New to Java™ Programming Center

Building an Application: Part 2: Introduction to Inheritance, Panels, and Layouts

by Dana Nourie, *December 2001*

Contents
Inheritance | NEXT>>Containers and Components

In Building an Application, Part 1, you learned about application objects, and that the plans for objects are written into files called classes. In addition, you learned to use predefined classes from the Java™ API, and to manipulate objects by calling methods, either your own or predefined.

So far, you constructed the first class of the Dive Log application and the place holder classes that DiveLog.java initializes.

Part 2 reinforces these concepts and introduces the `Welcome` class, which covers:

- Inheritance
- Images and text objects
- String concatenation
- Layout managers

Getting Started

In Part 1, you created the `DiveLog` class, which contained a constructor that builds the frame for the Dive Log application, a `JMenu`, and initializes a `JTabbedPane` object with six titled tabs. Each tab creates an object from a placeholder class that, for now, does nothing.

For this part of the tutorial, you need images and the `Welcome.java` placeholder class. You can use different images than those provided here, but to prevent problems with layout make them the same size as the images provided. As you work through the tutorials, you'll discover more ways to customize your application and its layout.

Follow these steps...

1. Save the following images to the `divelog/images` directory:
   - `diveflag.gif`
     Image size: 32 by 32 pixels.
   - `diver.jpg`
     Image size: 393 by 187 pixels.
2. Or create images of your own, but use the same pixel sizes as those listed above.

Note: It's assumed you installed the Java[TM] 2 Platform, Standard Edition on your system, and that you have completed Part 1 of the Building an Application tutorial.

Inheritance

Applications can consist of a single class, but most are built with many classes. The classes that make an application often communicate through a reference to an object and methods, using the dot operator. You've seen examples of this in the `DiveLog` class:

```
dlframe.setSize(765, 690);
```

In this example, `dlframe` is the reference to the instance of a `JFrame` object you created, but the `JFrame` class doesn't define a method called `setSize`. So, where does this method come from? How can you call a method on a `JFrame` object, when the `JFrame` class doesn't define that method? This process works similar to the way your hair color is passed down to you through your mother or father--through inheritance. But Java class inheritance gives a developer much more control over the child object than human inheritance does.

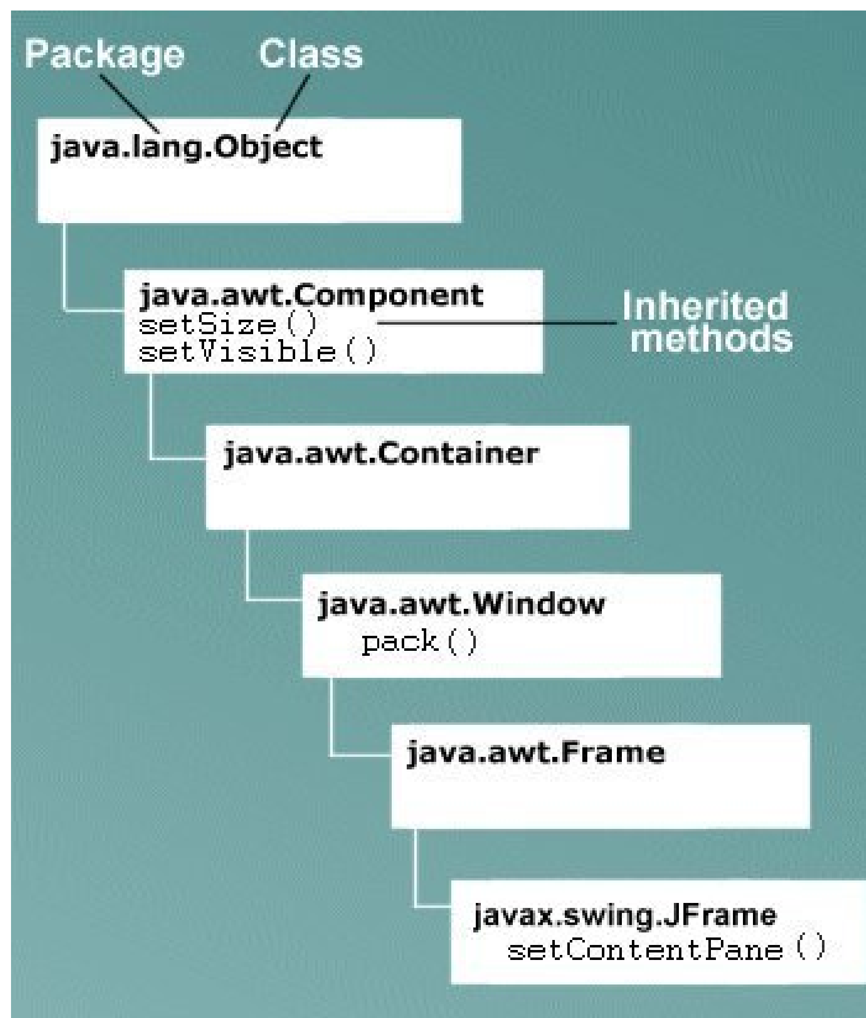
More on Inheritance

What is Inheritance?

Managing Inheritance

Examples of Inheritance

Because you instantiated an object of type `JFrame`, the `DiveLog` class inherited all the methods that `JFrame` contains. In addition, the `DiveLog` class inherited the methods that `JFrame` inherited. The `JFrame` class inherits methods and fields from several classes up the hierarchy tree:



JFrame Class Hierarchy

All classes inherit from class `Object` automatically. In addition, when you create an object of the `JFrame` type, this new object also inherits from the `Frame`, `Window`, `Container`, and `Component` classes. To call a method from one of these inherited classes, you generally use the dot operator with your reference variable. The `setSize` method was inherited from the `Component` class.

Not all inherited methods and fields are accessible, yet they are part of the make-up for that object. Later, you'll learn more about accessing certain types of data from parent classes.

There is a more direct way to inherit from specific classes: use the `extends` keyword in the class declaration. By using `extends`, your *child* (also called subclass or derived class) inherits from the *parent* or *super* class(es) and frees you from having to:

- Instantiate that desired inherited class to get to its methods and fields.
- Call a method from the super class with a variable reference and dot operator.

In other words, the `extends` keyword allows your class to inherit from a class of your choosing, unlike human inheritance in which you have no choice of who your parents are or what traits you inherit from them.

To make the `DiveLog` child class a child of the `JFrame` parent class, you type:

```
public class DiveLog extends JFrame
```

Now you have specified a class you want your subclass to inherit from, and it becomes that type of class. Using the above statement, you make the `DiveLog` class a type of `JFrame` object, just like a Labrador puppy is a

Labrador type of dog.

By extending to the the `JFrame` class, there is no need to instantiate `JFrame` in your class to get to its methods, as shown in the previous lesson:

```
dlframe.addWindowListener(new WindowAdapter()
dlframe.getContentPane().add(tabbedPane);
dlframe.setJMenuBar(mb);
```

Overriding Methods

A child class can change the behavior of any inherited method by *overriding* that method.
See Overriding Methods

Instead, you can write the method call without the variable `dlframe`. You call the inherited methods by their names:

```
getContentPane().add(tabbedPane);
addWindowListener(new WindowAdapter()
getContentPane().add(tabbedPane);
setJMenuBar(mb);
```

The parent class, `JFrame`, has a constructor you can call by using the `super` keyword. `DiveLog` can call the parent `JFrame` constructor and supply the `String` to appear at the top of the frame window as follows:

```
super("A Java(TM) Technology Dive Log");
```

In human inheritance, you get some genes from your mother and some from your father. With class inheritance, by using the `extends` keyword, the `DiveLog` object is a `JFrame` object with additional features that you added. In other words, the `DiveLog` object has everything a `JFrame` object has and more. In addition to having a frame with a title, the `DiveLog` object, derived these from the `JFrame` object, has:

- a `TabbedPane` object
- tabs with titles
- memory reserved from place holder classes where you create more objects.

To improve this class, you can move the `setSize(765, 690)`, `setBackground(Color.white)`, and `setVisible(true)` methods into the `main` method. Since `DiveLog` is a type of `frame` object, it makes sense to set size and background on the newly instantiated `DiveLog` object once it's built, rather than when it's being constructed. But either way works.

You don't need to rewrite your `DiveLog.java` to continue with this tutorial, but it is a good exercise in inheritance.

1. Open `DiveLog.java` in your editor.
2. Change the class from:

   ```
   public class DiveLog
   ```

   to:

   ```
   public class DiveLog extends JFrame
   ```
3. Delete the variable declaration: `private JFrame dlframe;`
4. Change the old `JFrame` constructor:

   ```
   JFrame dlframe =
      new JFrame("A Java(TM) Technology Dive Log");
   ```

   to:

   ```
   super("A Java(TM) Technology Dive Log");
   ```
5. Remove the `dlframe.` from the methods.
6. After initializing a `DiveLog` object in `main`, use the `dl` variable to call the `setSize(765, 690)`, `setBackground(Color.white)`, and `setVisible(true)` methods.

   For example:

   ```
   dl.setSize(765, 690);
   ```
7. Save the file.
8. Compile the [DiveLog.java]() class with:

   On a Windows platform:

   ```
   C:\divelog>javac -classpath C:\ DiveLog.java
   ```

   In the Solaris™ operating environment:

   ```
   divelog% javac -classpath /home/usr/ DiveLog.java
   ```

Note: You must be in the `divelog` directory while running this command. Make sure there is a space between the last \ or / before `DiveLog.java`.

The next Dive Log class `extends` a class to gain the benefits of inheritance. The `Welcome` class holds the content for the first tab in the Dive Log application.

Which class should the `Welcome` class extend?

A. `JFrame`
B. `JPanel`
C. Neither

Which class should the `Welcome` class extend?

The `Welcome` class extends the `JPanel` class, making it a `JPanel` object without having to instantiate the `JPanel` class with the `new` keyword. In addition, the `JPanel` is an important type of container.

Containers and Components

Containers and components are the ingredients of the GUI construction kit on the Java™ platform. The `java.awt` and `javax.swing` packages provide predefined containers and components to use in your applications, and you can develop or customize your own.

The Dive Log application uses predefined containers and components, customized to meet the needs of a simple dive log. In the `DiveLog` class, you created a frame container, either by extending to the `JFrame` class, or by creating a `JFrame` object with the keyword `new`. In addition, you created a second container: `JTabbedPane`, on which you initialize containers and components.

Containers and Components

Swing Components and the Containment Hierarchy

Widgets, Widgets, Widgets

Using Top-Level Containers

Using Intermediate Swing Containers

Before continuing with the `Welcome` class, where you add another container and components, you should understand the purpose of containers and components.

Containers

Real world objects need containers. Your car has a frame that holds other types of containers such as doors, a trunk, a front-end that encloses the engine, and a compartment that contains passenger objects like seats and controls to drive the car. In addition, your car has numerous other containers with smaller parts, or components, within. Software application containers also hold containers that hold numerous objects and often other containers.

A GUI application has at least these objects:

- A frame container to hold the visual and non-visual parts of the GUI.
- A panel container that is sectioned into a specific layout.
- Components that are added to the panel within the specific layout sections.

Follow these steps...

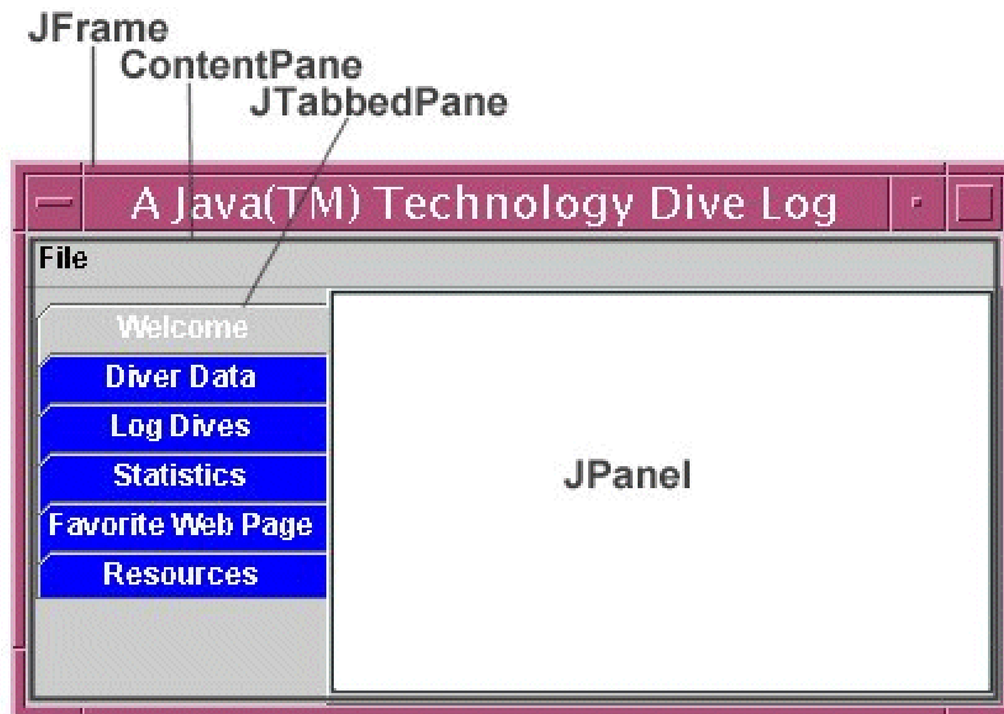
1. Open the Welcome.java file in your text editor.
2. Extend the `Welcome` class to `JPanel` in your class declaration if you have not done so already in Part 1:

```
public class Welcome extends JPanel
{ //Opens class

}//Closes class Welcome
```

3. Save the file.

Since the `Welcome` class extends the `JPanel` class, you are making this object a panel, in particular a `JPanel`, which is a container. Now you've created an application with:

- A **frame** container that contains a
- **content pane** container that contains a
- **tabbed pane** container that contains a
- **panel** container to hold other containers or components



Containers created so far

Components

Software components enable users to interact with an application, both passively and actively:

- Passively
  Components display text, titles, and images.

- Actively
  Components enable users to request or give information through buttons, menus, radio buttons, text fields (holds short strings), and text areas (holds long strings).

You see many components on the browser you are using now, such as:

- Buttons
- Scroll bars
- Text fields
- Text and images

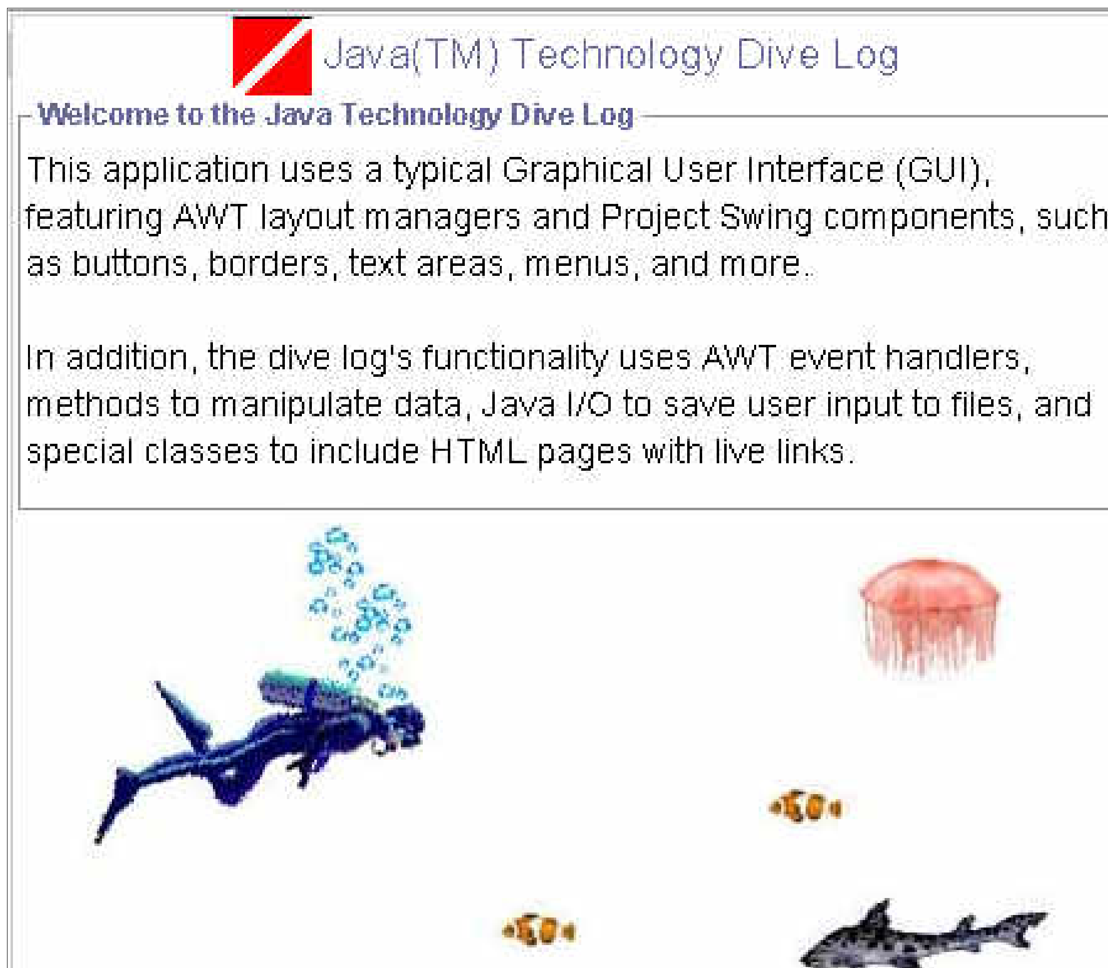
The Project Swing library has many predefined components available for use in your applications. When you create components, these objects can inherit methods from the `JComponent` class, such as:

- `setBackground(Color bg)`
- `setForeground(Color fg)`
- `setOpaque(boolean isOpaque)`
- `setToolTipText(String text)`
- `setVisible(boolean aFlag)`

Some of these methods should look familiar, as they were used in the `DiveLog` class. The important point about these methods is that you can call them on any component you create. You'll see those methods at work soon.

In the next programming exercise, you will create these passive components, shown below:

- Two images
- Text
- A text area
- Border with a title



Components in the Welcome class

And these objects:

- Label with an image and text
- Text area
- Second label with an image

Follow these steps...

1. Open the [Welcome.java]() file in your text editor.
2. Add the following variables just after the opening curly brace for the class. This variables do not yet have assignments:
   - `private JLabel jl;`
   - `private JTextArea ta;`
   - `private JLabel diver;`
3. Save the file.

These variables are declared `private`. There isn't any reason for outside classes to access them directly. In addition, these variables aren't given their assignments yet, because declaring them outside a method makes them available to the rest of the class. Initializing the variables within the class constructor forces these objects to be

built when the class itself is initialized.

Setting up the rest of the `Welcome` panel takes four steps. Here, step two is missing:

1. Declare variables.
2.
3. Create GUI objects, such as labels, images, and so forth.
4. Add the objects to the panel.

Which is step 2?
A. Set a layout manager in the `Welcome` constructor. B. Create a `main` method for this class. C. Neither.

Which is step 2?

Setting a layout manager in the constructor is the second step in the creating the `Welcome` panel. The `DiveLog` class is the entry point for the Dive Log application and contains the only `main` method that this application needs. From there, all other classes you created are initialized, so no other `main` methods are needed.

However, a layout manager for each class is necessary.

Layout Managers

Before building and adding components to a container, it's good to have some idea of how you want those components arranged. Real world objects, like the furniture in your home, are arranged and set in specific locations, according to which objects need to be close to one another and which objects can be spaced farther apart. For instance, it might not make sense to put your couch beside your television, but it would make sense to set the couch in front of the television with some space between the two objects.

More on Layout Managers

Exploring the AWT Layout Managers

Effective Layout Management

Lesson: Laying Out Components Within a Container

Application objects also need to be arranged. Layout managers are special objects that position the objects you create and add to containers. Each container, such as `JFrame` and `Jpanel` have a layout manager associated with them.

The Swing tool set primarily consists of the layout classes provided by the Abstract Window Toolkit (AWT), which has five layout manager classes:

- `FlowLayout`
- `GridLayout`
- `BorderLayout`
- `CardLayout`
- `GridBagLayout`

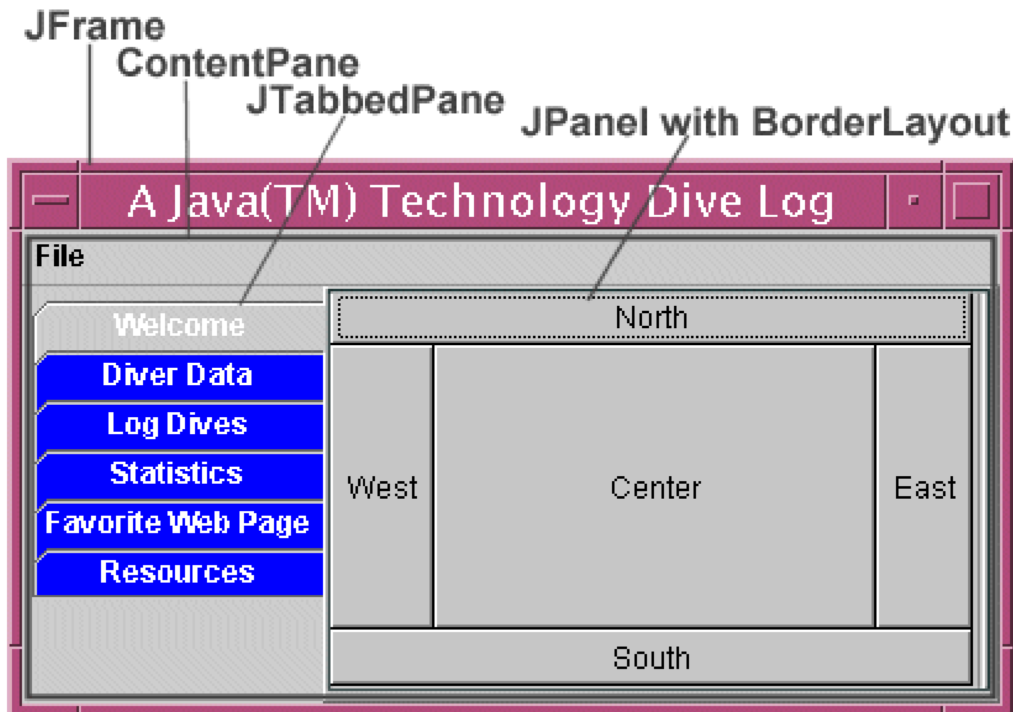
In addition, Swing has several layout managers, such as the box, scroll pane, overlay, and viewport layouts. Deciding which manager to use depends on how you want to arrange the components within a container.

The simplest layout manager is flow layout because it arranges components from left to right. There are limited uses for this layout. If you have only one or two components in your application, then flow layout might be sufficient.

Border Layout Manager

The border layout manager is an easy-to-use layout manager. It is popular because you can do so much or so little with it.

Five components may be added to the border layout manager and arranged on any of the four borders and in the center. This pattern is similar to geographical locations indicated by North, South, East, West, and Center in the image below:



Dive Log containers with Border layout showing all five regions

If the application is resized, the components are resized accordingly, and the arrangement is preserved and maintained by the layout manager. Each region is stretched to fill the container. Notice how North and South regions extend to the left and right edges. Also note that the West, Center, and East areas do not extend to the upper and lower edges.

Only five components may be added to the border layout, one in each region. You can overcome this limitation by adding another container, such as a `JPanel`, with its own layout within that container. For instance, you might add a `JPanel` to the South panel, and this new `JPanel` might also use the border layout. Then you can add five more components to the South panel. You can do this endlessly and make complex layouts with many different layout managers. You'll see an example of this technique later.

Contrary to placing layouts within layouts, you don't have to make use of all five regions of the border layout manager.

Border layout using only three regions

Like the illustration above, the `Welcome` class uses the border layout with only three regions occupied by objects.

To start, you need to set up the class constructor, and call the method to use the border layout manager for this `Welcome` container.

Follow these steps...

1. Open the `Welcome.java` file in your text editor.
2. Create a constructor for this class:

```
public Welcome()

    {  // Opens constructor



    } // Closes constructor
```

3. Immediately following the opening curly brace of the constructor, call the `setlayout` method:

```
setLayout(new BorderLayout());
```

4. In addition, set the background color of this panel:

```
setBackground(Color.white);
```

5. Save the file.

Your code should look match this Welcome.java file.

The `setLayout` method comes from the `Container` class and is inherited by making this class a `JPanel`. The method sets a layout for the container by initializing a layout object that you specify as a method argument. In this case, you initialized an object of the `BorderLayout` class, which makes this `Welcome` container managed by a border layout manager. Instructions for adding objects to this container in the arrangement of a border layout is covered later.

At this point, you are ready to create the components that are later added to the panel and arranged by the layout manager.

You declared three variables at the beginning of the `Welcome` class. Within the constructor, you create the objects to assign to those variables:

- A dive flag image with a title
- A text area with a titled border
- An image of a diver and fish

Which classes should you initialize for the image, text, and text area objects?

A. Create `JLabel`, `ImageIcon`, and `JTextArea` classes.
B. Declare `JText`, `JGIF`, and `JTextArea`.
C. Neither.

Which classes should you initialize for the image, text, and text area objects?

To display text and images, the `JLabel` and `ImageIcon` classes are used. A label is a short string of text with, or without, an associated icon image. Labels are generally used to display titles or to label other application objects, such as text fields or text areas. You use the `JLabel` and `IconImage` class to display the title and dive flag on the `Welcome` panel. The class `JTextArea` is used for text areas.

Labels and Images

A `JLabel` object is one of the most simple components in the Project Swing library. You use `JLabel` to display text and or images in much the same way as a real-world label. The label itself holds the text and image much like the label around a can of vegetables.

More on the ImageIcon and JLabel Classes

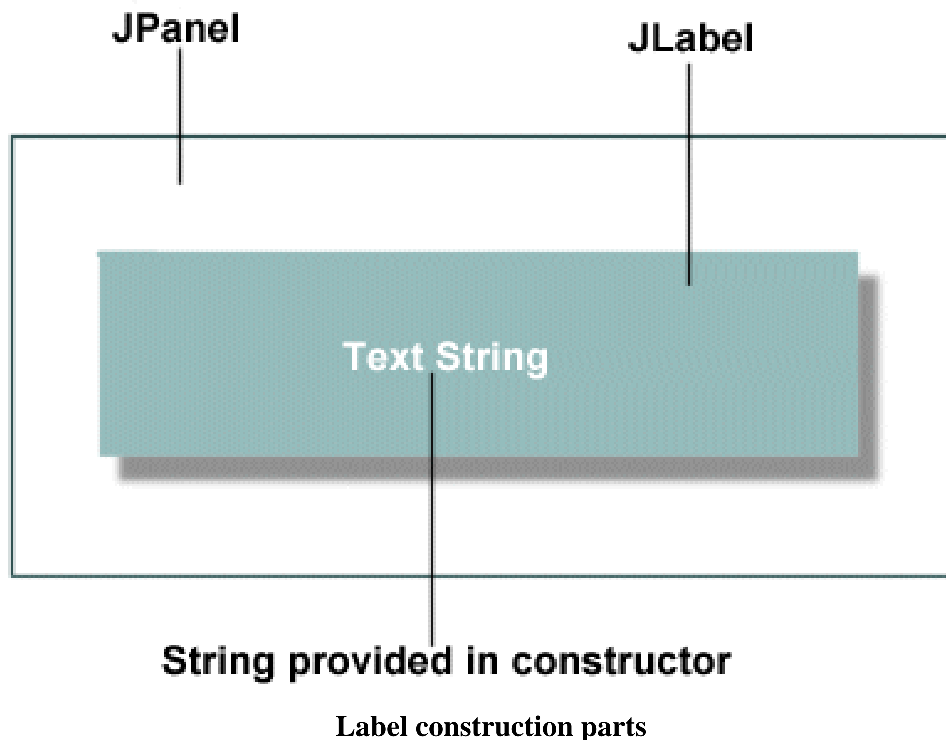
How to Use Labels

How to Use Icons

This is how you display a text label in an application:

1. Initialize an object of the `JLabel` class with the `new` keyword.
2. Assign it to a variable name that is short yet descriptive.
3. Supply the desired text in the `JLabel` constructor.

4. Add the label to the panel.



Label construction parts

The illustration above shows how the various parts of a label construction come together. The label is built with a `String` to display by adding it to the `JLabel` constructor, and the label is added to a `JPanel`. This label also has a blue background and a white foreground.

In the code below, a simple `JLabel` is created:

```
JLabel text = new JLabel("This is text on a JLabel.");
```

So far a label object with text is created by identifying the object as type `JLabel`, assigning it to the variable `text`, using the keyword `new`, and providing the text as a `String` object to appear on the label.

To actually display it in the application, however, the label must be added to the panel.

Recall that the `Welcome` class is a type of `JPanel` because it extends to the `JPanel` class. This subclass `Welcome` inherits methods from the `Container` class, including one called `add`. To add the label to the panel, call the method by name, `add`, and provide the name of the object you want to add to the panel, in this case `text`:

```
add(text);
```

Those two lines of code display the text `This is text on a JLabel.` You can change the way the text appears by using different `JLabel` constructors, or by referencing the variable name with a method, such as applying a specific font face, or having the label appear in a specified color.

As you saw, methods from the `Component` class are inherited and readily available by using your object variable name with the dot operator and calling the method:

```
text.setForeground(Color.red);
```

The `JLabel` class also has methods that allow you to manipulate your label object. The

`setHorizontalAlignment` method positions the label object:

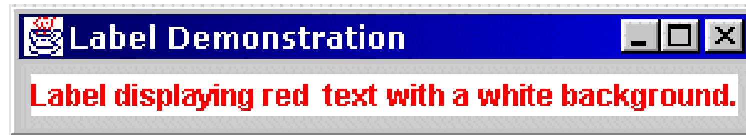
```
text.setHorizontalAlignment(SwingConstants.RIGHT);
```

Here, as before, name the object you want to manipulate, then call the method using the dot operator and the method name. Pass in the `SwingConstants` class, the dot operator, and the position you want the object to be placed.

In addition, you can create `JLabel` objects with constructors that build some of these details, such as position, into the label upon creation:

```
JLabel text = new JLabel(
   "This text will be centered.", JLabel.CENTER);
```

An application called Label.java demonstrates the concepts covered so far, and results in the following:



The running application with one label

Note: Because only one component is added to this application, the default flow layout manager is sufficient.

The `JLabel` class also offers a constructor that displays and image icon with the text, which is detailed in the next section.

Displaying Images with ImageIcon and JLabel

To display images, you create an object with the `ImageIcon` class and provide the constructor with an image file in a format supported by the `ImageIcon` class, such as `.gif` or `.jpg`. The image format can be passed to the the constructor as a:

- `String` with the file name
- `URL` object specifying the location of the image
- `Image` object
- Btye array of the image data.

The `ImageIcon` object can then be loaded into a `JLabel`, `JButton`, `JMenuItem`, or similar component.

All this is done in one step due to a handy `JLabel` constructor:

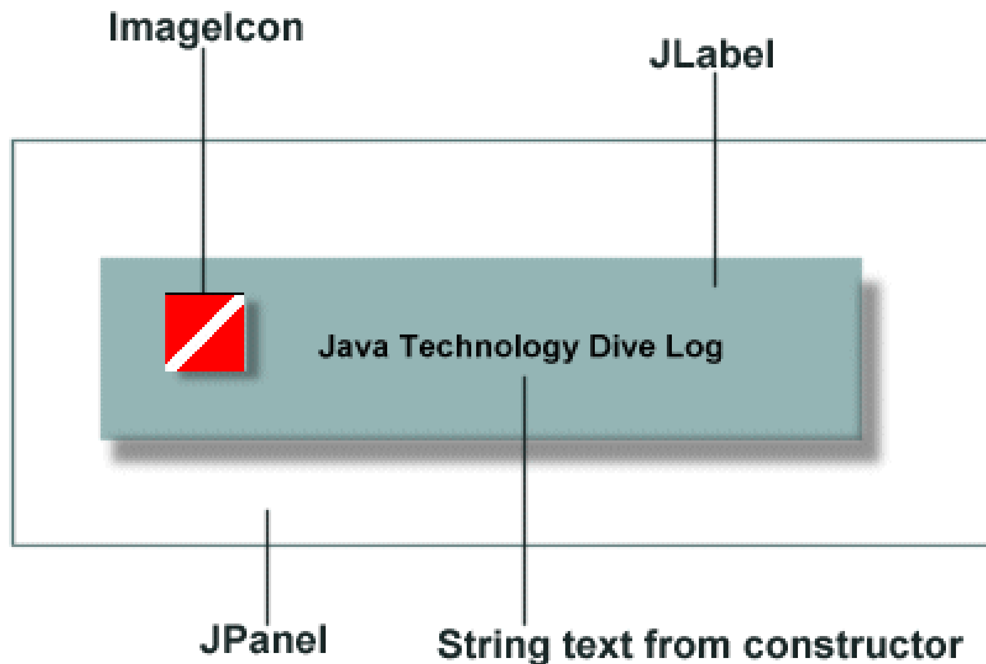
```
JLabel(String text, Icon icon,
          int horizontalAlignment)
```

Where it says `Icon icon`, you insert the `ImageIcon` constructor that you want to use. For instance:

```
// assigns the variable imageLabel
// as a JLabeltype
JLabel imageLabel =
// Initializes a JLabel object with
```

```
// a String object,
// Label with an Image.
      new JLabel("Java Technology Dive Log",
// Initializes an ImageIcon object,
// specifically a diveflag.gif in a
// directory called images.
           new ImageIcon("images/diveflag.gif"),
// Centers the JLabel object.
                                 JLabel.CENTER);
```

The code above creates a `JLabel` object that acts as a container for the `ImageIcon` object and text, as illustrated below:



JLabel with ImageIcon

Now that the label and image object are initialized and assigned to `jl`, you can call other methods on that object to change some of the default characteristics, such as the font face used in the text.

The `JComponent` class has a method `setFont` just for this purpose. The method requires a `Font` object as an argument, meaning you must initialize the `Font` class with the `new` keyword. The `Font` class initialization requires three arguments:

- A `String` representation of the font, such as `"Courier"`
- The style of the font, such as `Font.BOLD`
- An `int` for the point size of the font, such as `14`

To change the default font displaying on the `JLabel`, you use:

```
jl.setFont(new Font("Times-Roman", Font.BOLD, 17));
```

This code sets the font to `Times Roman`, in `bold`, at `17` points on the `jl` object.

1. Open the `Welcome.java` file in your text editor.
2. On the line after the call to the `setLayout` method, Initialize the `JLabel` and `ImageIcon` object:

```
jl = new JLabel("Java(TM) Technology Dive Log",
          new ImageIcon("images/diveflag.gif"),
                                  JLabel.CENTER);
jl.setFont(new Font("Times-Roman",
                               Font.BOLD, 17));
```

3. Save the file.

Now the text appears in the the font face, style and size that you set.

Text Areas with JTextArea

The `JLabel` is used to display small strings of text. To display large blocks of text, you initialize an object of the `JTextArea` class. You have already set up a variable for a text area object called `ta` at the top of the class. Next you create the text area object and assign it to the variable you created.

In the `Welcome` class, you use the following `JTextArea` constructor:

```
JTextArea(String text)
```

Your code will looking something like:

```
ta = new JTextArea("This application uses a
     typical Graphical User Interface (GUI),
     featuring AWT layout managers and Project
     Swing components . . .");
```

But there is a problem with the code above. If you compile that constructor as written, you get an error.

Why won't the code sample compile?
A. The `String` can't contain an ellipsis. B. The `String` contains hard returns. C. Neither.

Why won't the code sample compile?

You get a compile error because of the hard returns in the `String`. You can write a long, continuous `String` without breaks or hard returns, but that makes it difficult to work in your text editor. Instead, break the `String`

into smaller pieces, staying within the screen of your text editor, using `String` concatenation.

Formatting Text Areas

Because a large block of text is displayed in the text area, some formatting is needed. Long strings need to be broken into smaller parts that can be joined as one, and paragraphs require new lines between them. In addition, you may want the text to appear in a specific font or style. For this kind of formatting, the Java™ programming language uses:

- String concatenation
- Character sequences
- Predefined methods

More on Concatenation and Text Areas

Strings and the Java Compiler

Class String

Using Text Components

Fundamentals of Swing: Part 1

Class JTextArea

String Concatenation

Joining two or more `String` objects is called string concatenation. To join the strings to form one `String` object, you use the + operator. Strings are always enclosed in quote `"  "` marks. The first quote tells the compiler when it reaches the beginning of one `String` object, and the next quote tells it when it reaches the end of that `String`.

The + operator tells the compiler to store the following object with the first object and assign the concatenated object to the variable on the left side of the assignment = operator:

```
firstString = "Java Programming";
secondString =  " is a lot of fun";
combineString = firstString + secondString;
```

Results in `Java Programming is a lot of fun` assigned to `combineString`. You get the same result with the following:

```
singleString =
    "Java Programming" + " is a lot of fun.";
```

And again the same results with:

```
singleString = "Java Programming" +
                 " is a lot of fun.";
```

Note that each quote pair is on a single line. It doesn't matter if the concatenation operator is with the first line or the second line, as long as the quote pairs appear on a line together. If the quote pair is broken so the first quote is on one line and the second on the next line, you get a compile error something like:

```
unclosed string literal
      println("Java Programming is
```

Character Sequences

String concatenation solves the problem of long strings that you need to break up, but it doesn't create space between paragraphs. To create a new line in the text you need to use an *escape sequence*.

The backslash (\) is an escape character that indicates to the compiler that you want a special character. You escape characters you want to display, such as quotes or < >. In addition, the backslash, in combination with certain characters, results in the insertion of a tab, carriage return, or a new line.

To create paragraph space, use `\n` to insert a new line. Now you can populate the `JTextArea` object with the text you want to display.

```
longString = new JTextArea("This shows you how to display" +
   " long strings by concatenating them." +
   " Enclose each string in quotes and" +
   " concatenate each line with the + operator.");
```

Change this text to suit the needs of your application, or use the text provided.

Follow these steps...

1. Open the `Welcome.java` file in your text editor.
2. Add the following to create a text area with the text and assign it to the variable `ta`:

```
ta = new JTextArea("This application uses a" +
   " typical Graphical User Interface (GUI)," +
   " featuring AWT layout managers and Project" +
   " Swing components, such as buttons, borders," +
   " text areas, menus, and more." +
   " \n\nIn addition, the dive log's functionality" +
   " uses AWT event handlers, methods to manipulate" +
   " data, Java I/O to save user input to files," +
   " and special classes to include HTML pages with" +
   " live links.");
```

3. Save the file.

Notice each line has a string enclosed in a set of quotes, followed by the concatenation operator to join the strings. The escape sequence `\n` creates a new line, and `\n\n` creates two new lines to give extra spacing between paragraphs.

Methods

The `JTextArea` class has methods to handle several formatting issues:

- `setFont(Font f)`
  Sets the current font face, style, or size.
- `setLineWrap(boolean wrap)`
  If set to `true`, lines wrap if they are too long to fit within the allocated width. If set to `false`, the lines are not wrapped.
- `setWrapStyleWord(boolean word)`
  If set to `true` the lines wrap at word boundaries such as white space. If set to `false`, the lines wrap at characters.

Many more methods are available in the JTextArea class, but those shown above are used in the Welcome class. In addition, JTextArea inherits methods from the JTextComponent class. A method specifically used in the Welcome class:

- setEditable(boolean b)
  Makes the text component either editable or not.

The JTextComponent class also has many methods available for using the JTextArea as an input box for users. Those features are demonstrated in other classes of the Dive Log application. In addition, JTextArea has other constructors, which allow you to set a specific size, and is also demonstrated later in the tutorial.

In the next programming exercise, you set a specific font face, style, and size for the text area, set line wrap to true, set the wrap style to wrap at word boundaries and disallows users to edit by putting setEditable on false.

Follow these steps...

1. Open the Welcome.java file in your text editor.
2. Add the following methods immediately after the text area initialization:

```
ta.setFont(new Font("SansSerif", Font.PLAIN, 14));
ta.setLineWrap(true);
ta.setWrapStyleWord(true);
ta.setEditable(false);
```

3. Save the file.

The setFont method is called on the ta object by using the dot operator. You provide the method arguments by initializing the Font class with the new keyword.

The Font class constructor needs the following arguments:

- A font name as a String
- Style field from the font class with the dot operator
- The point size

The three methods setLineWrap, setWrapStyleWord, setEditable require a boolean keyword, meaning simply they are set to true or false. Setting setLineWrap and setWrapStyleWord to true forces the ta object to wrap the lines of text to the size of the text area window, and it makes the breaks at white spaces between words. The argument false for the setEditable method makes the text area read only and does not allow for any editing features.

Decorating with Borders

Border decorations are aesthetically pleasing and allow developers to add a topic to a section, or title an object. In addition they aid the user by defining specific areas of a screen, or sectioning a screen into logical areas.

You can create beveled, etched, empty, compound, matte, raised bevel, or titled borders with methods from the BorderFactory class. First, though, you call the setBorder method from the JComponent class.

1. Open the `Welcome.java` file in your text editor.
2. Call `setBorder` on the object you want to have a border.
3. Call the `BorderFactory` class and the `createTitledBorder` method with the dot operator:
   `ta.setBorder(BorderFactory.createTitledBorder`
4. Provide the `String` you want to appear as the title in the `createTitleBorder` method:
   `("This is a Title"));`

The method call should look like:

```
ta.setBorder(BorderFactory.createTitledBorder(
        " Welcome to the Java Technology Dive Log "));
```

- Save the file.

The `ta` variable name identifies which object needs a border. The `setBorder(Border border)` takes the `BorderFactory` class as an argument, calling its `createTitledBorder` method with the dot operator, and provides the `String` to display with the border.

Try out other borders, checking to see what kinds of arguments they require from the `BorderFactory` documentation.

To add the last image of the `Welcome` panel:

1. Open the `Welcome.java` file in your text editor.
2. Initialize a `JLabel` with the `new` keyword.
3. Include an empty `String` (No text appears with this image).
4. Initialize an instance of the `ImageIcon` class.
5. In the `ImageIcon` class initialization, include:
   - A `String` for the image name and path
   - A `JLabel` field to designate position
   - Add the following code to your file, just after the methods you called on `ta`:

     ```
     diver = new JLabel("",
             new ImageIcon("images/diver.jpg"),
                                   JLabel.CENTER);
     ```
6. Save the file.

Note: You can write the above all on one line. It has been split here only to prevent printing problems, but it compiles if you copy and paste because quote pairs have been not been split.

So far you

- Declared three variables.
- Initialized and assigned objects to each of those variables in the constructor.
- Sent called methods on those objects to manipulate them using the dot operator.

The last step in completing this panel is adding the objects to each to a region of the layout. This brings you back to the border layout manager.

Constants are easy to identify in code because they are written
A. In uppercase letters, such as `PI` B. Written in initial caps `LoanAmount` C. With the first word in lowercase, and the second word in uppercase, such as `redBackground`

Constants are easy to identify in code because they are written

Constants are written in all uppercase letters, such as `PI`, `NORTH`, `RATE`. A constant is a value that never changes. The `Math` class, for instance, has constants you can use, such as `PI`. The `BorderLayout` class calls its constants *constraints* to identify a specific location in the layout. You might use constants of your design to represent a fixed date, a rate that remains the same for long periods, or a specific day of the year.

Adding Objects to the Panel

Adding the objects you created to the panel is the last step in creating the `Welcome` panel.

Recall that you don't need to use all five regions of the border layout. Because there are only three objects to add, you only need to use three regions of the border layout:

Reviewing Concepts

Class JPanel

Class JTextArea

Class BorderLayout

Class BorderFactory

Using three border layout regions

Each region is identified by its location and its behavior. The locations are similar to geological mapping, and each are given values called constraints. The behaviors affect how much space appears around a component as a window is resized:

- `BorderLayout.NORTH` constraint
  Placement: Uppermost region, approximately
  Behavior: This area only gets as *tall* as the component you place within it, but it stretches horizontally to fill the window.
- `BorderLayout.SOUTH` constraint
  Placement: Bottom area of the screen, approximately
  Behavior: Like `NORTH`, this area only gets as *tall* as the component you place within it, but it stretches horizontally to fill the window.
- `BorderLayout.EAST` constraint
  Placement: Right-hand screen area, approximately
  Behavior: This area only gets as *wide* as the component you add, but it stretches vertically to fill the window.
- `BorderLayout.WEST` constraint
  Placement: Left-hand screen area, approximately
  Behavior: Like `EAST`, this area only gets as *wide* as the component you add, but it stretches vertically to fill the window.
- `BorderLayout.CENTER` constraint
  Placement: In the center if components surround it, otherwise from center to whichever side does not have an occupied region.
  Behavior: Like `EAST` and `WEST`, this area only gets as *wide* as the component you add, but it stretches vertically to fill the window.

The purpose and importance of the region behaviors becomes more obvious as you add components and experiment with the different regions you can add them to. For now, add the components, using only the `NORTH`, `CENTER`, and `SOUTH` constraints.

1. Open the `Welcome.java` file in your text editor.
2. On a line before the closing curly brace of your class constructor, add the following lines of code to add the objects to the panel, using the border layout manager regions:

   ```
   add(jl, BorderLayout.NORTH);
   add(ta, BorderLayout.CENTER);
   add(diver, BorderLayout.SOUTH);
   ```
3. Save the file.

Your `Welcome.java` file should match this `Welcome.java`

Compiling the application is the same as the `DiveLog` class. When you compile the `DiveLog` class, all class that it calls compile as well, including the `Welcome` class that you just added to.

Compiling Code and Running the Application

Assuming you have the Java™ 2 Platform, Standard Edition (J2SE™) installed, and you've created a directory called `divelog`, compile the `DiveLog.java` file.

1. Compile `DiveLog.java` as follows:

   On a Windows platform:

   ```
   C:\divelog>javac -classpath C:\ DiveLog.java
   ```

   In the Solaris™ operating environment:

   ```
   divelog% javac -classpath /home/usr/ DiveLog.java
   ```

   Note you are in the `divelog` directory while running this command, and be certain to insert a space between the last \ or / before `DiveLog.java`.
2. Run the Dive Log with the following:

   Windows:

   ```
   C:\divelog>java -classpath C:\ divelog.DiveLog
   ```

Solaris:

```
divelog% java -classpath /home/usr/ divelog.DiveLog
```

The application should look similar to the image below:

Click to enlarge

Test the way the different regions behave by adding the components to different regions. This is done by changing:

```
add(jl, BorderLayout.NORTH);
```

to:

```
add(jl, BorderLayout.WEST);
```

Also, try changing the pane layout to `FlowLayout`, then add the components with:

```
add(jl);
```

In the next part of the tutorial, you will work with the border layout manager again and learn more about the region behaviors and how different types of components resize within those regions. For now, experiment with adding additional components and moving them to different regions of the border layout manager.

Summary

Part 2 of the Dive Log tutorial reviewed Java programming concepts from Part 1 and introduced new ones.

Inheritance

- A class directly inherits from a specific class by using the `extends` keyword.
- Extending a specific class gives your child class features of the parent class and more.
- Extending a specific class allows you to override methods of the parent class

Containers and Components

- Containers and components are the basic ingredients of the Java GUI.
- A panel is a container that can hold components and other containers.

- A `JLabel` is a simple container that holds text and images.
- A `JTextArea` is a component that holds multi-line text and has predefined methods for some formatting.

Layout Managers

- A layout manager is an object that arranges components on a panel.
- There are many predefined layout managers for developers to use in applications.
- The border layout is a flexible layout manager that has five usable regions, arranged much like a geographical map.

The Dive Log application classes serve as introductory examples to Java programming. It is not a comprehensive guide to the Java programming language, but instead as an example of an application that teaches basic Java programming concepts. The concepts are repeated in subsequent Dive Log tutorial parts as each class that makes up the tabbed panes is defined.

The Dive Log tutorial series covers more about methods, objects, and constructors in addition to creating other Swing GUI components and functionality. Each Dive Log class introduces new ideas as well as repeats what has been presented in earlier parts of the tutorial. In addition, each class representing a tabbed pane gets progressively more complex in terms of features and programming concepts.

Look for Part 3: Event Handling, Text Fields, Check Boxes, Inner Classes.

About the Author

Dana Nourie is a JDC staff writer. She enjoys exploring the Java platform and creating interactive web applications using servlets and JavaServer Pages™ technologies, such as the JDC Quizzes, Learning Paths and Step-by-Step pages in the New to Java Programming Center. She is also a certified scuba diver and enjoys exploring the kelp forests and colorful tube anemone-covered floor of the Monterey Bay.

Reader Feedback

Tell us what you think of this tutorial.

Very worth reading   Worth reading   Not worth reading

If you have other comments or ideas for future articles, please type them here:

Have a question about Java programming? Use Java Online Support.

Subscribe to the New to Java Programming Center Supplement to learn the basics of the Java programming language and keep up-to-date on additions to the JDC's New-to-Java Programming Center.

```
package divelog;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;


public class DiveLog extends JFrame
{

  private JTabbedPane tabbedPane;




      public DiveLog()
      {

      //Create a frame object to add the application
      //GUI components to.

        super("A Java(TM) Technology Dive Log");

        // Closes from title bar
        //and from menu
        addWindowListener(new WindowAdapter()
        {
          public void windowClosing(WindowEvent e)
          {
             System.exit(0);
           }
         });


         // Tabbed pane with panels for Jcomponents

         tabbedPane = new JTabbedPane(SwingConstants.LEFT);
         tabbedPane.setBackground(Color.blue);
         tabbedPane.setForeground(Color.white);

         //A method that adds individual tabs to the
         //tabbedpane object.
         populateTabbedPane();

         //Calls the method that builds the menu
         buildMenu();

         getContentPane().add(tabbedPane);


         }

         private void populateTabbedPane()
           {
           // Create tabs with titles


           tabbedPane.addTab("Welcome",
                               null,
                               new Welcome(),
                               "Welcome to the Dive Log");
```

```
      tabbedPane.addTab("Diver Data",
                        null,
                        new Diver(),
                        "Click here to enter diver data");

      tabbedPane.addTab("Log Dives",
                        null,
                        new Dives(),
                        "Click here to enter dives");

      tabbedPane.addTab("Statistics",
                        null,
                        new Statistics(),
                        "Click here to calculate dive statistics");

      tabbedPane.addTab("Favorite Web Site",
                        null,
                        new WebSite(),
                        "Click here to see a web site");
      tabbedPane.addTab("Resources",
                        null,
                        new Resources(),
                        "Click here to see a list of resources");
                } //Ends populateTabbedPane method


    private void buildMenu()
    {
      JMenuBar mb = new JMenuBar();
      JMenu menu = new JMenu("File");
      JMenuItem item = new JMenuItem("Exit");

      //Closes the application from the Exit
      //menu item.
      item.addActionListener(new ActionListener()
      {
        public void actionPerformed(ActionEvent e)
        {
           System.exit(0);
         }

        });// Ends buildMenu method

        menu.add(item);
        mb.add(menu);
        setJMenuBar(mb);
        }



public static void main(String[] args)
{

 DiveLog dl = new DiveLog();
 dl.pack();
 dl.setSize(765, 690);
 dl.setBackground(Color.white);
 dl.setVisible(true);
```

```
    }


} //Ends class
```

```
package divelog;
/**
 * This class creates the content on the
 * Welcome tabbed pane in the Dive Log
 * application.
 * @version 1.0
*/

import javax.swing.*; //imported for buttons, labels, and images
import java.awt.*; //imported for layout manager


public class Welcome extends JPanel
{ //Opening class


  }//Closes class Welcome
```

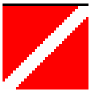
```
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.*;


public class Label extends JPanel
 { // Opens class Label

    // Variable to use for label
    JLabel text;
    // Class constructor where the label
    // is created.

    public Label()
     { // Opens constructor for Label objects

       // Uncomment the line below if you want
       // the panel itself to be white as well.
       // This line of code sends the method
       // message to the super class, which is
       // JPanel.
       // super.setBackground(Color.white);

       // Creates the label with a text String
       // and centers it.
        text = new JLabel(
                "Label displaying red text " +
                   "with a white background.",
                               JLabel.CENTER);
       // Sets the labels background
        // color to white.
        text.setBackground(Color.white);
        // Sets the foreground color, in this
        // case the text, to the color red.
        text.setForeground(Color.red);

        // Make certain the label object is
        // opaque. If you don't call this
        // method, the background color white
        // won't show, as it will be
        // transparent.

        text.setOpaque(true);
        // Adds the label to the panel.
        add(text);
   } // Closes constructor

    public static void main(String[] args)
    { // Opens main method

       // Create a window using JFrame
         JFrame frame =
           new JFrame("Label Demonstration");

      // Code to close down the app cleanly.
      frame.addWindowListener(new WindowAdapter()

         { // Opens addWindowListener method
          public void windowClosing(WindowEvent e)
```

```
        { // Opens windowClosing method
            System.exit(0);
        } // Closes windowClosing method
        }); // Closes addWindowListener
            // method.


      frame.setContentPane(new Label());
      frame.pack();
      frame.setVisible(true);
    } // Closes main method
} // Closes class
```

```
package divelog;
/**
 * This class creates the content on the
 * Welcome tabbed pane in the Dive Log
 * application.
 * @version 1.0
*/

import javax.swing.*; //imported for buttons, labels, and images
import java.awt.*; //imported for layout manager


public class Welcome extends JPanel
{ //Opening class

  // Variables for objects
 private JLabel jl;
 private JTextArea ta;
 private JLabel diver;

 public Welcome()

 { // Opens constructor

  setLayout(new BorderLayout());
  setBackground(Color.white);


  } // Closes constructor


  }//Closes class Welcome
```

```
package divelog;
/**
 * This class creates the content on the
 * Welcome tabbed pane in the Dive Log
 * application.
  * @version 1.0
*/

import javax.swing.*; //imported for buttons, labels, and images
import java.awt.*; //imported for layout manager


public class Welcome extends JPanel
{ //Opens class Welcome


   // Variables for objects that are created.
   // in the class constructor

   // Label that to hold the title and an image.
   private JLabel jl;

   // Variable for the text area.
   private JTextArea ta;

   // Label that to hold the image of a diver and fish.
   private JLabel diver;


    // Class constructor that provides instruction on
    // how the Welcome object is built.

    public Welcome()
    {  // Opens constructor

       // Sets the layout by instantiating a
       // BorderLayout container in the
       // setLayout method.

       setLayout(new BorderLayout());

       // Sets the background color for this Welcome
       // panel object.

       setBackground(Color.white);

       // The dive flag image is created by making an
       // instance of a JLabel on which an image ojbect
       // is initialized, calling the ImageIcon class
       // constructor.
       jl = new JLabel("Java(TM) Technology Dive Log",
            new ImageIcon("images/diveflag.gif"), JLabel.CENTER);

       // Sets the font face and size for title.
       jl.setFont(new Font("Times-Roman", Font.BOLD, 17));

       // Initialize a text area object that contains the text.
       // String concatenation is used to limit line length,
       // and \n creates new lines for a paragraph space.

       ta = new JTextArea("This application uses a" +
```

```
     " typical Graphical User Interface (GUI), featuring AWT layout "+
     "managers and Project Swing components, such as buttons, borders," +
     " text areas, menus, and more." +
     "\n\nIn addition, the dive log's functionality uses AWT event handlers" +
     ", methods to manipulate data, Java I/O" +
     " to save user input to files, and " +
     "special classes to include HTML pages with live links.");


     // Sets the font face and size for the text in the
     // text area. Line wrap is also set for the text
     // area, and the text area cannot be edited.

     ta.setFont(new Font("SansSerif", Font.PLAIN, 14));
     ta.setLineWrap(true);
     ta.setWrapStyleWord(true);
     ta.setEditable(false);

     // The following method creates a titled border
     // around the entire text area, using the BorderFactory
     // class.
     ta.setBorder(BorderFactory.createTitledBorder(
               " Welcome to the Java Technology Dive Log "));
      // Creates an image object on the label object
     diver = new JLabel("",
         new ImageIcon("images/diver.jpg"), JLabel.CENTER);
     // Each of the objects jl, ta, and diver are
     // added to the layout with the add method.
     // The objects are postioned with the constraints:
     // NORTH, CENTER, and SOUTH.
     // Note that no objects have been added to East
     // or West.

     add(jl, BorderLayout.NORTH);
     add(ta, BorderLayout.CENTER);
     add(diver, BorderLayout.SOUTH);

    }// Closes Welcome constructor

}// Closes class Welcome
```



A Java(TM) Technology Dive Log

File

Welcome

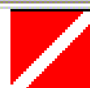
Diver Data

Log Dives

Statistics

Favorite Web Site

Resources



Java(TM) Technology Dive Log

Welcome to the Java Technology Dive Log

This application uses a typical Graphical User Interface (GUI), featuring AWT layout managers and Project Swing components, such as buttons, borders, text areas, menus, and more.

In addition, the dive log's functionality uses AWT event handlers, methods to manipulate data, Java I/O to save user input to files, and special classes to include HTML pages with live links.