getPreferredSize( )
 {
    //If the current image is not null, then return the size
    //of the image. If it is null, defer to the super class.
    //Insert "Controller getPreferredSize"
```

Locate the Controller update clipping in the Controller folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Gets the size the controller should be to look its best
 * @return the dimensions the controller renders its self the best.
 */
 Public Dimension getPreferredSize( )
 {
    //If the current image is not null, then return the size
    //of the image. If it is null, defer to the super class.
    //Insert "Controller getPreferredSize"
    if (image != null)
        return new Dimension (image.getWidth(frame),
        image.getHeight(frame));

    return super.getPreferredSize( );
}
```

As you can see from the JavaDoc comments, we first check to see if the background image is `null`. If it is not, we create a new dimension object with the width and height of the background image object. If the image is `null`, we return the preferred size of our parent object.

Back to top

Summary

That completes this file. We have built a controller that contains a dumb Container responsible for the layout of the three-image button controls. We created an inner class to handle action events and registered ourselves as a listener of the image buttons. We created some action routines to allow clients of the controller (`SlideShow`) to register with the `Controller` class for `ActionEvents`. Finally, we rounded out our class by adding a paint method, and `getPreferredSize( )`.

Next we will examine the file SlideShow.java. Return to the main tutorial file for preparatory instructions by clicking here.

Building the Slide Show

File: SlideShow.java

Contents

Summary

Overview

The **slideshow** is the main class of our application. Not only does it provide the main entry point to our application, it ties together all of the classes that we previously defined. It allows the user to select a number of image files which it will display sequentially when the user clicks on the play button. It creates a number of menu items that allows the user to specify options such as full screen and hide the control strip. It also provides facilities for advancing and going backwards through the image list.

This class demonstrates a number of interesting concepts such as how to handle menus, manipulate windows, and provide advanced Macintosh functionality via the `MRJUtilites` classes. With these classes, we can respond to core `AppleEvents` and handle files in a more mac-like manner.

Steps to Follow

Back to top

Step 1 - Declaring and Defining Constants

Before we start declaring our class, we must first import all of the packages that we will be using. The second set of imports are of interest. The **com.apple.mrj** packages are special **MRJToolkit** packages that give your application behaviors specific to the Macintosh. We will discuss these in more detail in Step 18 - Registering Special MRJ Handlers.

Our `SlideShow` class derives from java.awt.Frame. A frame is a top-level window (it is in fact derived from the `window` class) that has a border and title bar. We use a frame instead of a borderless window because we want to have a title bar for our window.

```
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.KeyEvent;
import java.io.File;
import java.util.Vector;

import com.apple.mrj.MRJApplicationUtils;
import com.apple.mrj.MRJOpenDocumentHandler;
import com.apple.mrj.MRJQuitHandler;
import com.apple.mrj.MRJAboutHandler;

public class SlideShow extends Frame
{
    //Declare and define constants
    //Insert "SlideShow Constants"
```

Locate the SlideShow Constants clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.KeyEvent;
import java.io.File;
import java.util.Vector;

import com.apple.mrj.MRJApplicationUtils;
import com.apple.mrj.MRJOpenDocumentHandler;
import com.apple.mrj.MRJQuitHandler;
import com.apple.mrj.MRJAboutHandler;

public class SlideShow extends Frame
{
    //Declare and define constants
    //Insert "SlideShow Constants"
    protected static final int SLEEP_DELAY = 1000;
    protected static final int WIDTH = 430;
    protected static final int HEIGHT = 270;
```

We declare three constants for our class. The first, `SLEEP_DELAY`, is the number of milliseconds to pause between slides before going to the next. The number 1000 corresponds to 1 second. The `WIDTH` constant is the default width of the slide viewing area while `HEIGHT` is the height of the area.

Now we will declare the class data members.

Back to top

Step 2 - Declaring the Slide Show Data Members

We will now declare a number of data members to store internal state such as whether or not we are running in full screen, mode, and for our list of images we are displaying.

```
    protected static final int WIDTH = 430;
    protected static final int HEIGHT = 270;
    //Declare data members
    //Insert "SlideShow data members"
```

Locate the SlideShow data members clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
    protected static final int WIDTH = 430;
    protected static final int HEIGHT = 270;

    //Declare data members
    //Insert "SlideShow data members"
```

```
    protected Vector files;
    protected Image currentImage;
    protected int currentIndex;
    protected Boolean isFullScreen;
    protected Boolean isPlaying;
    protected PlayRunnable playRunnable;
    protected Thread thread;
    protected FileDialog openFileDialog1;
    protected Controller controls;
    protected AboutDialog aboutDialog1;
```

We declare a number of data members that we will use to store internal data. All are declared protected so that derived classes have access to them, but clients may only access the data through pre-defined data accessor routines. This is good coding practice and prevents users from causing problems by directly accessing this data. Since there are a lot of variables here, we will examine them one at a time.

```
protected Vector files;
```

First, we declare a vector to store our image files that will be successively displayed as the slide show. This vector is similar to the vector of images used by the button, but instead of button images, this vector holds the set of slides to be displayed.

```
protected Image currentImage;
protected int currentIndex;
```

Next, we declare an image variable that stores the image currently being displayed, and an integer that stores the index in the vector of the image being displayed.

```
protected Boolean isFullScreen;
protected Boolean isPlaying;
```

We declare two Booleans to keep track of whether we are currently in full-screen mode, and one to keep track of whether the slide show is playing. In full-screen mode, we black out the desktop, and center the images in that area. If the image is larger than the screen, it is scaled to fit.

```
protected PlayRunnable playRunnable;
```

This is our `Runnable` object which contains the body of our thread. We use this to automatically progress through the slides when the user clicks the play button, or chooses the play menu item.

```
protected Thread thread;
```

This is the thread we use to execute our `PlayRunnable` from.

```
protected FileDialog openFileDialog1;
```

We declare a file dialog for creation and storage of our file management object. Our `FileDialog` is used to facilitate the selection of images to be part of the slide show using the standard file mechanism. If you are using MacOS 8.5 or later, this dialog will use Navigation services.

```
protected Controller controls;
```

Our controller variable is used to store an instance of our `Controller` class which contains the buttons and user interface for our slide show. The controller sends action events to our slide show which we will respond to based on the message.

```
protected AboutDialog aboutDialog1;
```

Lastly, we declare an instance of our `AboutDialog` class from which we will instantiate our about box if the user chooses **About SlideShow** from the Apple Menu.

Now that we have our data members, it is time to declare our menu items.

Back to top

Step 3 - Declaring our application Menus

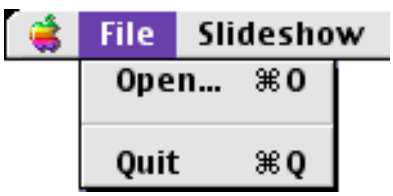
The application defines a series of menus and menu items that may be used to add images to the slide show and configure the viewer.

```
    protected Controller controls;
    protected AboutDialog aboutDialog1;
    //DECLARE_MENUS
    //Declare Menus, Menu Items and the Menu Bar
    //Insert "SlideShow declare menus"
```

Locate the SlideShow declare menus clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

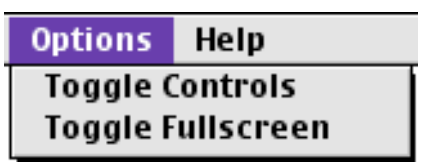
```
    protected Controller controls;
    protected AboutDialog aboutDialog1;

    //DECLARE_MENUS
    //Declare Menus, Menu Items and the Menu Bar
    //Insert "SlideShow declare menus"
    protected MenuBar menuBar1;
    protected Menu fileMenu;
    protected MenuItem openMenuItem;
    protected MenuItem quitMenuItem;
    protected Menu slideShowMenu;
    protected MenuItem playMenuItem;
    protected MenuItem backMenuItem;
    protected MenuItem forwardMenuItem;
    protected Menu optionsMenu;
    protected MenuItem controlsMenuItem;
    protected MenuItem fullScreenMenuItem;
```

We are declaring our menus and menu items that will be used in our application. The first object is our main menu bar object which will hold all of our menus.



The next three items are the **File** menu, the **Open** item of the **File** menu, and the **Quit** item of the **File** menu. This menu will look like the image above. When we build the menu, we will be placing a separator between the Open and Quit items. This will be covered in Step 7 - Initializing the Application Menus.

Next, we declare the **Slideshow** menu and its three menu items: **Toggle Play**, **Back**, and **Forward**. This menu is pictured above. We will be adding the Command Key equivalents to the menu items in Step 7 - Initializing the Application Menus.



Lastly, we declare three variables for the final menu, the **Options** menu and its two menu items, **Toggle Controls**, and **Toggle Fullscreen**.

Now that we have all of our class data members declared, it is time to work on our `main( )` routine,

the main entry point for our application.

Back to top

Step 4 - Creating the main entry point for the application

The main entry point of an application is the initial execution point. It is the routine that gets called first when the application is run. In our `main( )` routine, we will create our `slideshow` class, make our frame visible, and register some special MRJ-specific handlers that will provide a more Mac-like experience for the user.

```
/**
 * The entry point to our application
 */
static public void main(String args[])
{
    //Instantiate our SlideShow, make it visible, and register
    //our MRJ handlers.
    //Insert "SlideShow main"
```

Locate the SlideShow main clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * The entry point to our application
 */
static public void main(String args[])
{
    //Instantiate our SlideShow, make it visible, and register
    //our MRJ handlers.
    //Insert "SlideShow main"
    SlideShow slideShow = new SlideShow( );
    slideShow.setVisible(true);
    slideShow.registerMRJHandlers( );
}
```

The first step in our main routine is to create an instance of our `SlideShow` object. Once this object is created in our constructor, we call `setVisible( )` with a Boolean parameter to make the frame visible. Finally we call our `registerMRJHandlers( )` method. This method installs a handler for handling AppleEvents such as a `QuitApplication` AppleEvent or an `OpenDocument` AppleEvent. We will talk about this routine in more detail in Step 18 - Registering Special MRJ Handlers .

Next we will implement the `SlideShow` constructor.

Back to top

Step 5 - Initializing the SlideShow's State

The constructor of the `SlideShow` has several important functions such as creating the application menus, initializing the state, initializing controls, and registering listeners. The first step we will perform is initializing the application's state.

```
public SlideShow( )
{
    //INIT_STATE
    //Initialize state information
    //Insert "SlideShow init state"
```

Locate the SlideShow init state clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public SlideShow( )
{
    //INIT_STATE
    //Initialize state information
    //Insert "SlideShow init state"
    isFullScreen = false;
    isPlaying = false;
    files = new Vector( );
    currentImage = null;
    currentIndex = -1;
```

We initialize several of the data members to give them initial values. When we first start up, we don't want to be full screen, so we set `isFullScreen` to `false`, and we don't want to be playing initially, so we set `isPlaying` to `false` as well.

We create a new vector object and store it in our files variable. This vector will keep track of the images in our slideshow. We use a vector object for this storage because that allows us to support an arbitrary number of images since the data structure is dynamically resizable.

Since we don't have any initial images, we set the `currentImage` variable to `null` **and** the `currentIndex` variable to -1. We choose -1 because it is not a valid vector index and also because it is a recognizable value. Any attempt to use it would throw an exception that we would easily be able to track down to this initialization. If we would have used a 0 or 1, this would be harder to debug since this could be a valid vector index.

Next we will set up and initialize our controls.

Back to top

Step 6 - Setting up and Initializing our Application Controls

Our application contains a number of controls and objects that need to be initialized in our constructor.

```
    isFullScreen = false;
    isPlaying = false;
    files = new Vector( );
    currentImage = null;
    currentIndex = -1;
    //INIT_CONTROLS
    //Setup and configure our SlideShow
    //Insert "SlideShow init controls"
```

Locate the SlideShow init controls clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
    isFullScreen = false;
    isPlaying = false;
    files = new Vector( );
    currentImage = null;
    currentIndex = -1;

    //INIT_CONTROLS
    //Setup and configure our SlideShow
    //Insert "SlideShow init controls"
    setLayout(null);
    setVisible(false);
    setSize(WIDTH, HEIGHT);
    Dimension screenSize =
       Toolkit.getDefaultToolkit( ).getScreenSize( );
    setLocation((screenSize.width - WIDTH) / 2,
       (screenSize.height - HEIGHT) / 2);
    setBackground(Color.black);
    openFileDialog1 = new FileDialog(this);
    openFileDialog1.setMode(FileDialog.LOAD);
    openFileDialog1.setTitle("Open");
    openFileDialog1.setFilenameFilter(new ImageNameFilter( ));
    setTitle("SlideShow");
    controls = new Controller(this);
```

Let’s look at these initialization statements line by line in order to better understand what they do.

`setLayout(null);`

We do not want to use a layout manager for our `SlideShow` window because we will be drawing the contents of the window ourselves. Thus, we set the layout manager to `null` in order to specify no layout.

`setVisible(false);`

We hide the window during its construction by calling `setVisible( )` with `false` as the

parameter. This is good practice, as we don't want the user to see the window pieces being constructed.

```
setSize(WIDTH, HEIGHT);
```

We set the initial size of our window be 430 pixels wide and 270 pixels tall. The `WIDTH` and `HEIGHT` values are the constants we defined previously.

```
Dimension screenSize = Toolkit.getDefaultToolkit( ).getScreenSize( );
```

We need to get the width and height of the screen that we are displaying on so that we can center our window, or resize the window to the size of the screen if we are in full screen mode. To get the screen size, we use a method in the java.awt.Toolkit class. The toolkit is a mechanism that gives us access to the underlying java peer classes, or native classes on which the awt is built. For example, in the case of `java.awt.Window`, there is a peer class that uses the native system call on the platform to create that window. On a Macintosh, the peer class for window uses the Macintosh Toolbox and Window Manager to create the window that corresponds to the Java object. On Windows, the peer class uses the MFC and the Windows API to create a native window.

By retrieving the default toolkit, we have access to platform specific information such as the call to getScreenSize( ) which uses a native call to retrieve the dimension of our screen.

```
setLocation((screenSize.width - WIDTH) / 2, (screenSize.height -
HEIGHT) / 2);
```

We use some basic math to center the window on the screen. The `setLocation( )` call, positions the top left-hand corner of the window based on the parameters which are offsets from the top and left corner of the screen.

```
setBackground(Color.black);
```

We want the window to have a black fill color, so we set the background to black.

```
openFileDialog1 = new FileDialog(this);
openFileDialog1.setMode(FileDialog.LOAD);
openFileDialog1.setTitle("Open");
openFileDialog1.setFilenameFilter(new ImageNameFilter( ));
```

Our application uses a file dialog to allow the user to specify the images to be used as part of the slide show. We create a new FileDialog object and assign it to our variable. A file dialog can be used to either open files, or save files. The function of the dialog is specified by setting the mode with the `setMode()` call. We want to use our dialog to open files, so we call `setMode( )` with `FileDialog.LOAD` as the parameter. Next, we set the title of the dialog to "Open", and specify a file filter to be used by the dialog. Our file filter will look at the names of the files and only display those names which end in a ".gif" or ".jpg" extension. We will look at this filter in detail when we examine the ImageNameFilter file.

```
setTitle("SlideShow");
```

We set the title of the window to "SlideShow" for lack of a better name.

```
controls = new Controller(this);
```

Lastly, we create our compound controller object that contains all of the buttons and controls for

controlling our application. We pass ourselves as the parent frame parameter to the constructor of the Controller.

Next, we will initialize our application menus.

Back to top

Step 7 - Initializing the Application Menus

We are now ready to create our application menu bar and associated menus and menu items.

```
    openFileDialog1.setTitle("Open");
    openFileDialog1.setFilenameFilter(new ImageNameFilter( ));
    setTitle("SlideShow");
    controls = new Controller(this);
    //INIT_MENUS
    //Create, configure, and setup the menubar,
    //menus, and menu items.
    //Insert "SlideShow init menus"
```

Locate the SlideShow init menus clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

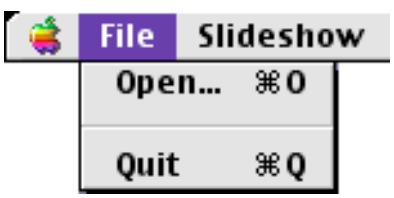
```
    openFileDialog1.setTitle("Open");
    openFileDialog1.setFilenameFilter(new ImageNameFilter( ));
    setTitle("SlideShow");
    controls = new Controller(this);

    //INIT_MENUS
    //Create, configure, and setup the menubar,
    //menus, and menu items.
    //Insert "SlideShow init menus"
    menuBar1 = new MenuBar( );

    //File menu
    fileMenu = new Menu("File");
    openMenuItem = new MenuItem("Open...");
    openMenuItem.setShortcut(new MenuShortcut
                             (KeyEvent.VK_O,false));
    fileMenu.add(openMenuItem);
    fileMenu.addSeparator( );
    quitMenuItem = new MenuItem("Quit");
    quitMenuItem.setShortcut(new MenuShortcut
                             (KeyEvent.VK_Q,false));
    fileMenu.add(quitMenuItem);
    menuBar1.add(fileMenu);

    //SlideShow menu
    slideShowMenu = new Menu("SlideShow");
```

```
    playMenuItem = new MenuItem("Toggle Play");
    playMenuItem.setShortcut(new MenuShortcut
                             (KeyEvent.VK_P,false));
    slideShowMenu.add(playMenuItem);
    backMenuItem = new MenuItem("Back");
    backMenuItem.setShortcut(new MenuShortcut
                             (KeyEvent.VK_OPEN_BRACKET,false));
    slideShowMenu.add(backMenuItem);
    forwardMenuItem = new MenuItem("Forward");
    forwardMenuItem.setShortcut(new MenuShortcut
                             (KeyEvent.VK_CLOSE_BRACKET,false));
    slideShowMenu.add(forwardMenuItem);
    menuBar1.add(slideShowMenu);

    //Options menu
    optionsMenu = new Menu("Options");
    controlsMenuItem = new MenuItem("Toggle Controls");
    optionsMenu.add(controlsMenuItem);
    fullScreenMenuItem = new MenuItem("Toggle Full Screen");
    optionsMenu.add(fullScreenMenuItem);
    menuBar1.add(optionsMenu);
    setMenuBar(menuBar1);
```

Since this is a lot of code, we will look at it in blocks. First, we will set up the main menu bar and **File** menu.

```
menuBar1 = new MenuBar( );
```

We create a main menu bar object and store it in our data member.




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

We create each of our menus by creating a new `java.awt.Menu` object with the string of the menu name. In this case, we create the **File** menu by using the string “File”, and assigning the result to our local data member.

```
openMenuItem = new MenuItem("Open...");
```

Menu items are created similarly. For every item we want to create, we create a new instance of java.awt.MenuItem with the string for the item as we want it displayed in the menu. Here we use the string “Open…”.

```
openMenuItem.setShortcut(new MenuShortcut (KeyEvent.VK_O,false));
```

To make our application more user friendly, we assign menu shortcuts to our menu items. These shortcuts translate to Command keys on the Macintosh, and accelerator keys on Windows platforms. To make a menu shortcut, we call `setShortcut( )` from the specific menu item, and pass a java.awt.MenuShortcut object which we are creating in place. The menu shortcut constructor takes a virtual key code that specifies the key for the shortcut and a Boolean which specifies whether the Shift key needs to be pressed or not. In our case, we use the Macintosh standard Command-O for our **Open** menu item, and we pass `false` to specify that the user does not need to hold down Shift-Command-O to perform the shortcut.

```
fileMenu.add(openMenuItem);
fileMenu.addSeparator( );
```

We add our newly created open menu item to our **File** menu and add a separator. This is a horizontal line that will appear between our open item and the Quit item.

```
quitMenuItem = new MenuItem("Quit");
quitMenuItem.setShortcut(new MenuShortcut (KeyEvent.VK_Q,false));
fileMenu.add(quitMenuItem);
menuBar1.add(fileMenu);
```

Now that we have walked through the **Open** menu item, it is a lot easier to understand the code for the **Quit** menu item. We create a new menu item and assign it to our `quitMenuItem` data member. We add a shortcut (Command-Q) for the item, and add the item to the **File** menu. Lastly, we add the completed **File** menu to our main menu bar. Menu items are added from the top to the bottom to menus, and menus are added from left to right in the menu bar.

The remaining two items are very straightforward.

```
slideShowMenu = new Menu("SlideShow");
playMenuItem = new MenuItem("Toggle Play");
playMenuItem.setShortcut(new MenuShortcut (KeyEvent.VK_P,false));
slideShowMenu.add(playMenuItem);
backMenuItem = new MenuItem("Back");
backMenuItem.setShortcut(new MenuShortcut
(KeyEvent.VK_OPEN_BRACKET,false));
slideShowMenu.add(backMenuItem);
forwardMenuItem = new MenuItem("Forward");
forwardMenuItem.setShortcut(new MenuShortcut
(KeyEvent.VK_CLOSE_BRACKET,false));
slideShowMenu.add(forwardMenuItem);
menuBar1.add(slideShowMenu);
```

The code above and the image of the menu the code represents should be the same. Yet, if you look

carefully, you will notice a mistake in the image. Can you find it?

We create a new **SlideShow** menu and a new **Toggle Play** item. We assign Command-P as the shortcut, and add the menu item to the menu. Next we create a **Back** and a **Forward** item with shortcut keys and add them to the menu. Finally, we add the **Slideshow** menu to the menu bar.


```
optionsMenu = new Menu("Options");
controlsMenuItem = new MenuItem("Toggle Controls");
optionsMenu.add(controlsMenuItem);
fullScreenMenuItem = new MenuItem("Toggle Full Screen");
optionsMenu.add(fullScreenMenuItem);
menuBar1.add(optionsMenu);
setMenuBar(menuBar1);
```

For the options menu, we create a new menu and the appropriate menu items. We do not assign command keys here because these items should not need be used frequently. Lastly, we call `setMenuBar( )` which changes the main menu bar to the one we just created.

Now we are ready for the last step in creating our constructor— registering our application event listeners.

Back to top

Step 8 - Registering our Application Event Listeners

Our application needs to install a number of event listeners in order to respond appropriately when events are fired. Not only do we need to listen to our controller and respond when it fires action events, but we also have to listen to all of our menu items so that we can respond when these items are selected from the menus.

```
    fullScreenMenuItem = new MenuItem("Toggle Full Screen");
    optionsMenu.add(fullScreenMenuItem);
    menuBar1.add(optionsMenu);
    setMenuBar(menuBar1);

    //REGISTER_LISTENERS
    //Register ActionListeners with the menu items and
    //the controller.
    //Insert "SlideShow register listeners"
```

Locate the SlideShow register listeners clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
    fullScreenMenuItem = new MenuItem("Toggle Full Screen");
    optionsMenu.add(fullScreenMenuItem);
    menuBar1.add(optionsMenu);
    setMenuBar(menuBar1);

    //REGISTER_LISTENERS
    //Register ActionListeners with the menu items and
    //the controller.
    //Insert "SlideShow register listeners"
    Action aAction = new Action( );
    openMenuItem.addActionListener(aAction);
    quitMenuItem.addActionListener(aAction);
    controlsMenuItem.addActionListener(aAction);
    fullScreenMenuItem.addActionListener(aAction);
    playMenuItem.addActionListener(aAction);
    backMenuItem.addActionListener(aAction);
    forwardMenuItem.addActionListener(aAction);
    controls.addActionListener(aAction);
}
```

This system of registering listeners for handling `ActionEvents` should be fairly familiar by now. As a result, we will talk about this code at a fairly high level. If you are having difficulty understanding, refer to our listener registration code in a previous class (such as in Controller.java).

First, we instantiate an instance of our inner class (that we will define later in Step 19 - Creating an Inner Class to Handle Action Events) that we will use to handle these event. Then we add an action listener to each class we want to be able to respond to. It is important to note that we register a listener with each menu item as well as our controller object.

Next, we will look at threading in our application and write our main thread class.

Back to top

Step 9 - Implementing the SlideShow Threading Model

In our application, we use a thread to handle the automatic advancement of frames in our slideshow. By using threading for timing and displaying subsequent images, we insure that the user interface of our application remains responsive.

We need to skip down in the source file a bit past the `togglePlaying( )` method to get to the definition of our inner thread class.

```
public void togglePlaying( )
{
    //Handle starting and stopping the automatic progression of
    //the show.
    //Insert "SlideShow togglePlaying"
}
```

```
//Inner class to implement our automatic progression of
//the show.
//Insert "SlideShow PlayRunnable"
```

Locate the SlideShow PlayRunnable clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public void togglePlaying( )
{
    //Handle starting and stopping the automatic progression of
    //the show.
    //Insert "SlideShow togglePlaying"
}

//Inner class to implement our automatic progression of
//the show.
//Insert "SlideShow PlayRunnable"
class PlayRunnable implements Runnable
{
    public Boolean isRun = true;

    public void run( )
    {
        while (isRun)
        {
            oneStep(true);

            try
            {
                Thread.sleep(SLEEP_DELAY);
            }
            catch (InterruptedException exc) { }
         }
     }
}
```

This code creates an inner class that implements the Runnable interface. The runnable interface specifies an API for simple thread classes that have a single method, the `run( )` method where the main work of the thread is performed. When the thread is started, the `run` method is called. Once execution of the `run` method is completed, the thread is no longer running. The thread still exists, but is no longer alive. We want our thread to continue running as long as we are in play mode. As a result, we have a while loop in our `run` method that executes as long as `isRun` is set to `true`.

Let’s study the code in detail. First, we declare our inner class that implements the `Runnable` interface. Secondly, we declare a public Boolean data member is run that we initialize to `true`. We

make this data member public, so we can access it from our application class. Next, we declare our `run` method. In the body, we call `oneStep( )` which displays the next image in the slide show. (We will look at this method in detail in Step 11 - Implementing oneStep( )). Lastly, we try to sleep the thread (or perform no operations for our delay interval which we defined to be one second. If for some reason, we can't sleep our thread because of a InterruptedException, we silently catch the exception and continue our loop.

Now we will go back and define our `togglePlaying( )` method.

Back to top

Step 10 - Implementing togglePlaying( )

The `togglePlaying( )` method is called when the user clicks on the play/pause button of the controller. If the application is in play mode, we need to stop playing by stopping the thread. If the application is in pause mode, we need to create a new `PlayRunnable` thread and start it.

Skip back in the source just above the `PlayRunnable` inner class we just created.

```
        forwardMenuItem.addActionListener(aAction);
        controls.addActionListener(aAction);
    }

    /**
     * Starts or stops cycling forward through the list of image
     * files to display.
     */
    public void togglePlaying( )
    {
        //Handle starting and stopping the automatic progression
        //of the show.
        //Insert "SlideShow togglePlaying"
```

Locate the SlideShow togglePlaying clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
        forwardMenuItem.addActionListener(aAction);
        controls.addActionListener(aAction);
    }

    /**
     * Starts or stops cycling forward through the list of image
     * files to display.
     */
    public void togglePlaying( )
    {
        //Handle starting and stopping the automatic progression of
        //the show.
        //Insert "SlideShow togglePlaying"
```

```
    if (isPlaying)
    {
        if (playRunnable != null)
            playRunnable.isRun = false;
        isPlaying = false;
    }
    else
    {
        if (thread == null || !thread.isAlive( ))
        {
            if (playRunnable != null)
                playRunnable.isRun = false;

            playRunnable = new PlayRunnable( );
            thread = new Thread(playRunnable);
            thread.start( );
            isPlaying = true;
        }
    }
}
```

This code for `togglePlaying( )` is fairly straightforward. We check the `isPlaying` variable (which is initially set to `false`). If the variable is true, meaning we are in play mode, we need to stop playing. To do so, we check to make sure our `playRunnable` thread exists (i.e., is non-null). If it has been created, we set the `isRun` data member of the thread to false. This will cause the while loop in our thread to stop executing, which causes our thread to stop. Lastly, we set our application data member `isPlaying` to `false` since we are now stopped.

If `isPlaying` is false when we enter this function, we are stopped and want to toggle our state to the play mode. Therefore, we check to see if our thread is null (which will be the case if we are entering `togglePlay` for the first time), and also check to make sure that the thread is not alive by calling `thread.isAlive( )`. If the `run( )` method of our `playRunnable` thread is executing, `isAlive( )` will return true. Otherwise, it will return false. What we are doing here is checking to see if our thread exists and if it does, make sure that it is not alive. If the thread is alive and non-null, we set the `isRun` variable to `false`, which will cause the thread to die. Now that we are assured that our thread is no longer running, we create a new thread and assign it to our data member and pass it our new runnable object. Lastly, we start the thread and set isPlaying to true.

Now that we have our threading set up, let's look at our `oneStep( )` method where we will add code to handle slide advancement.

Back to top

Step 11 - Implementing the oneStep( ) method

Our `oneStep( )` method is called from the `playRunnable` object running in our thread. It is responsible for advancing to the next slide in the image list and displaying the image. If we reach the end of the image list, it loops to the first slide. This method is also going to be called in response to

clicking the forward and backwards button in the controller. As a result, our routine takes a Boolean parameter that specifies whether we are advancing, or going backwards. A value of `true` means that we are going forward, while a value of `false` means that we are stepping backwards to the previous image. Let's go down past our `playRunnable` inner class to the declaration for our oneStep( ) method.

```
            try
            {
                Thread.sleep(SLEEP_DELAY);
            }
            catch (InterruptedException exc) { }
        }
    }
}

/**
 * Steps the slide show forward or backwards by one image.
 * @param if true, step forward, if false, step backward.
 */
public void oneStep(Boolean isForward)
{
    //Handle stepping forward or backward in the list of
    //image files, load the image, and repainting.
    //Insert "SlideShow oneStep"
```

Locate the SlideShow oneStep clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
            try
            {
                Thread.sleep(SLEEP_DELAY);
            }
            catch (InterruptedException exc) { }
        }
    }
}

/**
 * Steps the slide show forward or backwards by one image.
 * @param if true, step forward, if false, step backward.
 */
public void oneStep(Boolean isForward)
{
    //Handle stepping forward or backward in the list of
    //image files, load the image, and repainting.
    //Insert "SlideShow oneStep"
```

```
    int size = files.size( );

    if (size > 0)
    {
        if (isForward)
        {
            currentIndex++;
            if (currentIndex >= size)
                currentIndex = 0;
        }
        else
        {
            currentIndex--;
            if (currentIndex < 0)
                currentIndex = size - 1;
        }

        File file = (File)files.elementAt(currentIndex);
        if (file != null)
        {
            Image image = Misc.loadImage(file.getPath( ), this,
                          false);
            if (image != null)
            {
                if (currentImage != null)
                    currentImage.flush( );
                currentImage = image;
                repaint( );
            }
        }
    }
}
```

This looks like a lot of code, but it is not as complicated as it may seem. Our first priority is to get the number of images that will be displayed as part of our `slideShow`. This is accomplished by checking our vector of image files and retrieving the size (the number of elements in the vector). We cache this in a local variable because we will need this number throughout this function.

The next line checks to see if we have more than zero images. If we don't, then we return, since the concept of going to the next image is meaningless if there are no images. Otherwise, we check our Boolean parameter to determine if we need to step forward or backwards. If we are going forwards (if `isForward` is true), we increment our index variable. Then if the index exceeds our image capacity (which means that we were on the last image in our show) we set the index to the beginning by setting it to 0.

Otherwise (if we are going backwards), we do a similar thing but we decrement our index and check to

see if we are at the first image (index zero) and wrap to the last image if that is the case.

Once we have determined which image will be displayed next, we retrieve a file reference from the vector of files and store it in a local variable. After checking to make sure that the file is not null, we load the image, flush the old image, and set the current image to be the image we just loaded.

Finally, we call `repaint( )` on our window so that our `paint( )` method is called and our new image is drawn in our window. We will examine `paint( )` in detail in Step 16 - Implementing the paint( ) method, but for now, all you need to know is that the paint method draws the image to the screen. Now lets look at our code for the `toggleFullScreen( )` method.

Back to top

Step 12 - Implementing the toggleFullScreen( ) method

When the user selects Toggle Fullscreen from the Options menu, it calls the `toggleFullScreen( )` method which is responsible for changing the full-screen mode.

```
/**
 * Handles sizing of the window to utilize the full screen size,
 * or to use the size of the image.
 */
public void toggleFullScreen( )
{
    //Handle toggling the frame window size between the image
    //size, screen size, and full screen.
    //Insert "SlideShow toggleFullScreen"
```

Locate the SlideShow toggleFullScreen clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Handles sizing of the window to utilize the full screen size,
 * or to use the size of the image.
 */
public void toggleFullScreen( )
{
    //Handle toggling the frame window size between the image
    //size, screen size, and full screen.
    //Insert "SlideShow toggleFullScreen"
```

```
    Dimension screenSize =
         Toolkit.getDefaultToolkit( ).getScreenSize( );

    if (isFullScreen)
    {
        int width = WIDTH;
        int height = HEIGHT;

        if (currentImage != null)
        {
            width = currentImage.getWidth(this);
            height = currentImage.getHeight(this);

            //Make sure the window fits on the screen
            width = Math.min(width, screenSize.width);
            height = Math.min(height, screenSize.height);
        }

        setLocation((screenSize.width - width) / 2,
                    (screenSize.height - height) / 2);
        setSize(width, height);

        isFullScreen = false;
    }
    else
    {
        int top = 21;
        int sides = 5;
        setBounds(-sides, -top, screenSize.width + 2 * sides,
                  screenSize.height + top + sides);
        isFullScreen = true;
    }
}
```

This method looks bad, but it is mostly just math. After we get the screen size from the toolkit, there are two main branches of execution. The first occurs if we are in full-screen mode and want to go to normal mode, and the second is if we are in normal mode and want to change to full screen. Let’s look at the first case.

Since we are in full screen and want to go to normal mode, we first set up two variables initialized to our default width and height. These default values will be used if for some reason our image is null. Next, if the image is not null, we retrieve its width and height and store them in our local variables. Then we assign width and height to the smaller of either the screen size for that dimension, or the image size. This insures that if the image is larger than the screen, that we make our window the size of the screen instead. If the image is smaller, we use the image size. This is accomplished by our judicious use of the min function that is part of the standard Math package.

Next, we set the position of the window so that it is centered on the screen, and set the width and height of the window. Finally we set the `isFullScreen` variable to `false`.

In the case where we are in normal display mode and want to go full screen, we set the bounds of the window to the screen size and then set `isFullScreen` to true. Note that since our window has a frame and a title bar, we had to correct for the height and width of this border. That is where the top and sides value comes in. We want to make sure that we move the window 21 pixels above the top of the screen so that we don't see the title bar. We do the same with the edges to ensure that the edge border is not visible.

Now it's time for the ever-popular `toggleControls( )`.

Back to top

Step 13 - Implementing toggleControls( )

The `toggleControls` method is called from the Toggle Controls menu item of the Options menu. It will show the controller window if it is hidden, and hide the window if it is visible.

```
/**
 * Shows or hides the control window.
 */
public void toggleControls( )
{
    //Handle toggling the visibility of the controller
    //Insert "SlideShow toggleControls"
```

Locate the SlideShow toggleFullScreen clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Shows or hides the control window.
 */
public void toggleControls( )
{
    //Handle toggling the visibility of the controller
    //Insert "SlideShow toggleControls"
    if (controls != null)
        controls.setVisible(!controls.isVisible( ));
}
```

What we do here is first make sure that our controller is not null. If it is, then all bets are off and we shouldn't do anything. Otherwise, we show the controller if it is hidden, and hide the controller if it is visible by setting the visibility of the object to the logical opposite of its current visibility state. Pretty slick.

For the next step, we will implement our quit handler.

Back to top

Step 14 - Implementing doOnQuit( )

Our `doOnQuit( )` method is called when the user selects the **Quit** item from the **File** menu or we receive a `QuitApplication AppleEvent`. More on this in Step 18 - Registering Special MRJ Handlers.

```
/**
 * Gets called when the user chooses the Quit menu item, or
 * when the application receives a quit message from the Finder
 * (or other app).
 */
protected void doOnQuit( )
{
    //Handle cleaning up, and quit.
    //Insert "SlideShow doOnQuit"
    //Do any clean up here.
```

Locate the SlideShow doOnQuit clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Gets called when the user chooses the Quit menu item, or
 * when the application receives a quit message from the Finder
 * (or other app).
 */
protected void doOnQuit( )
{
    //Handle cleaning up, and quit.
    //Insert "SlideShow doOnQuit"
    //Do any clean up here.
    //Exit with success.
    System.exit(0);
}
```

When we receive a quit message, we call java.lang.System.exit( ). This routine is very similar to the C call, `ExitToShell( )`. It shuts down the application and terminates the Virtual machine. The parameter is the message that would be returned on a command-line based system. We pass zero as the parameter to indicate that we exited because the user quit, not because of an error condition.

Now we will implement `doAbout( )` which displays our about dialog box.

Back to top

Step 15 - Implementing doAbout( )

The `doAbout` routine is called when the user selects **About SlideShow** from the **Apple** menu. It displays our About dialog (see image above) by instantiating our `AboutDialog` class. We use the MRJ toolkit to place the About item on the Apple menu. Normally, the Apple menu would not be accessible in Java (since it is platform specific), but we use some Macintosh-specific routines to give our users a more Mac-like experience. We will discuss the steps that we took to do this later in Step 18 - Registering Special MRJ Handlers.

```
/**
 * Gets called when the user selects the about menu item in
 * the Apple Menu.
 */
protected void doAbout( )
{
    //Handle displaying about information here
    //Insert "SlideShow doAbout"
```

Locate the SlideShow doAbout clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
/**
 * Gets called when the user selects the about menu item in
 * the Apple Menu.
 */
protected void doAbout( )
{
    //Handle displaying about information here
    //Insert "SlideShow doAbout"
    if (aboutDialog1 == null)
        aboutDialog1 = new AboutDialog(this, true);
    aboutDialog1.setVisible(true);
}
```

The `doAbout` method checks to see if the dialog is null (it will be if it has not previously been displayed). If the dialog is null, we create a new dialog passing our window as the parent frame. Then we display the dialog by calling `setVisible( )` with a true argument.

Roll up your sleeves, because it is time to implement `paint`.

Back to top

Step 16 - Implementing the paint( ) method

`Paint` is responsible for the drawing of our window. It handles drawing of the images as well as centering and scaling within the window. This extra work adds a bit of complexity, but makes the

application much nicer and polished.

```
public void paint(Graphics g)
{
    //Handle scaling and drawing the image to fit in the
    //frame content area.
    //Insert "SlideShow paint"
```

Locate the SlideShow paint clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public void paint(Graphics g)
{
    //Handle scaling and drawing the image to fit in the
    //frame content area.
    //Insert "SlideShow paint"
     if (currentImage != null)
     {
         Dimension s = getSize( );
         int iWidth = currentImage.getWidth(this);
         int iHeight = currentImage.getHeight(this);

         int scaleWidth = iWidth;
         int scaleHeight = iHeight;

         int wDelta = s.width - iWidth;
         int hDelta = s.height - iHeight;

         if (wDelta > 0 && hDelta > 0)
         {
             //The image fits, just draw it.
             g.drawImage(currentImage, (s.width - iWidth) / 2,
                  (s.height - iHeight) / 2, this);
         }
         else
         {
             //The image doesn't fit. We need to scale it
             //down to fit.

             float ratio = 1;

             if (wDelta < hDelta)
             {
                 if (iWidth > 0)
                 {
                     //width needs to be scaled to fit
                     ratio = s.width / (float)iWidth;
```

```
                }
            }
            else
            {
                if (iHeight > 0)
                {
                    //height needs to be scaled to fit
                    ratio = s.height / (float)iHeight;
                }
            }

            scaleWidth = (int)(iWidth * ratio);
            scaleHeight = (int)(iHeight * ratio);

            g.drawImage(currentImage,
                   (s.width - scaleWidth) / 2,
                   (s.height - scaleHeight) / 2,
                    scaleWidth, scaleHeight, this);
        }
    }
}
```

Whoa Nellie! That's some funky math. Don't worry about it. Understanding the algorithm isn't as important as understanding what the `paint` method does

First, we make sure we have a non-null image. If the image is null, there is nothing to draw, and we are done. If the image is not null, we get the size of the image and check to see if it fits inside our window. If the image fits, we draw it centered within the window. If it does not fit, we scale the image to fit and then draw the scaled image in our window.

The main drawing task is done by `drawImage( )`. When we are called, we are passed in a graphic context to draw into, and we call java.awt.Graphics.drawImage( ) from this context. We pass the image object to draw, the X location, Y location, width, height, and ourselves as the observer. The observer is an object that will receive notification when the drawing of the image is complete. We pass ourselves as the observer, even though we don't do anything special when we are notified of completion.

Next, we will implement `setVisible( )`.

Back to top

Step 17 - Implementing setVisible( )

We override the `setVisible( )` method to ensure that if the main window is made visible, the controller is made visible as well. Conversely, if the main window is hidden, we want to hide the controller.

```
public void setVisible(Boolean b)
{
```

```
    //Make sure the controls are visible only when the
    //frame is visible.
    //Insert "SlideShow setVisible"
```

Locate the SlideShow setVisible clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
public void setVisible(Boolean b)
{
    //Make sure the controls are visible only when the
    //frame is visible.
    //Insert "SlideShow setVisible"
    super.setVisible(b);

    if (controls != null)
        controls.setVisible(b);
}
```

Since we are adding functionality to our `setVisible` method override, we call `setVisible( )` from our base class (`Frame`) to insure that we inherit the default behavior. Next, we check to see if the controls are null. If they aren't we set the visibility of the controller to match the visibility of our window.

In the next step, we will register handlers to add Macintosh-specific functionality to our application.

Back to top

Step 18 - Registering Special MRJ Handlers

Users expect their Macintosh applications to behave in a consistent way and are unwilling to accept "But this is a Java program" as an excuse for loss of functionality. As a result, we add three specific handlers for our application to add functionality, an open document handler, an about handler, and a quit handler. The open document handler allows us to receive notification when documents are dropped on our application icon (in the case of a `JBound` application). When this occurs, the Finder sends an `OpenDocument` event to our application which will call our registered handler. In our case, any image files dropped on our application icon should be added to our image list.

The second handler, the about handler, notifies us when the user chooses the about item in the **Apple** Menu. There is also some additional work in the form of a Macintosh resource file that needs to be completed in order for this to look just right. We will cover this step in Making a Double-clickable Application

The third and final handler, the quit handler, allows the application to respond to quit events from the Finder. A quit event is generated when the user selects **Shut Down** from the **Special** Menu in the Finder. All running applications are notified via the `Quit AppleEvent`. This gives the user a chance to save any open documents, clean up, etc., before the system shuts down. If we did not handle this, and our program was running, the computer would not be able to shut down because the Finder would be waiting for our application to terminate.

```
protected void registerMRJHandlers( )
{
    //Register MRJ handlers for open, about and quit.
    //Insert "SlideShow registerMRJHandlers"
```

Locate the SlideShow registerMRJHandlers clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
protected void registerMRJHandlers( )
{
    //Register MRJ handlers for open, about and quit.
    //Insert "SlideShow registerMRJHandlers"
    MRJI IMRJI = new MRJI( );
    MRJApplicationUtils.registerOpenDocumentHandler(IMRJI);
    MRJApplicationUtils.registerQuitHandler(IMRJI);
    MRJApplicationUtils.registerAboutHandler(IMRJI);
}
```

This code may look a little strange at first, but it will make more sense if we examine the next code block in tandem.

```
    MRJApplicationUtils.registerAboutHandler(IMRJI);
}
//Inner class defining the MRJ Interface
//Insert "SlideShow MRJI"
```

Locate the SlideShow MRJI clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
    MRJApplicationUtils.registerAboutHandler(IMRJI);
}

//Inner class defining the MRJ Interface
//Insert "SlideShow MRJI"
```

```
class MRJI implements MRJOpenDocumentHandler, MRJQuitHandler,
                      MRJAboutHandler
{
    /**
     * This gets called by MRJ for each file/folder dropped
     * onto the application.
     * If the file is a directory, this recurses through the
     * directory structure collecting image files.
     * @param the file to add to the list of image files to
     * display, or the File object to recurse to look for image
     * files to add to the list.
     */
     public void handleOpenFile(File file)
     {
         if(file != null)
         {
             if (file.isDirectory( ))
             {
                  String directory = file.getPath( );
                  if (!directory.endsWith("/"))
                       directory += "/";

                  String[] fileList = file.list( );
                  for (int fileInd = 0;
                           fileInd < fileList.length;
                           fileInd++)
                  {
                      this.handleOpenFile(new File(directory +
                          fileList[fileInd]));
                  }
              }
              else
              {
                  String upperCaseName =
                         file.getName( ).toUpperCase( );
                  if (upperCaseName.endsWith(".JPG") ||
                      upperCaseName.endsWith(".JPEG") ||
                     upperCaseName.endsWith(".gif"))
                  {
                      files.addElement(file);
                  }
               }
          }
       }

       /**
        * This gets called when the application receives a
```

```
         * quit event
         */
         public void handleQuit( )
         {
             doOnQuit( );
         }

        /**
         * This gets called when the About menu item in the
         * Apple Menu is selected.
         */
         public void handleAbout( )
         {
             doAbout( );
         }
    }
}
```

OK. Now that we have the inner class and the handler registration routine, we can look at both as a single unit, starting with the inner class.

We declare our inner class:

```
class MRJI implements MRJOpenDocumentHandler, MRJQuitHandler,
MRJAboutHandler
```

We need to implement three interfaces from the `MRJApplicationUtilities` package so that we can handle specific custom event types. By implementing these three interfaces, we provide three methods can be called by the MRJ in response to specific events: `handleOpenFile( )`, `handleQuit( )`, and `handleAbout( )`.

This organization is exactly the same as the one we use to handle events such as the `ActionEvent`. We make an inner class that implements an interface specifying the signature of the class to be called. In this case, instead of implementing `actionPerformed( )`, we are implementing three custom handlers. These handlers each call a routine to respond the particular event.

Let's look at the code in detail starting with our `MRJI` inner class:

```
class MRJI implements MRJOpenDocumentHandler, MRJQuitHandler,
MRJAboutHandler
```

Our class implements the `MRJOpenDocumentHandler` interface as well as the `MRJQuitHandler` and `MRJAboutHandler` interfaces. We need to implement these interfaces in order to handle these specific event types.

Let's start with the open document handler:

```
public void handleOpenFile(File file)
{
    if(file != null)
```

```
        if (file.isDirectory( ))
        {
```

Our `handleOpenFile( )` method takes a file object as the parameter. This is the file that the user dropped on our application. If multiple files were dropped, `handleOpenFile( )` will get called once for each individual file. We first check to see if the file is null. If it is, we don't do anything. Otherwise, we check to see if our file is a directory. If it is, we need to iterate through all of the files in that directory. If it is not a directory, then it is an individual file and we can deal with that directly.

Let's look at the directory case:

```
    String directory = file.getPath( );
    if (!directory.endsWith("/"))
        directory += "/";
    String[] fileList = file.list( );
    for (int fileInd = 0; fileInd < fileList.length; fileInd++)
    {
        this.handleOpenFile(new File(directory + fileList[fileInd]));
    }
}
```

Our first task is to get the path of the directory and store it in a local variable as a string. We accomplish this by calling `getPath( )` from the java.io.File class. Next, we append a slash ("/") character if the path does not end with one. Then we create an array of files that contains the list of files in the directory we are passed. Lastly, we loop through this array and call ourselves recursively with each file in the directory.

If we were not passed a directory, we execute the following code:

```
else
{
    String upperCaseName = file.getName( ).toUpperCase( );
    if (upperCaseName.endsWith(".JPG")
         || upperCaseName.endsWith(".JPEG")
         || upperCaseName.endsWith(".gif"))
    {
        files.addElement(file);
    }
}
```

We get the name of the file, convert it to uppercase, and store in a temporary variable. Then, we look at the file name, and only add it to our list of files if it ends with an appropriate file extension.

The other two handlers are very simple.

```
public void handleQuit( )
{
   doOnQuit( );
}
```

```
public void handleAbout( )
{
    doAbout( );
}
```

Both simply call the methods that we previously implemented that do the real work. Now let's look at our registration function:

```
protected void registerMRJHandlers( )
{
    MRJI IMRJI = new MRJI( );
    MRJApplicationUtils.registerOpenDocumentHandler(IMRJI);
    MRJApplicationUtils.registerQuitHandler(IMRJI);
    MRJApplicationUtils.registerAboutHandler(IMRJI);
}
```

This method which we call from our constructor registers our `MRJI` class with `MRJ` to be used as the handler for the `open`, `quit`, and `about` events. In our method, we create a new instance of our inner class handler. Then we call registration functions from the `MRJApplicationUtils` package with our inner class as a parameter. These methods are documented in the Adobe Acrobat file "About MRJ Toolkit," which is distributed as part of the MRJ SDK,

Next, we will begin to create our event handling architecture starting with the implementation of an inner class for handling `ActionEvents`.

Back to top

Step 19 - Creating a Inner Class to Handle ActionEvents

In this step, we will be creating our inner class that will be used to handle action events from the menus, and the controller.

```
    public void handleAbout( )
    {
        doAbout( );
    }
}

//Inner class for handling ActionEvents
//Insert "SlideShow Action"
```

Locate the SlideShow registerMRJHandlers clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
    public void handleAbout( )
    {
        doAbout( );
    }
}

//Inner class for handling ActionEvents
//Insert "SlideShow Action"
class Action implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        Object object = event.getSource( );
        if (object == openMenuItem)
            openMenuItem_ActionPerformed(event);
        else if (object == quitMenuItem)
            quitMenuItem_ActionEvent(event);
        else if (object == controlsMenuItem)
            controlsMenuItem_ActionPerformed(event);
        else if (object == fullScreenMenuItem)
            fullScreenMenuItem_ActionPerformed(event);
        else if (object == playMenuItem)
            playMenuItem_ActionPerformed(event);
        else if (object == backMenuItem)
            backMenuItem_ActionPerformed(event);
        else if (object == forwardMenuItem)
            forwardMenuItem_ActionPerformed(event);
        else if (object == controls)
            controls_ActionPerformed(event);
    }
}
```

Once again, this code should be familiar by now. Just like we have done in numerous other classes, our inner class implements the `ActionListener` interface. We override the `actionPerformed` method so that when we receive an action event, we can process it. The only difference is that we are handling events many different object types. Our class is a big if-then-else statement that compares the source of the event with a list of object types and then calls a specific method to handle the event if the types match. Most of these handlers are for menu items, except the last item, which handles events from the controller.

Let’s look at the implementation of the individual action performed methods for each class.

Back to top

Step 20 - Responding to ActionEvents

In our final step, we will implement all of the methods that handle the action events dispatched from our inner class event handler.

```
        else if (object == controls)
            controls_ActionPerformed(event);
    }
}
//Routines for handling the various ActionEvents
//Insert "SlideShow Action Management"
```

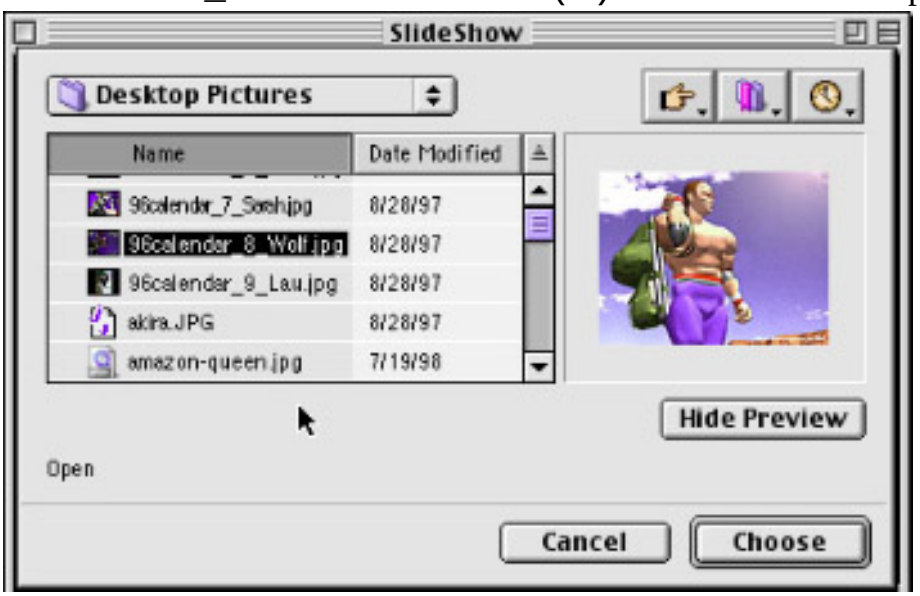
Locate the SlideShow Action Management clipping in the SlideShow folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
        else if (object == controls)
            controls_ActionPerformed(event);
    }
}

//Routines for handling the various ActionEvents
//Insert "SlideShow Action Management"
void openMenuItem_ActionPerformed(ActionEvent event)
{
    //Present the load file dialog.
    openFileDialog1.setVisible(true);

    //Make sure a valid value is returned (user could cancel).
    String resultPath = openFileDialog1.getDirectory( );
    String resultFile = openFileDialog1.getFile( );
    if(resultPath != null && resultPath.length( ) != 0
       && resultFile != null && resultFile.length( ) != 0)
    {
        //Construct a File object from the information from the
        //dialog.
        File file = new File(resultPath + resultFile);
        if(file != null)
        {
            //Add the file to our list of image files.
            files.addElement(file);

            //Load the image from the file, and display it as
            //our current image.
            Image image = Misc.loadImage(file.getPath( ), this,
                                         false);
            if (image != null)
            {
                if (currentImage != null)
                    currentImage.flush( );
                currentImage = image;
                currentIndex = files.size( ) - 1;
                repaint( );
```

```
            }
        }
    }
}

void quitMenuItem_ActionEvent(ActionEvent event)
{
    doOnQuit( );
}

void controlsMenuItem_ActionPerformed(ActionEvent event)
{
    toggleControls( );
}

void fullScreenMenuItem_ActionPerformed(ActionEvent event)
{
    toggleFullScreen( );
}

void playMenuItem_ActionPerformed(ActionEvent event)
{
    togglePlaying( );
    controls.setPlayState(!isPlaying);
}

void backMenuItem_ActionPerformed(ActionEvent event)
{
    oneStep(false);
}

void forwardMenuItem_ActionPerformed(ActionEvent event)
{
    oneStep(true);
}

void controls_ActionPerformed(ActionEvent event)
{
    String command = event.getActionCommand( );

    if (command.equals(Controller.BACKWARD_COMMAND))
        oneStep(false);
    else if (command.equals(Controller.FORWARD_COMMAND))
        oneStep(true);
    else if (command.equals(Controller.PLAY_COMMAND))
        togglePlaying( );
    else if (command.equals(Controller.PAUSE_COMMAND))
        togglePlaying( );
```

```
    }
}
```

The first method `openMenuItem_ActionPerformed( )` is called when the open item on the **File** menu is chosen.



The first step is to display our open file dialog.

```
openFileDialog1.setVisible(true);
```

This call does not return until the user clicks on either the **Choose** button (OK for systems that do not have Navigation Services installed) or **Cancel**.

Once the user exits the dialog, we get the file from the dialog and test it to see if it is valid:

```
String resultPath = openFileDialog1.getDirectory( );
String resultFile = openFileDialog1.getFile( );
if(resultPath != null && resultPath.length( ) != 0 && resultFile !=
null && resultFile.length( ) != 0)
```

We get the file and the directory from the file dialog as a string which we cache in temporary variables. Then we check to make sure that the strings are not empty or null. If the user clicked the **Cancel** button, these would be null, and we would then not attempt to open the file.

Otherwise, we create a new file object and check to make sure it is valid:

```
File file = new File(resultPath + resultFile);
if(file != null) { //Add the file to our list of image files.
    files.addElement(file);
```

Our new file object takes a string argument which is the concatenation of the directory and the file. We make sure the file is not null, and if it is valid, add the file to our internal file list.

Finally, we load the image from the file and display it:

```
Image image = Misc.loadImage(file.getPath( ), this, false);
```

```
if (image != null)
{
    if (currentImage != null)
        currentImage.flush( );
    currentImage = image;
    currentIndex = files.size( ) - 1;
    repaint( );
}
```

We load the image using our miscellaneous class and store a local reference. If we have a current image in memory (i.e., the image reference is not null), we flush it from memory. This removes the object from memory immediately. We could set the variable to null and wait for the garbage collector, but these images could be of potentially very large size, so we want to free up the memory immediately.

We set the current image to our newly loaded image, and set the current index variable to be to location of our picture which is placed at the end of the vector. We use `files.size( )` = -1 because the vector is zero based. To get the last item, we need to subtract one. We then call `repaint` so that our new image is drawn.

Our remaining handlers are much simpler. The next, `quitMenuItem_ActionEvent( )`, is called when the user selects **Quit** from the **File** menu.

```
void quitMenuItem_ActionEvent(ActionEvent event)
{
    doOnQuit( );
}
```

This method simply cause the `doOnQuit( )` method we previously implemented. The method `controlsMenuItem_ActionPerformed( )` is called when Toggle Controls is selected from the Options menu.

```
void controlsMenuItem_ActionPerformed(ActionEvent event)
{
    toggleControls( );
}
```

This method simply calls toggleControls( ). Next is fullScreenMenuItem_ActionPerformed( ) which is called when the user selects Toggle Fullscreen from the options menu.

```
void fullScreenMenuItem_ActionPerformed(ActionEvent event)
{
    toggleFullScreen( );
}
```

This method is simple as well. It calls `toggleFullScreen( )`. `PlayMenuItem_ActionPerformed( )` is called when the **Toggle Play** item is selected from

the **SlideShow** menu.

```
void playMenuItem_ActionPerformed(ActionEvent event)
{
    togglePlaying( );
    controls.setPlayState(!isPlaying);
}
```

This method calls `togglePlaying( )` and then notifies the controller that the play state has changed by calling `setPlayState` from the controller object. The method `backMenuItem_ActionPerformed( )` is called when the **Back** menu item is selected from the **SlideShow** menu.

```
void backMenuItem_ActionPerformed(ActionEvent event)
{
    oneStep(false);
}
```

Here, we call `oneStep( )` with `false` as the parameter. The value of `false` tells us to go backwards instead of forwards. Our next method, `forwardMenuItem_ActionPerformed( )` is very similar. It is called when the **Forward** menu item is selected from the **SlideShow** menu.

```
void forwardMenuItem_ActionPerformed(ActionEvent event)
{
    oneStep(true);
}
```

This method does the same thing as the back method, but we pass true to `oneStep( )` in order to go forward instead of backward. Our last method, `controls_ActionPerformed( ),` is called in response to any button in the controller being pressed.

```
void controls_ActionPerformed(ActionEvent event)
{
    String command = event.getActionCommand( );
    if (command.equals(Controller.BACKWARD_COMMAND))
        oneStep(false);
    else if (command.equals(Controller.FORWARD_COMMAND))
        oneStep(true);
    else if (command.equals(Controller.PLAY_COMMAND))
        togglePlaying( );
    else if (command.equals(Controller.PAUSE_COMMAND))
        togglePlaying( );
}
```

This method is a bit more complex because it has to handle the events from the controller and respond appropriately depending on the individual button that sent the event. First, we set up a string variable to store the action command of the button.

Next, we compare the command to a number of pre-defined strings in the controller. If there is a match, we respond appropriately. For example, if the command tells us that the even is coming from the backwards button, we call `oneStep( )` with a false argument. This is the same as what would happen if the user chose the **Back** item from the **SlideShow** menu. However, we are responding to a message from the controller instead of a message from a menu item.

Back to top

Summary

We did a lot of work in this class. We started out setting up the application menus and initializing class data members. Then we initialized the application state in the main routine and created the controls. This completes all of the methods in the main application class, `SlideShow`. We registered our listeners and implemented the threading model responsible for advancing images in the slide show. Next, we implemented the methods called in response to the various menu items and buttons, and implemented some custom `MRJ` handlers for responding to AppleEvents. Finally, we implemented our main application event handling routines.

In our last file, we implement the ImageNameFilter used by the open file dialog in this class. To return to the main tutorial file, click here.

Building the Image Name Filter

File: ImageNameFilter.java

Contents

Overview

The `ImageNameFilter` class is designed to "filter" files presented in a dialog. A filter takes a group of files and determines if they meet a certain criteria. If a file adheres to the specifications, it is displayed in the dialog. Files that don't meet these criteria are "filtered" and not displayed in the file list of the dialog. Our `ImageNameFileFilter` filters files based on whether or not they appear to be image files (which is based on the extension of the filename).

Our class implements the `FilenameFilter` interface, which specifies that derived classes implement a single `accept( )` method.

Steps to Follow

Step 1 - Implementing the accept( ) routine

When a filename filter is used by a file dialog, the `accept( )` method of the filter is called once per file in the file list. It is the job of the method to either accept or reject the file by returning a Boolean. The file returns `true` if it meets the criteria of the filter and should be displayed or false otherwise.

```
import java.io.File;
import java.io.FilenameFilter;

public class ImageNameFilter implements FilenameFilter
{
   /**
    * Tests if a specified file should be included in a file
    * list.
    *
    * @param   dir    the directory in which the file was found.
    * @param   name   the name of the file.
    * @return  true if the name should be included in the file
    *          list; false otherwise.
    * @since   JDK1.0
    */
    public Boolean accept(File dir, String name)
    {
        //Need to filter for image files (files whose names
        //end with ".jpg", ".gif", or ".jpeg", regardless
        //of case).
         //Insert "ImageNameFilter accept"
```

Note that we import both **java.io.File** and **java.io.FileNameFilter**. As we previously stated, our class implements the FilenameFilter interface, and we implement the single method `accept( )`. This method returns a Boolean and takes both a file reference and string (the filename). These references are passed to us automatically by the `FileDialog` we are registered with.

Locate the ImageNameFilter accept clipping in the ImageNameFilter folder and drag it directly below the last line of code shown above. Your code should now look like this:

```
import java.io.File;
import java.io.FilenameFilter;

public class ImageNameFilter implements FilenameFilter
{
   /**
    * Tests if a specified file should be included in a file
    * list.
    *
    * @param   dir    the directory in which the file was found.
    * @param   name   the name of the file.
    * @return  true if the name should be included in the file
    *          list; false otherwise.
    * @since   JDK1.0
    */
    public Boolean accept(File dir, String name)
    {
```

```
        //Need to filter for image files (files whose names
        //end with ".jpg", ".gif", or ".jpeg", regardless
        //of case).
         //Insert "ImageNameFilter accept
    String upperCaseName = name.toUpperCase( );
        return (upperCaseName.endsWith(".JPG") ||
                upperCaseName.endsWith(".JPEG") ||
                upperCaseName.endsWith(".gif"));
    }
}
```

Our code for determining whether the file is an image or not is very simplistic. We take the file name, and we convert it to uppercase characters, storing the result in a temporary variable. Then we return the result of the Boolean expression that returns `true` if the name string ends with ".jpg", ".jpeg", or ".gif". This is a simple way of checking, but it won't work if the extension of the file is incorrect or absent.

Summary

`FileName Filters` are very simple classes. They implement an interface which contains a single method named `accept( )`. The method takes a ile reference and a string reference (the name of the file), and the filter can check these objects against some criteria and accept or reject the file based on the return value of the method.

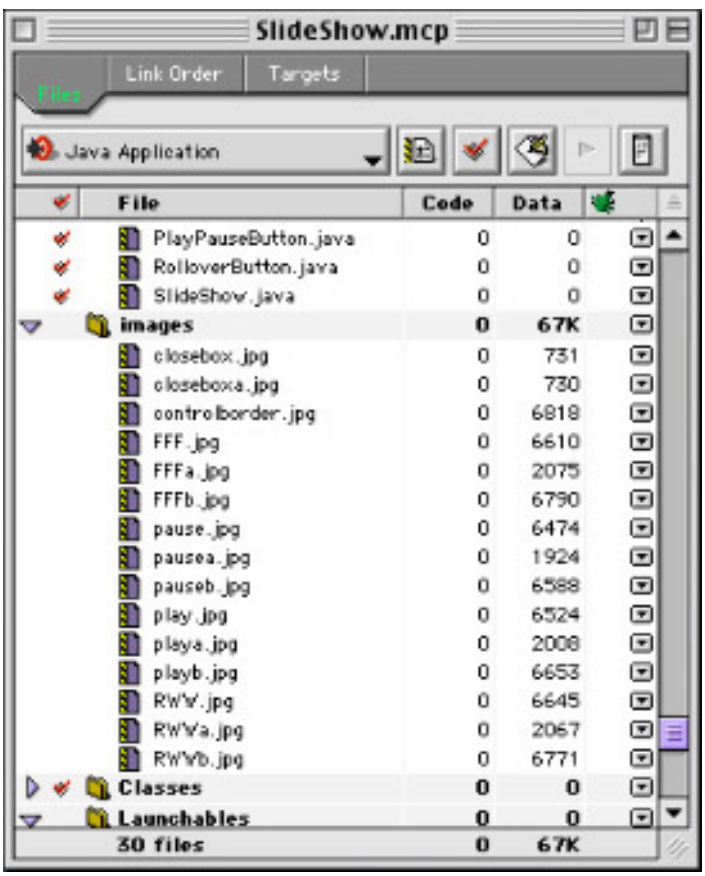
Our filter simply checks the end of the file name to see if it matches a specific extension.

This concludes the source file modification for our tutorial. Click here to return to the main tutorial file where we will put together the finishing pieces needed to build our application.

An Introduction to Java Programming

Previous Section | Table of Contents | Next Section

Step 11 - Adding the Image Resources



Now, that we have completed all of the code for our tutorial, it is time to configure our project to use the image resources needed for our application. As you can see from the picture (right), our project has all of the images used by our image buttons directly added to the project in an images group. This was accomplished by dragging the images folder from the Finder into the project window.

When you have an application that uses image resources and you want them bundled in to your application, it is a good idea to add them to your project. Once these resources are part of your project, you can select all of the images, and then click on the project inspector

button in the project window so that you can tell CodeWarrior to copy the image files directly into the output file for the application.

The project inspector dialog is shown in the image below. For image files, you will need to click the checkbox **Merge Into Output**. This tells CodeWarrior that you want to copy the image file into the jar that is output when you build the application.

This is a much more desirable approach then having a folder of images used by your application that the user could muck with an image editing tool, or move out from under

your application.

The project associated with this tutorial should already be set up so that all of the image files will be merged into the output file. This is mentioned primarily for future reference.

If you do not perform this step, the SlideShow application will not be able to find the image files when it looks for them.

Now that we have our image resources configured, close the inspector windows.

Back to top

Previous Section Table of Contents Next Section

An Introduction to Java Programming

Previous Section | Table of Contents | Next Section

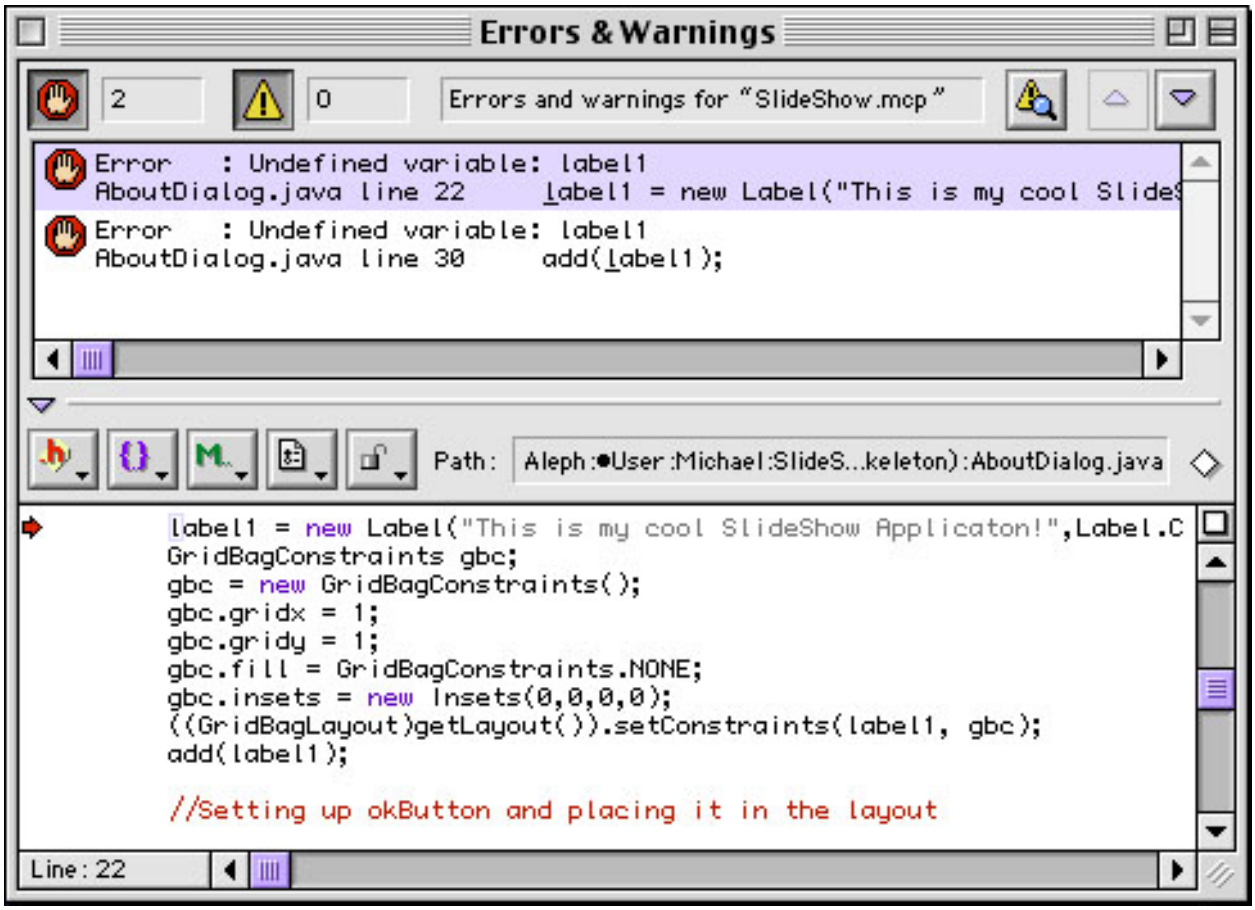
Step 12 - Building the Application

Now that all of the project sources are completed and the image resources and project settings are configured properly, we are ready to build the application.

To compile all of the sources, choose **Make** from the **Project** menu or use the Command key equivalent— <Command-M>.

You will see a build progress dialog similar to the one shown above. This dialog shows the status of the build operation including the name of the file being compiled, and the number errors or warnings encountered.

If you have any build errors, you will see a dialog that looks like the picture below. This dialog will list all of the errors found by the compiler. At the top of the window is a stop icon that shows the number of errors. The alert icon shows the number of warnings.



The pane at the top shows all of the errors and warnings in the file. If you select an error from this list by clicking on the message, the bottom panel will show the line of code where the error appears.

The picture illustrates two errors caused by a missing clipping. In this example, our code references the variable `label1`, but the code where that data member is declared is missing.

If you encounter compile errors, compare your modified skeleton source file with the completed source file in the **sources (completed)** folder. You must eliminate any compile errors before going to the next step.

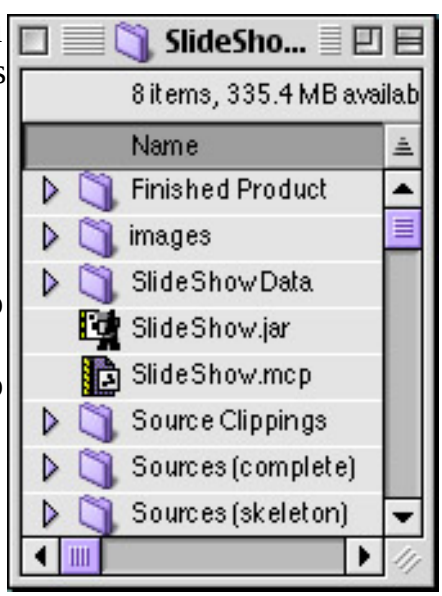
Back to top

Previous Section Table of Contents Next Section

An Introduction to Java Programming

Previous Section | Table of Contents | Next Section

Making a Double-Clickable Application



Now that you have successfully built your application, you should see a new file, “SlideShow.jar,” at the same level as your project file (see picture right). This file is the output file created when you perform a make operation on your project. This file contains all of the compiled class files and image resources needed by the application.

This file is a library file; it is not an application. In order to make our application runnable in a convenient format, we need to build a double-clickable Macintosh application. To do so, we need to use a tool called JBindery which is part of the MRJ SDK.

JBindery is located in the MRJ SDK folder in a folder called JBindery. The application icon is shown above. Launch JBindery by dropping the SlideShow.jar icon on the JBindery icon. (If JBindery doesn’t accept the drop, that means that your desktop database needs to be rebuilt. Double-click on the application icon, and follow the steps carefully to make sure that the setting match). If you need additional information,

consult the JBindery Adobe Acrobat documentation file, *Using JBindery*, located in the same folder as the application.

When JBindery launches, you will see a dialog box with the Command page showing.


Command

If the Command page is not visible, click on the Command icon. The first text field labeled “Class name” needs to match our application class that contains the main entry point of the application, the `main( )` routine. This field needs to read “SlideShow” since that is our main class.

The additional fields allow us to specify arguments to main (the **Optional parameters** textfield), and redirect either `stdin` or `stdout` (the console).

Since we don’t use these features, we can use the default values for these items.

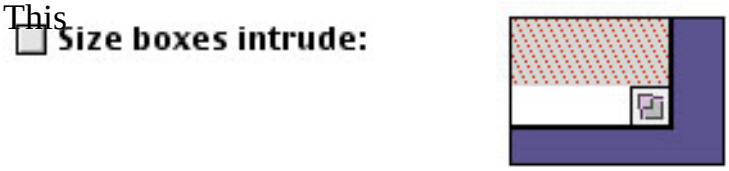

Classpath

Now click on the Classpath icon in the left column to go to the classpath page. The classpath page contains a panel for adding class files, .jar files, and .zip files that contain resources needed to run the application. The first item `$CLASSPATH` is the implicit system path where Java classes are located. Any local classes should be placed after this entry.

As the above shows, our SlideShow.jar needs to be added to the classpath so that our application classes can be found by the class loader. If SlideShow.jar does not appear in this list, you may add it by dragging it from the Finder into the window below the $CLASSPATH line. Since we only have one .jar file for all of our classes, that is the only entry that we need to add. If we had multiple class files or .jar archives, we would want to add all of them here.

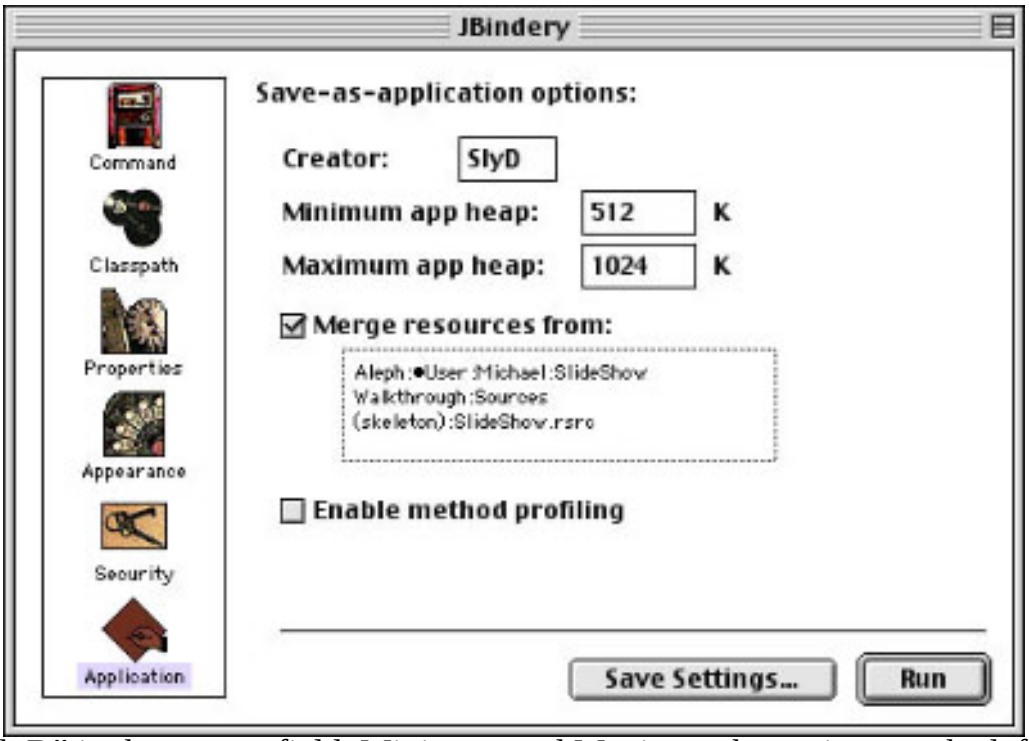
Now click on the Appearance icon in the left column to go to the appearance page in JBindery.

The only item that needs to be changed is the checkbox that reads **Size boxes intrude**. This



needs to be checked in order to make our application more Mac-like.

Lastly, click on the Application icon in the leftmost pane. Here, we will edit the settings that will be used by the double-clickable application. We set the 4-character creator code of the application, minimum and maximum application heap sizes, and specify the resource file to be used.



Type “SlyD” in the creator field. Minimum and Maximum heap sizes can be left at their default values.

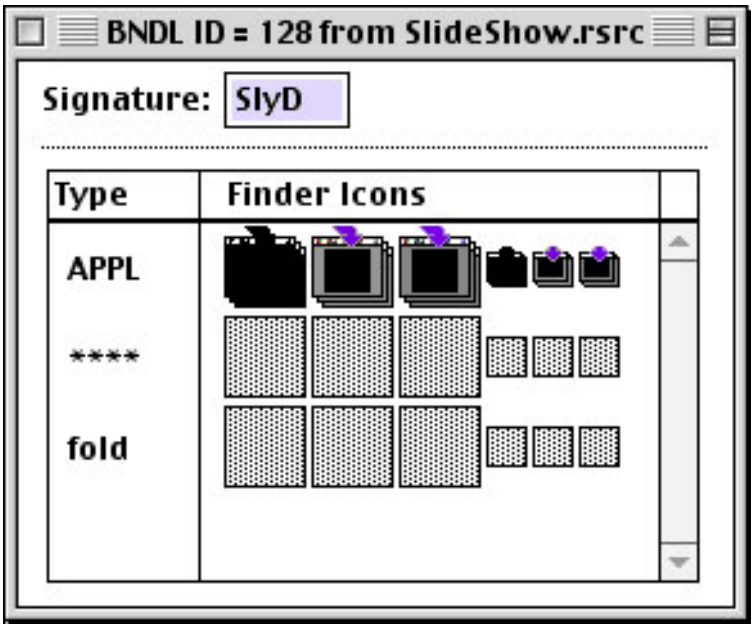
Drag the file “SlideShow.rsrc” from the **Sources (skeleton)** folder in the Finder to the rectangular area below the **Merge resources from** checkbox. When this is successfully completed, you should see the full path of the resource file in the non-editable text field.

This resource file has been pre-created for your convenience to include common Macintosh resource types such as a version resource. We have also added two resource types that you may wish to consider using in your own applications.

The first is a `'MENU'` resource. We add a `'MENU'` resource with an ID of `1128` with an Apple icon as the menu, and **About SlideShow…** as the sole menu item. By using this special ID, and registering our about handler, `MRJ` knows to insert this resource as the first menu item in the Apple Menu. Our menu handler is called when this item is chosen. This also removes the default **Quit** menu item from the Apple Menu, which is not preferred for finished Macintosh applications.

It is important that the menu is created in exactly this manner, or it will not work correctly. Additional information is provided in the Adobe Acrobat Document *About MRJ Toolkit* that ships as part of the MRJ SDK.

The other main resource we added to the resource file was a `'BNDL'` resource. This resource tells the Finder what icons to use for the application and associated files. It also tells the Finder what type of files your application can accept for drag and drop.



The image (left) shows the `'BNDL'` resource in ResEdit. We specify the signature (which is needs to be the same as the creator we specified in the **Application** pane in JBindery). Our bundle supports three different types. The APPL type is our application, and we specify the 1-bit, 4-bit, and 8-bit small and large icons to be used. The `'****'` entry specifies that we can handle any file type (we don't create this type, so we didn't specify a custom icon), and the `'fold'` item specifies that we can accept folders. As you recall, our `OpenDocument` handler knows how to deal with folders and files.

Now that we have looked at the resources and specified the resource file to be used in JBindery, we can build the application. Click on the **Save Settings…** button and save the application in the same location as your project file using the name "SlideShow".

A double-clickable application will be created and should have the icons we specified. That's it! Congratulations! You have made your first real Java-based application. Feel free to launch your application and drop some image files on it to see how it works.

Back to top

Previous Section Table of Contents Next Section

An Introduction to Java Programming

Previous Section Table of Contents Next Section

Summary

We've covered a lot of ground here. First, we talked a little bit about our application and configured our project correctly. We are building an application, so we needed to specify "Application" as the project type and set the output format to use the ".jar" format. Once our project was set up, we jumped right into the implementation of our classes.

We started out with the `AboutBox`, and then moved on to our button classes, which are a classic example of good object-oriented programming methodology. We built broad low-level functionality into the base class and then derived a series of subsequent classes which successively refined the underlying classes.

Next, we implemented the `Controller` class which demonstrated event handling and creating and managing floating windows. Finally, we implemented the `Application` class where we learned about threads, menu management, and special MRJ handlers.

Once we built our .jar file, we used `JBindery` to create a double-clickable Macintosh application.

Back to top

Previous Section Table of Contents Next Section

An Introduction to Java Programming

Previous Section | Table of Contents

Where to Go From Here

Now that you have made your first application, there are many interesting places where you can go to find out more about Java Programming, and Java on the Macintosh. The following is a list of recommended links you can follow for additional information.

The best place to start is on Apple's Java Developer page:

<http://developer.apple.com/java/>

There you will find lots of useful information including introductory material at:

<http://developer.apple.com/java/javaintro/>

Sun has a wealth of Java Information on their Java pages:

<http://www.javasoft.com/>

There you can find the Java Tutorial:

<http://java.sun.com/docs/books/tutorial/>

as well as API documentation

<http://java.sun.com/products/products.a-z.html>

I hope your journey was interesting and informative. If you encounter difficulty, errors, or have suggestions on how to improve this tutorial, please enter your feedback in the BugReporter tool.

Back to top

Previous Section Table of Contents
