Building an Application: Part 3: Receiving Input with Text Fields and Check Boxes, and Event Handling Basics

by Dana Nourie, *January 2002*

Contents

Preparing for User Input | NEXT>>Setting Up the Diver Data Pane

By now you should feel comfortable with objects and methods. You have initialized instances of predefined objects, such as panels and labels, and you have called methods on these objects using a variable, or identifier, and the dot operator. In addition, you have defined your own objects and methods to build GUI components.

More on User Input Components

An Example of Using Each Text Component

Text Components

In Building an Application, Part 2, the `Welcome` class introduced inheritance, illustrated how to display images and labels, and demonstrated how to arrange objects using a layout manager.

Part 3 reinforces these concepts and introduces the `Diver` class shows you how to:

- Retrieve and display user input.
- Understand encapsulation and access control.
- Arrange layouts within layouts.
- Understand event handling basics.
- Use inner classes.

Getting Started

In Part 1, you created the `DiveLog` class with 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. In Part 2, you designed the `Welcome` class, which appears on the `Welcome` pane.

For this part of the tutorial, you need one image and the `Diver.java` placeholder class that creates the Diver Data pane. You can use a different image than the one provided here, but to prevent problems with layout, make the image same size as the image provided.

Follow these steps...

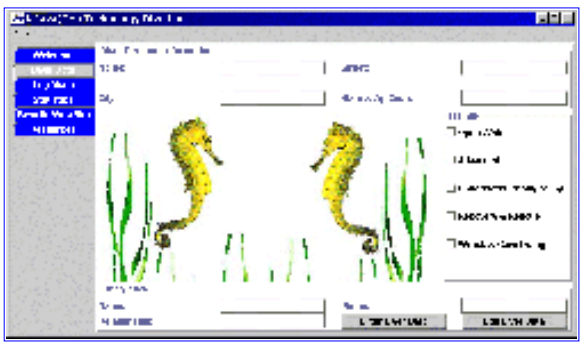
1. Save the following image to the `divelog/images` directory:
   - `2seahorses.gif`
     Image size: 450 by 275 pixels.

Or create an image of your own, but use the same pixel size as the one listed above.

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

Preparing for User Input

Click to enlarge



Completed Diver panel

Displaying images and labels is useful, but this aspect of an application is passive, only showing information. Most applications also require parts of an application to be active, accepting user input and doing something with the data received.

The Java API contains many types of predefined components you can use to collect user data. Here are a few:

- Text fields
- Text areas
- Check boxes
- Radio buttons
- Drop-down menus
- Buttons

When designing an application, consider the type of data to be entered by a user, and how to collect the information. A Dive Log generally contains the name of the diver. In addition, contact information is needed in case of emergency. Other personal data is valuable in a dive log as well, such as the diver's level of formal training.

The Diver Data pane accepts input so the diver can enter the following types of information:

Personal

- Name
- Address

Emergency Contact

- Name
- Relationship to that person
- Phone number

Formal Training

- List of various training levels

Remember the Java™ programming language is all about objects and manipulating data within those objects by calling methods. Listing out your application's data requirements helps you decide what kinds of objects you're going to need to create or initialize. The list above should give give you a good idea about the object types you need.

Data input for the list of above consists of strings, so use text fields to collect this information. The Java API has a predefined class to create text fields to collect user input.

You could also use a text field to request information about training, but that's not appropriate for a Dive Log, since training is not necessarily in a hierarchy. Instead, try check boxes. A diver can check each completed training course.

Once the user has entered data into the fields and checked the appropriate boxes, the user needs to be able to submit the information. The most common object used to signal the application to do something with the information entered is a button.

The Java API has a handy class for creating a button component. You'll learn more about `JTextField`, `JCheckBox`, and `JButton` soon.

For each item, you need an object to label the object that the user types in, and you need an object to collect the listed data. In other words, for each item above, you'll need at least two objects. The next section shows what is needed and how to handle the Diver Data tabbed pane.

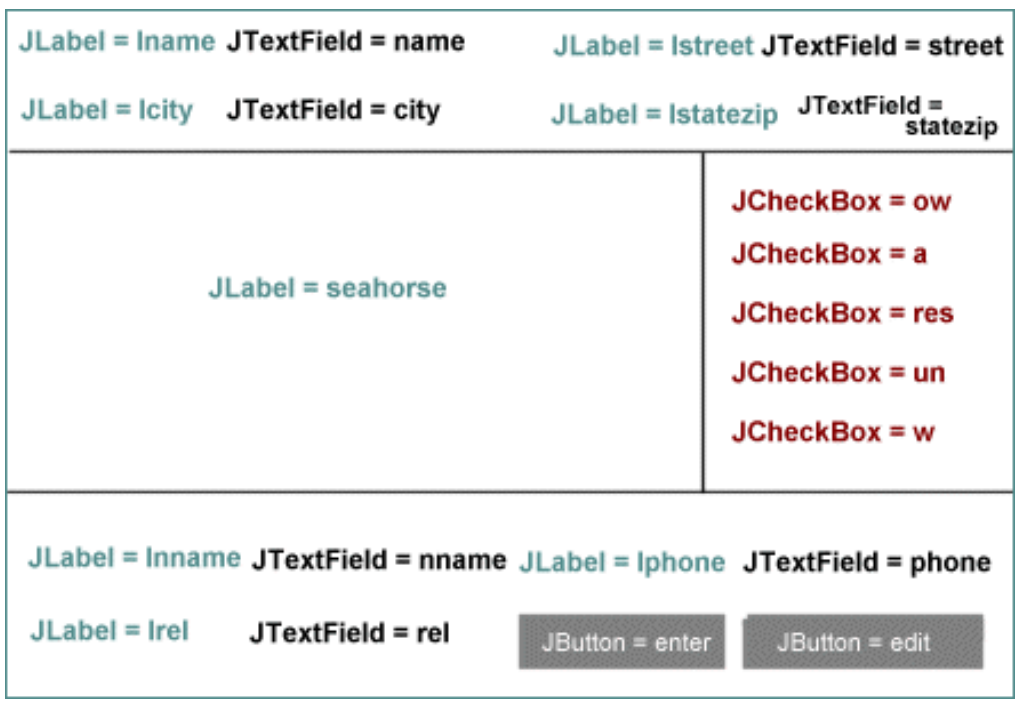
Planning the Diver Data Pane

To plan the Diver Data tabbed pane, itemize the objects you need and organize that data at the same time with a sketch. Use either paper and pencil, or a drawing program.

For now, list out the label, text field, and check box variable names you'll need, and organize them into a desired layout. This helps you decide:

- The objects to create
- Their reference variable names
- Where they should be positioned on the pane

Once this is completed, you can begin writing the code. The image below illustrates this planning concept:



Preliminary object and layout plan

Notice that:

- The object type is on the left of the = sign and the variable name on the right, much like you'd write it in code.
- Label objects are aquamarine.
- Text field objects are black.

- Check box objects are maroon.

This makes it easy to see what objects you need to create, therefore which classes you need to initialize, and how you want those object positioned on the pane.

So far there are a total of 22 objects on this pane. You may recall that the `BorderLayout` class allows only one object to be added to each region, totaling five objects in all.

Can you add 22 objects to this Diver Data pane, using the border layout manager?

- A. Yes
- B. No

Submit

Can you add 22 objects to this Diver Data pane, using the border layout manager?

Yes, you can add 22 objects (or more) to the Diver Data pane using the border layout by adding panels with their own layout managers to the regions of the border layout. Each panel then has additional objects, making it possible to create numerous objects on this pane. This lesson describes the details.

Setting Up the Diver Data Pane

More on Variables and Access Control

Declaring Member Variables

Basics of Objects

The Diver Data pane contains labels to signal what information users should enter in the text fields and check boxes. In addition, an image is included for aesthetic purposes. These objects are arranged so that personal diver data appears at the top of the pane, training information to the far right, the image to the center and left, and emergency information at the bottom. In addition to listing variable and object information, the sketch also reveals a layout pattern.

Layout of Diver Data Pane

The image to the left shows four basic areas that can easily become regions of the border layout. Since you have more than five objects to add, you need to create panels to hold these objects, and the panels are then added to the four regions. The layouts for each panel vary, depending on the requirements of the objects added.

Like the Welcome pane, the Diver Data pane starts with a plan, the `Diver` class. Use the placeholder `Diver.java` class you created in Part 1. This class should already use the keyword `extends` to make this pane a `JPanel` through inheritance.

Building the `Diver` class requires the following steps:

1. Declare the necessary variables for each label, text field, check box, and image object.
2. Declare a constructor.
   Within the constructor:
   1. Set the layout and background color.
   2. Initialize each object previously declared.
   3. Call methods to create each panel.
   4. Add each constructed panel to the border layout regions of the Diver pane.

3. Define the methods that construct each panel.
4. Create a class to handle the check boxes.

Since you've already listed the variables in a sketch, you can use this listing to declare the variables in the `Diver` class.

Follow these steps...

1. Open the `Diver.java` file in your text editor.
2. Add the following list of variables just after the opening curly brace of the class declaration:

```
// Text fields for user input
private JTextField name;
private JTextField street;
private JTextField city;
private JTextField statezip;

// Labels to go with each text field
private JLabel lname;
private JLabel lstreet;
private JLabel lcity;
private JLabel lstatezip;

// Check boxes for types of diver training
private JCheckBox ow;
private JCheckBox a;
private JCheckBox res;
private JCheckBox un;
private JCheckBox w;

// Text fields for Emergency box
private JTextField nname;
private JTextField phone;
private JTextField rel;

// Text fields for Emergency Contact
private JLabel lnname;
private JLabel lphone;
private JLabel lrel;

// Buttons and image
private JButton enter;
private JButton edit;
private JLabel seahorse;

// Panels to be built and added
// to the border layout of this
// panel.

private JPanel images;
private JPanel jaddress;
```

```
        private JPanel emerg;
        private JPanel training;
        // Class to handle functionality of check boxes
        ItemListener handler = new CheckBoxHandler();
        // ItemListener and CheckBoxHandler classes are
        // explained later.
```

3. Save the file.

Note that all the variables are declared `private`. This protects the data from being manipulated from outside classes. Declaring data `private` and some methods `public` is a part of a concept called encapsulation.

Encapsulation and Access Control

In Part 1 you learned a little about the access attributes: `private`, `public`, and `protected`. Access control plays a big part in encapsulation. Encapsulation refers to how to hide and protect data and methods within an object from outside interference and misuse. The class definition is the foundation for encapsulation.

Classes define the instructions for the type of access protection each data item and method has, and which methods have access to that data. In addition, classes working together takes advantage of hiding the workings of methods and objects. In other words, encapsulation aides you in building applications with predefined objects and objects of your own making.

You don't need to understand the mechanics of the predefined classes you use in the same way you don't need to be concerned with the workings of a radio you install in your car. A class from the Java API is a predefined component just as a radio is a predefined component.

In designing and writing the code for the Dive Log, you have already dealt with encapsulation in two ways:

1. **Hidden Implementation**:
   - Initializing objects from predefined classes
     Consider the `JFrame` object. By initializing or using the `extends` keyword to create a `JFrame` object, you created a frame with minimize and maximize buttons, and an area that can contain other objects. You likely have no knowledge how this object is built. The details of building the frame are hidden from you. All you did was create a frame using either the `new` or `extends` keyword with `JFrame`, calling the constructor necessary to build a frame the way you want it.
   - Calling predefined methods
     When you called `setBackground` and passed in the `Color` class, you didn't need to know how that method paints the background to the color field you chose. The `setBackground` method isn't defined in a class you created. Instead, `setBackground` comes from another predefined class. The instructions of this method are hidden from you. You only call the method on the object you want to add a color background to, and the method did the work for you.
2. **Access Control**
   - The predefined methods you've called so far have been declared `public`, giving you access to them.
   - The methods you defined in the Dive Log classes were also given `public` access.
   - The data variables you declared were given `private` access to protect them.

Safe programming practices encourages `private` access control on data variables. To access that data, you define `public` methods to either *get* the data or *set* the data. To prevent data from being manipulated, you declare the data `private` and do not provide access methods to that data.

The `setBackground` method is an example on an accessor method that allows you to *set* the background color of an object.

In creating the `Diver` class, you call predefined accessor methods, taking advantage of encapsulation. Later in this tutorial series, you define other types of access methods, some of which are declared `private` to protect data.

Defining the Constructor

With the variables declared, you can move onto defining the constructor for the `Diver` class. Here's how this constructor builds the Diver Data pane:

- Sets the border layout manager
- Sets a white background
- Initializes the objects for the pane
- Calls four methods to build four panels
- Adds the panels to the pane

Recall that the class constructor must have the same name as the class, and it does not return a value. It does provide the instructions for how the object is built on initialization, including other objects that need to be initialized. In addition, this constructor needs the `public` access modifer so that the `DiveLog` class can initialize this class when it calls the `Diver` constructor.

1. Open the `Diver.java` file in your text editor.
2. Begin the definition of the class constructor and calling the methods to set the layout manager and set the background color to white. Place the following code immediately after the variable declarations:

```
public Diver()
 { // Opens Constructor
   // Sets layout for Diver panel
   setLayout(new BorderLayout());
   // Sets background color
   setBackground(Color.white);


 } // Closes Constructor
```

3. Save the file.

The next step is to initialize each variable you previously declared at the top of the class.

How you initialize these objects depends on which constructor for that type you call. Remember, the keyword `new` signals the compiler that you are calling the constructor of the class you name. The documentation for each class defines constructors for each of these objects.

Construct a `JTextField` object that initializes with the text `Enter Your Name` and assign it to the variable `name`. Which line of code is correct?

- A. `name = JTextField("Enter Your Name");`
- B. `name = new JTextField(Enter Your Name);`
- C. Neither

Submit

Which line of code is correct?

Neither of those lines are correct. Line **A.** `name = JTextField("Enter Your Name");` is missing the keyword `new`. Line **B.** `name = new JTextField(Enter Your Name);` is missing the enclosing quotes around the `String`. Quotes tell the compiler that the characters are a part of a `String`.

The correct code is:

```
name = new JTextField("Enter Your Name");
```

Input Objects

More on Components

Class Component

Class Container

Class JComponent

The Java API has numerous classes for building input objects to collect user data. In this pane, use the following user input objects:

- `JTextField`
- `JCheckBox`
- `JButton`

Each of these objects accepts user information from text input to a click of a mouse button. What happens to the data once entered or chosen is up to you.

The Diver Data pane starts with simple input concepts and displays the user data on the screen. In Part 4 you learn about handling data in more complex ways, such as reading data from files.

The JTextField Class

Text fields are familiar to most users. You find them on paper forms and most applications use them to collect text data such as names, addresses, and so forth. To create a text field in your application, initialize an instance of the `JTextField` class like this:

```
name = new JTextField();
```

Initializing the `JTextField` object in this way creates and empty text field. Other constructors for the class allow you to create text fields with default text displayed within the text field, or you can set a specific size.

Since you are also providing labels for each text field, the default `JTextField` constructor is sufficient for each of the text fields in the Diver Data pane, though you'll see an example of a text field displaying text.

Follow these steps...

1. Open the `Diver.java` file in your text editor.
2. Initialize an instance of the `JTextField` class for each of the text field variables you set up earlier in the class:

```
// Initializes text fields for
// diver information
name = new JTextField("Enter Your Name");
street = new JTextField();
city = new JTextField ();
statezip = new JTextField ();

// Initializes text fields for
// emergency information
nname = new JTextField();
```

```
phone = new JTextField();
rel = new JTextField ();
```

3. Save the file.

Each of these text fields needs a label object so that the user knows what text to enter. Look at the list of variables you declared at the top of the class, then initialize `JLabel` label objects for each label item, providing the text to appear in the label.

Follow these steps...

1. Open the `Diver.java` file in your text editor.
2. After the list of `JTextField` initialized objects, initialize a `JLabel` object for each label, providing the necessary text as a `String`:

```
// Initializes labels for text fields
lname = new JLabel("Name: ");
lstreet = new JLabel("Street: ");
lcity = new JLabel ("City: ");
lstatezip = new JLabel("State & Zip Code: ");
lnname = new JLabel("Name: ");
lphone = new JLabel("Phone: ");
lrel = new JLabel ("Relationship: ");
```

3. Save the file.

So far, you declared the variables for the text fields and labels at the top of the class, and you initialized these objects within the constructor. Later, in the definition of the method that builds the panel these objects are added to, you call special accessor methods to *get* the text entered in the text fields.

The JCheckBox Class

Check boxes are another familiar tool used to collect user data, and they are especially useful when you have lists of items to display. Users intuitively know how to select or deselect the boxes.

The `JCheckBox` class has a number of constructors you can call to create boxes with or without text, boxes with icons, and boxes that are initially selected.

The Diver Data pane calls the `JCheckBox` constructor that includes text along with the box, eliminating the need for labels. The first box, Open Water, is initially selected. The constructor to call appears in the documentation as follows:

```
JCheckBox(String text, boolean selected)
```

This line instructs you to provide the text you want to appear as a `String` object, so the text must appear within a pair of quotes. In addition, you provide a boolean. A boolean is an expression that can only have a `true` or `false` value. In this case, you specify `true` or `false` depending if you want the box initially selected. The default is `false`.

To create a checked check box with identifying text, use the following format:

```
JCheckBox checkedBox;
checkedBox = new JCheckBox("Checked Box", true);
```

For a default check box without a check, use:

```
JCheckBox checkedBox;
checkedBox = new JCheckBox("Checked Box");
```

Or:

```
JCheckBox checkedBox;
checkedBox = new JCheckBox("Checked Box, false");
```

Follow these steps...

1. Open the `Diver.java` file in your text editor.
2. Initialize the `JCheckBox` items with the titles of each of the diving courses immediately following the initialization of the `JTextField` and `JLabel` objects. Have the first check box, Open Water, checked on initialization:

```
// Initializes checkboxes with titles
ow = new JCheckBox("Open Water", true);
a = new JCheckBox("Advanced");
res = new JCheckBox("Recovery & Rescue");
un = new JCheckBox("Underwater Photography");
w = new JCheckBox("Wreck & Cave Diving");
```

3. Save the file.

Note: If you're reformatting this application into an exercise log, or some other tool, change the strings for the check boxes. For instance, instead of diving course titles, supply past diets or programs you've tried.

The JButton Class

After users enter text, they need to signal the application to retrieve and do something with the data. Buttons are commonly used for this purpose. In addition, a user may want to edit entered data, and again, a button is a good way for a user to signal this need.

You create buttons by initializing an instance of the `JButton` class. This class has constructors that build buttons with text, icons, or both. You create a button with text similiarly to a `JLabel`:

```
JButton button1;
button1 = new JButton("Button 1");
```

This code creates a button, but the button doesn't do anything when you click it.

Follow these steps...

1. Open the `Diver.java` file in your text editor.
2. After the lines of code that initialized the check boxes, initialize two buttons: one for entering information, one for editing.

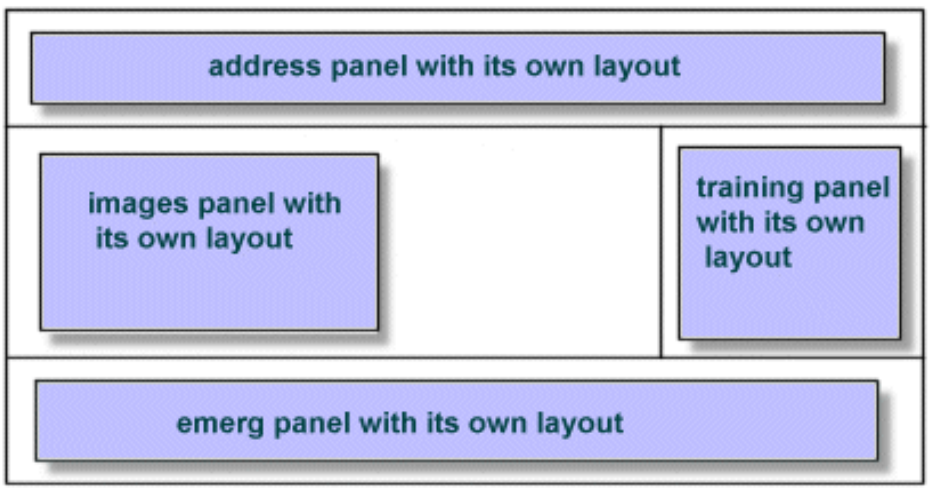
```
// Initialize buttons and image
enter = new JButton("Enter Diver Data");
edit = new JButton("Edit Diver Data");
```

3. Initialize the image object, using the `2seahorses.jpg` you downloaded earlier.

```
seahorse = new JLabel("",
    new ImageIcon("images/2seahorses.jpg"),
                              JLabel.CENTER);
```

4. Save the file.

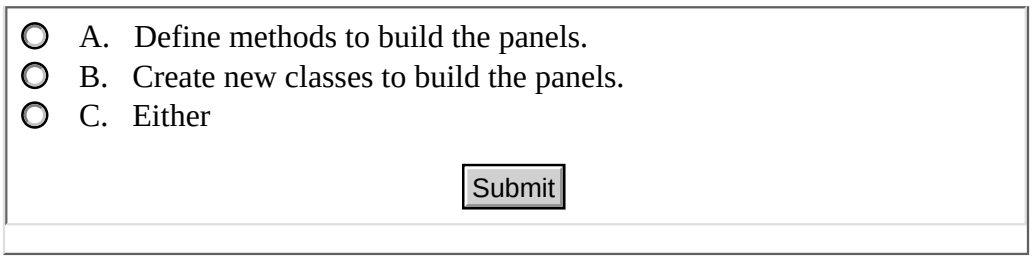
All the objects to go on the Diver Data pane are now initialized within this class constructor. However, you don't add them directly to the `Diver` panel object. Instead add them to four panels that have their own layouts, then add each panel to the layout of this main `Diver` panel object.



Panels added to class `Diver`

The panels have not yet been created. You could do that here within the constructor, but that would clutter the constructor and make it difficult to read.

How should you build the four panels to be added to the Diver Data pane?



How should you build the four panels to be added to the Diver Data pane?

You can create separate classes and initialize each in the class constructor, or you can define methods to build the panels within this class and call those methods from the class constructor.

For the Diver Data pane, you define methods to build four panels. Later, you'll learn to create new classes for various sections of a pane.

Constructor Completion

You did a lot of work within the constructor, but it isn't complete yet. The plan for this constructor was:

1. Set the layout and background color.
2. Initialize each object previously declared.
3. Call methods to create each panel.
4. Add each constructed panel to the border layout regions of the Diver pane.

More on Constructors

Providing Constructors for Your Classes

Creating Objects

Classes, Objects, and Constructors

Steps one and two are complete. Now you begin step three: Calling methods to create each panel.

Naming convention recommends that method names begin with a verb describing what the method does. In this case, the methods you're calling in the constructor build panels to hold objects. So it makes sense to begin the names with the word *build*. In fact, the method names can specify which panel each method builds. For instance, the `buildImagePanel` builds the panel that displays the sea horses image.

To call the methods within the constructor, simply name them and provide empty parenthesis (). Some methods require parameters to go within the parentheses, but these particular methods don't require additional information passed into them.

Follow these steps...

1. Open the `Diver.java` file in your text editor.
2. Add the following method calls to the constructor:

```
// Calls method to build image panel, which
// is defined outside of the constructor.
buildImagePanel();

// Calls method to build address panel, which
// is defined outside of the constructor.
buildAddressPanel();

// Calls method to build emerg panel, which
// is defined outside of the constructor.
buildEmergencyPanel();
```

```
// Calls method to build training panel, which
// is defined outside of the constructor.
buildTrainingPanel();
```

3. Save the file.

From top to bottom, the constructor creates each object, calls the methods that build the panels, and adds those objects to each panel. The final step in completing the constructor brings you back to the `Diver` panel object. You add the panels to the border layout regions of the `Diver` object.

As you may recall from adding objects to the Welcome pane, to add the panel objects to the `Diver` panel layout regions you:

- Call the add method
- Supply the following parameter list:
  - Variable name of the object to add
  - Name of the layout class
  - Location constant

To get the list of panel variables you had set up previously, either look to the top of your class, or to the preliminary drawing you made.

Follow these steps...

1. Open the `Diver.java` file in your text editor.
2. Add the four panels:`jaddress`, `images`, `training`, `emerg` to four regions of the border layout manager as follows:

```
add(jaddress, BorderLayout.NORTH);
add(images, BorderLayout.CENTER);
add(training, BorderLayout.EAST);
add(emerg, BorderLayout.SOUTH);
```

3. Save the file.

Your constructor should look like this example.

Instances of Classes

If you compile the `Diver` class at this point, you get errors. Although you have called methods to build panels, you haven't defined those methods. With all four steps of the constructor completed, now you *move* outside the constructor to define the methods needed to build the panels. Since you have called four methods, you must define the following four methods:

- `buildImagePanel`
- `buildAddressPanel`
- `buildEmergencyPanel`
- `buildTrainingPanel`

Because the `Diver` class extends `JPanel`, it is a panel and you've set it to use the border layout manager. By building panels with their own layout managers, you can add many more objects to the Diver Data pane, built from the `Diver` class.

Though each pane is not created in its own class but instead in its own method, you can't use inheritance through the

`extends` keyword. However, you can create a panel object by creating an instance of the `JPanel` class. Once you have a panel object, set a layout for it, and add objects as you like.

First, begin the method definition with an accessor attribute, the method type, and the method name, followed by empty parentheses:

```
private void buildImagePanel()
```

Because only this class needs access to this method, you can safely declare the method `private`. The method type is `void` because this method does not return a value. Instead, it simply builds a few GUI objects. Using naming convention for the Java programming language, this method is named `buildImagePanel` to clearly describe what the method does. This helps you as well as any other developer looking at the code understand what the method does at a glance.

In the beginning of this class you declared a variable called `images` of type `JPanel`. Now assign this variable to an instance of the `JPanel` class using the keyword `new` to create a `JPanel` object:

```
images = new JPanel();
```

Because you created an instance of the `JPanel` class, you have access to methods that `JPanel` inherits, including the `setLayout` method from the `Container` class, and the `setBackground` from the `Component` class. Set the background color and the layout on this instance of the `JPanel` object.

Since only one image is added to this panel, a simple layout manager like the flow layout manager suffices. You set this layout similiarly as you did for the Welcome and Diver panes by calling the `setLayout` method, but this time you also name the object that needs the layout manager using the dot operator. Then initialize the proper class for the layout you want in the parameter list of the method:

```
images.setLayout(new FlowLayout);
```

You've created a panel object that uses an instance of the `FlowLayout` class as its layout manager. Next, set the background color as you did for the other panel, but this time you must name the panel object and the dot operator:

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

The images panel background is now white.

Why didn't you have to create an instance of the `Color` class using the keyword `new` in the `setBackground` method as you did in the `setLayout` method?

- A. The `Color` class isn't a real class.
- B. The `Color` class provides `static` fields.
- C. Neither

Submit

Why didn't you have to create an instance of the `Color` class using the keyword `new` in the `setBackground` method as you did in the `setLayout` method?

If a field or method is `static`, you do not need an instance of the class for access. Instead, you need only name the class that contains the field or method, followed by the dot operator and the name of the method or field you want to use.

Instance and Class Members

When you created the variables `name` and `street`, you created instance variables. This means that in order to use these

variables, you must create an instance of the `Diver` class. If you created several instances of the `Diver` class, each instance would contain its own copy of the data variables `name` and `street`.

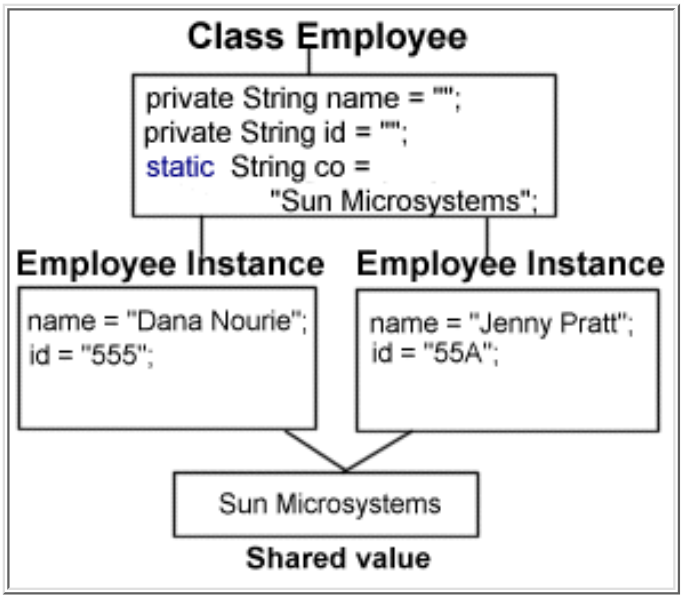
Instance and Class Members

Understanding Instance and Class Members

Details of a Method Declaration

Controlling Access to Members of a Class

Sometimes you don't want each instance to have its own copy. There are times when you may need objects to use a class variable, or rather a `static` variable for the following reasons:



- To force an object to share data rather than having its own copy.
  For instance, a counter might be shared by several class instances. Every time a new instance of a class is initialized, the counter increments by 1, sharing that data with each instance of the class. If you incorrectly used an instance variable instead, then each new class instance might reset the counter variable to 0, count the initialization as 1, and always give the incorrect count as 1, instead of however many initializations there really are.
- To initialize data before or as an object is created.
  For instance, you might want an `account` object to check the balance of the account before initializing the `account` object. Or maybe you just want to get the current date before assigning it to an object.
- To allow access to data without having to instantiate its class.
  The `Color` and `Math` classes have fields and methods you can use without having to create an instance of those classes.

When you read through the API documentation, you know youonly need to name the class followed by the dot operator if it has `static` fields or methods.

So far the only `static` method you created was the main method in the `DiveLog` class. The reason `main` is `static` is because is must be called before any objects exist, since it is the entry point for an application.

Static methods do have a few restrictions:

- Can only call other `static` methods
- Must only access `static` data
- Cannot refer to `this` or `super`

In the next tutorial part, you learn how to define `static` members. For now, know when you are using `static` members, and recognize when they are available to you by reading the API documentation.

Now, that the panel is initialized, the background set to white, and the `FlowLayout` class initialized as the layout manager, you just need to add the image object you initialized in the constructor to this panel.

Follow these steps...

1. Open the `Diver.java` file in your text editor.
2. Add the following method just outside the closing constructor bracket:

```
// This method creates a panel called images.
private void buildImagePanel()
 { // Opens method
   // Initializes a new JPanel object.
   images = new JPanel();
   // Sets the color, layout, and adds the
   // seahorse object.
   images.setLayout(new FlowLayout() );
   images.setBackground(Color.white);
   images.add(seahorse);
 } // Closes method
```

3. Save the file.

Building the Grid Layout

The next method to define builds the Diver Data address panel. As with the previous method, you create an instance of the `JPanel` class, then later add the text field and label objects to the panel. Because there are eight objects to add to this panel, the border layout is not appropriate. Instead, you use the grid layout manager.

The grid layout manager is created by initializing the `GridLayout` class as a parameter in the `setLayout` method:

```
setLayout(new GridLayout)
```

The grid layout allows you to add objects onto a rectangular grid of rows and columns. You define how many rows and columns when you initialize the `GridLayout` class. In addition, you can add vertical and horizonal gaps:

```
// Sets the grid to 3 rows and 2 columns
setLayout(new GridLayout(3,2));
// Sets the grid to 5 rows, 4 columns, and
// verical and horizontal gaps of 5
setLayout(new GridLayout(5,4,5,5));
```

Refer to your sketch to decide how many rows and columns you need, and experiment with the `Vgap` and `Hgap` values to decide how much, if any padding you need around the objects.

Notice the grid layout for the address panel on the Diver Data pane:

Address objects added to a grid layout

When you add the objects to the panel that uses the grid layout manager, objects are added in the order you code them. Objects are inserted from left to right in the first row, then left to right in the second row, and so forth. You don't specify a region as you did with the border layout manager. All components added to the grid layout are given equal size. For the address objects, it's not a problem.

The `GridLayout` class has accessor `set` and `get` methods if you need to manage a layout dynamically. For simplicity, hardcode the number of rows and columns for this panel using the `GridLayout` constructor in the `setLayout` method.

Follow these steps...

1. Open the `Diver.java` file in your text editor.
2. Begin the `buildAddressPanel` method as follows:

```
private void buildAddressPanel ()
 { // Opens method
  jaddress = new JPanel();
  // Sets color and layout.
  // Adds the textfields and labels for
  // diver input.
  jaddress.setBackground(Color.white);
  jaddress.setLayout( new  GridLayout(2, 4, 20, 20) );
  //Adds each component to the panel
 jaddress.add(lname);
 jaddress.add(name);
 jaddress.add(lstreet);
 jaddress.add(street);
 jaddress.add(lcity);
 jaddress.add(city);
 jaddress.add(lstatezip);
 jaddress.add(statezip);


 } // Closes method
```

3. Save the file.

So far the `buildAddressPanel` method does the following:

- Assigns an instance of the `JPanel` class to the variable you set up earlier `jaddress`.

- Sets the background of the `jaddress` panel object to white, using the dot operator and the variable identifier.
- Creates an instance of the `GridLayout` class with two rows, four columns, and a value of `20` for `Vgap` and `Hgap` to use as the layout manager, using the `setLayout` method.
- Adds the label and text fields objects to the the panel in the order they need to appear.

Since the Diver Data panel consists of several panels requesting different types of information, providing text to identify this particular area improves usability. You could use a simple label object to label this area of the panel, but a border groups the components neatly.

Here is a border creation example:

```
JPanel pane = new JPanel();
     pane.setBorder(
        BorderFactory.createLineBorder(Color.black));
```

The object to have the border is named first, then the `setBorder` method, inherited from `JComponent`, is called. Pass the `BorderFactory` class as a parameter, and call the method you want to use, using the dot operator. Note that you are not creating an instance of the `BorderFactory` class. Instead, one of its methods is called with the dot operator.

How can you get to the `BorderFactory` methods without creating an instance of the class? As you learned previously, through `static` methods. The `BorderFactory` class has a number of static methods available to use simply by using the class name, the dot operator, then calling the method for the border you want to create.

To create a colored line border, as the code above demonstrates, the `createLineBorder` needs you to pass in the `Color` class with one of its `static` fields for the color of your choosing.

In the Welcome pane you created a titled border. Do so again for the Diver Data address panel.

Follow these steps...

1. Open the `Diver.java` file in your text editor.
2. Create a titled border in the `buildAddressPanel` method as follows:

```
jaddress.setBorder(BorderFactory.createTitledBorder(
                        "Diver Personal Information"));
```

3. Save the file.

So far you've created the panel, set the background to white, set the grid layout manager, added the components, and drew a border around them. If you compile, the panel should appear as you set it up. However, if you type into the text fields and press Return. Nothing happens. You have the basics of the GUI, but no functionality.

Software without functionality is of limited use. This Diver Data pane is intended to do more than display information. Once the user enters the requested information and clicks the Enter button or presses Return, the data display to the screen. So far you've only set up the components to accept data, now you must set up the components to listen for events, then handle those events. Event handling makes GUI software functional.

To add functionality to components, you return to the top of the `Diver` class.

Follow these steps...

1. Open the `Diver.java` file in your text editor.
2. Add the import statement:

   ```
   import java.awt.event.*;
   ```
3. Change the class declaration from:

   ```
   public class Diver extends JPanel
   ```

   to:

   ```
   public class Diver extends JPanel
                       implements ActionListener
   ```
4. Save the file.

The code in bold is added in the class declaration. It can be written on one line, but is split here because of web page constraints.

What does `implements ActionListener` mean?

- A. The `Diver` class promes to define methods named in the `ActionListener` interface.
- B. The `Diver` class inherits a second class called `ActionListener`.
- C. Neither

Submit

What does `implements ActionListener` mean?

When you write `implements ActionListener` into your class declaration, you promise to implement, or provide the code for, any methods that are declared in the `ActionListener` interface. In other words, any methods declared in `ActionListener` have to be defined in your class, the class implementing `ActionListener`. This is explained below.

Classes Versus Interfaces

Classes are templates for objects. They define the type of data the object will contain and the method instructions to operate on that data. An instance of that class is the template filled in with the data and calls to the methods--the instance is the object.

You can create many instances of the same class, creating many objects of that type. Each object has its own data, and some objects share specific data, depending on whether the data has been declared `static` or `instance` data.

More about Interfaces

What Is an Interface?

Implementing an Interface

Implementing Listeners for Commonly Handled Events

You do not have to write a class from scratch. If a class already exists that has similar features as a class you want to create, you can use the `extends` keyword to inherit fields and methods from another class, then add more fields and methods to create a richer class than the parent class. But you're not stuck with every inherited field or method. You can override methods, or hide data of the parent

class to suit the needs of the child class.

This is true of concrete classes, but not so for interfaces.

Interfaces declare methods, and possibly static fields, but they do not define the methods. Instead, interfaces only declare methods with absolutely no instruction included. In other words, interfaces are like templates that always remain templates. You cannot instantiate an interface to create an object. They are like classes without implementation. So why do they exist?

They serve a purpose: To force the developer to provide those methods, with details, in the class that `implements` the interface. In other words, implementing an interface means you are making a promise to use certain methods, but you, the developer define, the details of those methods.

Why is that helpful or necessary?

Suppose you have a team of developers who are creating classes to make different types of animal objects. All these animals are going to have two things in common, they:

- Use some form of locomotion
- Eat some kind of food

The difference between the animals lies in how each one moves around and what or how it eats. In other words, they each need to have the methods `locomotion()` and `eat()`, but each individual, animal class defines the details of those methods separately based on the needs of the species of animal.

You design an interface to ensure each animal object does certain things, and your team develops classes as follows:

Animal Interface

```
public interface Animal
 {
    public void locomotion();
    public void eat();
 }
```

Class Shark

```
public class Shark implements Animal
 {
    public void locomotion()
    {
      System.out.println("I swim.");
    }
    public void eat()
    {
      System.out.println("I hunt for seals.");
    }
 }
```

Class Dog

```
public class Dog implements Animal
 {
   public void locomotion()
    {
      System.out.println("I run on four legs.");
    }
   public void eat()
    {
      System.out.println("I eat kibble.");
    }
 }
```

```
public class  AnimalTest
 {
   public static void main(String[] arg)
     {
       Shark shark = new Shark();
       shark.locomotion();
       shark.eat();
       Dog dog = new dog();
       dog.locomotion();
       dog.eat();
     }
 }
```

Note: Constructors are left out of these classes for brevity, so the default no argument constructors are called by the compiler.

In the Animal interface, methods are declared, but not defined. Notice that those methods (in blue) are defined in the concrete classes `Shark` and `Dog`. In the above example, each method prints a line of text, indicating what the animal eats or how it moves from one place to another. The last class is an application that initializes the two concrete classes `Shark` and `Dog`, and calls the methods from those classes, using the dot operator.

When the `AnimalTest` is compiled and run, you get the following results:

```
I swim.
I hunt for seals.
I run on four legs.
I eat kibble.
```

Software games are another example of how you might implement interfaces. For instance, you might design the following games: Water Pistol Wars, Asteroid Archery, Rubberband Rally, and Cannon Craze. All involve firing a weapon, but the weapons and what they shoot are different. An interface would ensure that each game implemented a `fire()` method, but it is up to each game designer as to what and how each type of ammunition is fired.

Event Handling Basics

The Java API has many interfaces you can implement. For GUI programming, an interface ensures that you provide some functionality in a specific method. To make the Dive Log interactive and do something with text entered into text fields, you implement the `ActionListener` interface and handle the events.

Before going into details about the `ActionListener` interface and some of its subclasses, you need to understand the basics of event handling.

Events are objects that a user initiates, such as text entered, a button pushed, or a mouse moved over a component. The component must *listen* for an event if you want to *handle* the event with action, such as displaying text or writing to a database. Creating an event-driven class takes three steps:

1. Decide which type of event is fired by a component, and implementing the right interface.
2. Register the component as a listener for that type of event.
3. Implement handler methods from the interface.

When a user fires an event through a GUI component, a method call is made to any objects that have been registered with the component as listeners for the type of event that took place. An event listener is an object that is notified when an event occurs. Ten categories of GUI events represent a different class.

To create the `Diver` class, you need only be concerned with the following event listeners and interfaces:

Event Class	Interface Listener	Handler Methods
`ActionEvent`	`ActionListener`	`actionPerformed(ActionEvent)`
`ItemEvent`	`ItemListener`	`itemStateChanged(ItemEvent)`

Events generated from buttons and text fields need the class to implement the `ActionListener` interface, which you have done for the `Diver` class. Next, you need to register the components you want to listen for events with the `addActionListener` method, which takes an event object as an argument. Registration methods are named addxxxListener, where xxx is the type of interface implemented. For instance, to register a text field to listen for events, you use the following:

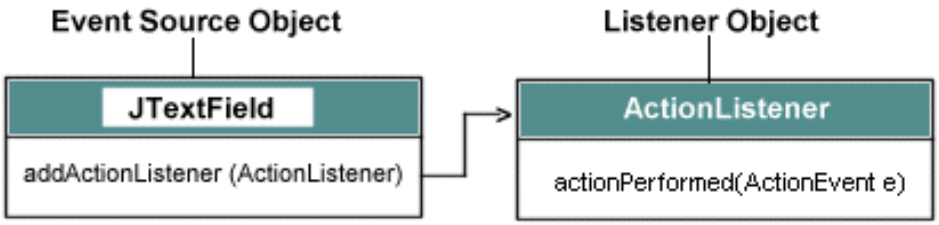
```
JTextfield textfield1 = new JTextfield();
textfield1.addActionListener( this );
```

The keyword `this` refers to this class, being the `Diver` class, that `implements` the proper interface. Once the class implements an `ActionListener`, and you've called the `addListener` method on the input object you want to listen for events, you can define the method you must implement from the `ActionListener` interface.

Recall that since `ActionListener` is an interface, you are promising to implement its methods. The documention reveals only one method you need to define:

```
public void actionPerformed(ActionEvent e)
```

This method is invoked when an event occurs, such as a button clicked or the Return key pressed. You define what you want to happen within this method, whether it is displaying text, writing to files, or some other action.



Event handling process

The image above shows the event handling process for a text field.

Follow these steps...

1. Open the `Diver.java` file in your text editor.
2. Register your text field components as listeners with the `addActionListener` method. Pass in the keyword `this` as a parameter so the compiler knows to look in this class for the method that handles the events. Add the following to the `buildAddressPanel` method, just before the closing curly brace:

```
//Listeners for each text field in the
name.addActionListener( this );
street.addActionListener( this );
city.addActionListener( this );
statezip.addActionListener( this );
nname.addActionListener( this );
phone.addActionListener( this );
rel.addActionListener( this );
```

3. Save the file.

Your `Diver.java` file should look like this example.

Next, you write the method to build the `emerg` panel. You construct this `buildEmergencyPanel` method as you did the other two:

Follow these steps...

1. Create an instance of a `JPanel`:

```
emerg = new JPanel();
```

2. Set the layout manager and background color:

```
emerg.setLayout( new  GridLayout(2, 4, 20, 0) );
emerg.setBackground(Color.white);
```

3. Add each object (labels, text fields, and buttons) to the panel:

```
emerg.add(lnname);
emerg.add(nname);
emerg.add(lphone);
emerg.add(phone);
emerg.add(lrel);
emerg.add(rel);
emerg.add(enter);
emerg.add(edit);
```

4. Create a border for each botton and the panel:

```
emerg.setBorder(BorderFactory.createTitledBorder("Emergency"));
edit.setBorder(BorderFactory.createRaisedBevelBorder());
```

5. Register the text fields with the appropriate listeners:

```
nname.addActionListener( this );
phone.addActionListener( this );
rel.addActionListener( this );
```

6. Register the buttons with the appropriate listeners:

```
enter.addActionListener( this );
edit.addActionListener( this );
```

Your `buildEmergencyPanel` method should look something like this example.

The `buildTrainingPanel` method you need to construct next uses check boxes instead of text fields. Each check box is constructed with text, so you needn't create labels.

The layout of this panel can easily make use of a simple grid that consists of 1 column and 6 rows. This example also supplies padding of 10 and 20 respectively.

Check box event handling requires you to implement the `ItemListener` interface. So to register the check boxes as event sources, you add a listener by calling the `addItemListener` method.

Instead of using the keyword `this` as a parameter in the `addItemListener` method to tell the method to look in *this* class for instruction on how to handle these events, you tell it to look to the reference variable `handler`.

Recall that you added this piece of code to the top of the `Diver` class:

```
// Class to handle functionality of check boxes
```

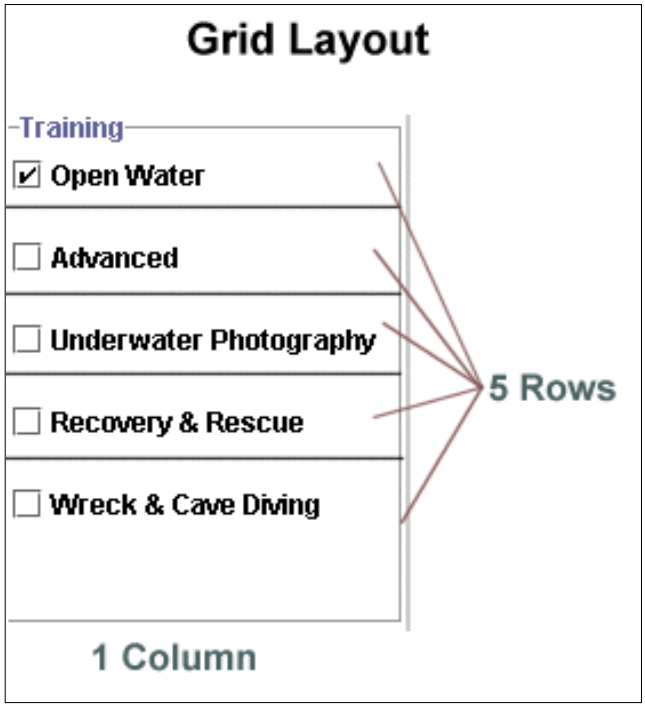
```
ItemListener handler = new CheckBoxHandler();
```

When you create another class to handle the functionality of objects, name the class as a parameter in the `addItemListener` method. You'll learn more about this inner class later.

To register a listener with the `ow` object, you write the following:

```
ow.addItemListener(handler);
```

When you specify that the instruction is in *this* class, or you provide the name of a class, you reveal is where the promised method is located. Remember, for event handling functionality, you implement certain interfaces. In doing so, you promise to provide the implementation, or details, to the methods declared in that interface. So the instructions are specifically enclosed in those methods you promised to provide.



The `Diver` class implements the `ActionListener` interface. Therefore, *this* class must implement the `actionPerformed(ActionEvent e)` method. When an event occurs, the `actionPerformed` is automatically invoked because you registered the object by calling the `add`*`Type`*`Listener`.

The `Diver` class does not implement the `ItemListener` interface. Instead, the inner class implements the `ItemListener` interface and its required method implementation.

Follow these steps...

Complete the `buildTrainingPanel` in the `Diver.java` file:

1. Create an instance of a `JPanel`:

   ```
   training = new JPanel();
   ```

2. Set the layout manager and background color, using the grid layout manager:

   ```
   training.setBackground(Color.white);
   training.setLayout(new GridLayout(5, 1, 10, 20 ) );
   ```

3. Add each check box object to the panel, and set the background color of each to white:

   ```
   training.add(ow).setBackground(Color.white);
   training.add(a).setBackground(Color.white);
   training.add(un).setBackground(Color.white);
   training.add(res).setBackground(Color.white);
   training.add(w).setBackground(Color.white);
   ```

4. Create a border for the panel:

   ```
   training.setBorder(BorderFactory.createTitledBorder("Training"));
   ```

5. Register the check box objects with the appropriate listener method, supplying the reference variable `handler` as a parameter:

   ```
   ow.addItemListener(handler);
   a.addItemListener(handler);
   ```

```
    un.addItemListener(handler);
    res.addItemListener(handler);
    w.addItemListener(handler);
```

Your `Diver` class should look something like this example.

So far you've done the following in the `Diver` class:

- Implemented the `ActionListener` interface.
- Declared variables for objects.
- Instantiated those objects in the class constructor.
- Wrote methods to create panels that contain the GUI objects.
- Registered some of those objects to be event sources.

Now, you need to provide the instruction for what happens when a user presses Return or clicks the Enter button within the method you promised to implement.

Which method defines the details for the functionality of pressing Return or clicking the Enter button?

- A. `buttonReturnFunctionality`
- B. `actionPerformed`
- C. Neither

Submit

Which method defines the details for the functionality of pressing Return or clicking the Enter button?

Because you're implementing the `ActionListener` interface, you are promising to provide the details for the `actionPerformed` method. When the user fires an event from a text field or button, the `actionPerformed` method is automatically called. It's within this method that you write the details for what you want to happen when the user presses Return or clicks the Enter button.

Adding Functionality

The GUI components for this pane have been built, and now you're ready to force action when data is entered as requested.

You've registered the text fields and button as event listeners using the `addActionListener` method, and you've implemented the `ActionListener` interface, making the `Diver` class an `ActionListener` object and promising to provide the `actionPerformed` method in *this* class, meaning the `Diver` class. Now you work out the details for that method.

More about Program Control

Control Flow Statements

Blocks and Statements

Conditional Execution

Defining the actionPerformed method

When you implement a method from an interface, you must use the identical signature. In other words, if the interface's method is `public`, `void`, and accepts a `String` as an parameter, then so should the method you implement.

Here is the signature of the method you must implement:

```
public void actionPerformed(ActionEvent e)
```

To implement the `actionPerformed` method, you must give it `public` access, it must not return a value, and therefore is `void`, and it must access an `ActionEvent` object as a parameter. The `e` is just a variable name for the `ActionEvent`

object, so you can change that if you want.

Follow these steps...

1. Open the `Diver.java` file in your text editor.
2. Add the following method signature right after the closing curly brace of the `buildTrainingPanel` method:

```
public void actionPerformed(ActionEvent evt)
  { // Opens method
  ...
  } // Closes method
```

3. Save the file.

Where the `...` appears, you define the code for the functionality of this pane.

Program Flow and Control Statements

Like the GUI, the functionality of an application needs a plan:

- Decide what you want the application to do.
- Decide which components are directly involved in those actions.
- Sketch out in pseudo code how these actions are carried out.

The Diver Data pane does the following:

- Accepts user input with text fields.
- Accepts user input with check boxes.
- If Return is pressed or the Enter button is clicked, displays user input and hides the input boxes.
- If Edit is clicked, displays user input boxes again.

You've completed the first two items and are ready to write pseudo code for the last two. First, break these two items into steps:

- If Return is pressed or the Enter button is clicked:
  - Display user input
  - Hide the input boxes
- If Edit is clicked:
  - Display user input boxes again

When you write code, the program runs one line after another, unless you tell it to do otherwise. Sometimes you need the code to run a block of code while a certain condition exists, and at other times you'll want to make certain the condition exists before the next block is run. This process is what is referred to as program flow and control.

In the case of the Diver Data pane, text displays only if the Return key is pressed or if the Edit button is clicked. If neither happens, then the application doesn't need to do anything. On the other hand, if the Edit button is clicked, make certain the the text fields are visible and ready to accept user input.

Below is the same paragraph you just read, but note the words in blue. They give clues as to what kinds of control statements you should use in your code:

> In the case of the Diver Data pane, text displays only if the Return key is pressed or if the Edit button is clicked. If neither happens, then the application doesn't need to do anything. On the other hand, if the Edit button is clicked, make certain the the text fields are visible and ready to accept user input.

Using `if` statements

An `if` statement is a conditional branch statement to control the flow of execution that is only known at run time. In an `if` statement, a condition or expression is tested. If the condition is `true`, then the statement in that block is executed. If the expression is `false`, you can provide statements to execute, or nothing happens. The expression to be tested must be enclosed in `()`.

This is the general form of the `if` statement:

```
if(condition)
statement;
 else
   statement;
```

You can have numerous `if` statements, and you can use nested `if` statements. If more than one statement appears after the `if` or `else`, you must use curly braces. The keyword `else` is optional:

```
if(condition)
 {
  statement1;
  statement2;
  statement3;
  }
 else
  {
   statement;
  }
```

Boolean logical operators

Another important word you may have noted in that paragraph is the word **or**. In the Diver Data pane, you need to test for two possible conditions to return `true`:

- If Return is pressed
  OR
- If Enter is clicked

In the Java programming language, logical operators operate on boolean values. For instance:

```
||  OR
&&  AND
!=  Not equal to
```

There are more, but you only need to be concerned with the ones listed here for now. Reword the pseudo code and add some real code:

```
if (data is entered) || (Enter button is clicked)
   {
    display the text to the screen
    hide the text fields
   }
if (Edit is clicked)
   {
   Make the text fields visible
   }
```

Now you have the basics of how you control the flow. Next, you need to understand what should go in as the conditions. Obviously the pseudo code `data is entered` is not going to work. What you are really testing is which of the text fields contains text entered by the user. You hooked up each text field as an event listener. When the user enters the text, an event is *fired* by pressing Return. You must tell your application to *get* that data and *set* that data to display it.

All event subclasses, in this case `ActionEvent`, inherits the `getSource` from the `EventObject` class. This method returns the object that fired the event.

In other words, if text is entered into a text field object called `company`, then `company` is returned. Now that you know which text field fired an event, you retrieve the text entered. To retrieve and display the data use two methods from the `JTextComponent` class:

- `getText` method retrieves the data
- `setText` sets the text

Remember since these methods are inherited, you don't need an instance of the `JTextComponent` class to access these methods. And then you assign that text to a `JLabel` object to display it.

As an example:

```
// The evt variable is from the ActionListener
// object that was passed in the
// actionPerformed method.
if (evt.getSource() == company)
// Create a variable to hold the text
// you retrieve.
String companyText = name.getText();
// Assign the text to a label to
// display the text.
JLabel companyLabel = new JLabel("");
companyLabel.setText("Company:    "
                        + companyText);
// Hides the text field of the company
// text field object.
company.setVisible(false);
```

This code works if the user presses the Return key, but what about an Enter button? In the code above, that scenario is not accounted for, but it's easily added with the `||` operator in the first line:

```
if ((evt.getSource() == company) ||
                    (evt.getSource() == enter))
...
```

The code is saying if Return is pressed in the text field object `company` or the `enter` object is returned, then do the following. Note the extra `()`. Each condition is enclosed within its own set of `()`, then the group is enclosed in another set of `()`. Be careful not to leave out one of the parenthesis.

The Diver Data pane also has an Edit button, which sets the text fields to visible. That bit of code can go into an `if` statement of its own. To make the text fields visible again, call the `setVisible` method and set it to `true`.

Follow these steps...

1. Open the `Diver.java` file in your text editor.
2. Within the `actionPerformed` method, write if statements to retrieve the text field object name using the `getSource` method.
3. Call the `getText` method on each of the text field objects to retrieve the text.
4. Assign the text to a `JLabel` object to display to the screen.
5. Call the `setVisible` method on each text field object and set it to `false` to hide the text fields.

Here's an example:

```
if ((evt.getSource() == name) ||
                 (evt.getSource() == enter))
     {// Opens if statement
      // Retrives the text from the name
      // text field and assigns it to the
      // variable nameText of type String.
      String nameText = name.getText();
      lname.setText("Name:    " + nameText);
      //After printing text to JLabel, hides
      // the textfield:
      name.setVisible(false);
      }// ends if statement
```

- Provide the control statements for the Edit button. Remember that if the Edit button is clicked, then the text fields are visible again.

Here is an example:

```
if (evt.getSource() == edit)
     { // Opens if statement

   // If edit button has been pressed
   // the following is set to visible
    name.setVisible(true);
    }// Closes if statement
```

- Save the file. Your `actionPerformed` method should look something like this example.

This completes the functionality for the text fields and the buttons, but does not define the functionality of the check boxes. You can address the check boxes by having the Diver class implement the `ItemListener` class, and defining the `itemStateChanged` method in this class as you did for the text fields and buttons. However, there is a better way.

The benefit of using object oriented programming is that it allows you to build software and make changes without having to change other parts of the code. You can change one object without causing problems in other objects because objects interact through methods.

The Diver Data pane defines GUI objects and keeps the functionality of those object in methods. This is good object oriented programming, but you can improve this model by confining the functionality to an inner class.

Do inner classes have direct access to instance variables and methods of the outer class?

- A. Yes
- B. No

Submit

Do inner classes have direct access to instance variables and methods of the outer class?

An inner class has the special privilege of unlimited access to its enclosing class' members, even if they're declared `private`, and the inner class relies on the enclosing class for its function. By using an inner class for event handling, you enclose the code for functionality neatly within an inner class, while using the outer class for the GUI components.

Inner Classes

Inner classes are defined within an enclosing class. In this case, the enclosing class is the `Diver` class and the inner class is the `CheckBoxHandler` class. There are four types of inner classes:

- Simple inner classes
- Inner classes within methods
- Anonymous inner classes
- Static inner classes

The Diver class encloses only a simple inner class. Later, you'll learn about anonymous inner classes.

More about Inner Classes

Inner Classes

Chapter 7: Inner Classes

How do inner classes work?

An instance of an inner class can only exist with an instance of the enclosing class. Using an inner class enforces the relationship between two classes, so another class cannot access this inner class without creating an instance of the enclosing class. In addition, the inner class has direct access to the instance variables and methods of the enclosing class without having to reference a handle or variable.

Inner classes can be defined as `public`, `private`, or `protected`. In the case of this `CheckBoxHandler` inner class you can give it `private` access since it is only used in context with the `Diver` class anyway. Inner classes can extend other classes as well as implement as many interfaces as necessary.

For example:

```
private class GUIEventHandler implements ActionLister
```

Because the `CheckBoxHandler` class is going to define the functionality of the check boxes, it needs to implement the `ItemListener` interface. As with the `ActionListener` interface, you must implement any methods that `ItemListener` declares. Again, there is only one method to be concerned with:

```
itemStateChanged(ItemEvent e)
```

This method is invoked when an item has been selected or deselected. The code written for this method performs the operations that need to occur when an item is selected or deselected. When a check box is selected a check appears in the box. When deselected, the check disappears. To add to this, the Diver Data pane check box text turns blue when a check box is selected, and returns to the default color black when deselected.

Before delving into the pseudo code to plan the functionality, declare the inner class within the Diver class and provide the

promised method declaration.

Follow these steps...

1. Open the `Diver.java` file in your text editor.
2. After the closing curly brace of the `actionPerformed` method, declare an inner class called `CheckBoxHandler` that implements the `ItemListener` interface:

```
private class CheckBoxHandler implements ItemListener
{ // Opens inner class
```

3. Declare a `itemStateChanged` method, using the correct signature provided in the Java API documentation:

```
public void itemStateChanged (ItemEvent e)
```

4. Save the file.

Pseudo code

List what should happen when the check boxes are selected or deselected:

If a check box is selected:

- Determine which check box is selected
- Make the text blue for that box

Or else if the box is deselected:

- Make the text black

You can see at a glance that an `if` statement can handle the functionality, but this pseudo code reveals another keyword `else`. You could use several `if` statements, but using `else` shortens the lines of code. The `if/else` statement says do this if this occurs, or else do that.

To find out which check box fired an event, and then change the text for that box to blue, you use the `getSource` and `getStateChange` methods. The `getSource` method returns the object that fired an event, and the `getStateChange` method tells you if the event was a box being `SELECTED` or `DESLECTED`.

So you only have to be concerned with boxes that are selected, call the `getSource` method, which returns objects that fired an event. Check if the event is `SELECTED`, then assign it to a variable that references a `JCheckBox` object.

Because you aren't instantiating the `JCheckBox` class and creating a new object but are instead assigning a returned object and typing it as a `JCheckBox` object, you must explictly *cast* it as a `JCheckBox` object. You do this with the following:

```
JCheckBox source = (JCheckBox) e.getSource();
```

This line of code calls `getSource`, which returns an object that fired an event, casts it as a `JCheckBox` and assigns it to `source`, which is of type `JCheckBox`.

Next, the `getStateChange` method compares the object with a constant of the `ItemEvent` class that tells if the box is selected. These constants are `static`, so you need only use the class name and dot operator:

```
e.getStateChange() == ItemEvent.SELECTED
```

Now you can adjust your pseudo code more accurately with some real code:

```
JCheckBox source = (JCheckBox) e.getSource();
   if ( e.getStateChange() == ItemEvent.SELECTED )
      source.setForeground(Color.blue);
   else
      source.setForeground(Color.black);
```

Assigning and casting returned objects, then using the if/else statement allows you to write instruction for any check box that returns an object. The GUI is separate then from the functionality so you can add or remove more check boxes without having to change the code for the funcitonality.

Follow these steps...

1. Open the `Diver.java` file in your text editor.
2. Implement the `itemStateChanged` method, providing instruction for each check box object: check to see if the object equals each of your variables, and if it is selected:

```
JCheckBox source = (JCheckBox) e.getSource();
   if ( e.getStateChange() == ItemEvent.SELECTED )
      source.setForeground(Color.blue);
   else
      source.setForeground(Color.black);
```

3. Save the file.

Your class should match this Diver.java file.

Now you have an inner class that provides the functionality of the input components, while the enclosing class creates the GUI features that are presented to the user. If you need to make changes to the functionality, you edit the inner class without affecting the enclosing class. Likewise, if you make changes to the enclosing the class the inner class remains unaffected unless you add components that also need functionality added.

Because the inner class only works with an instantiation of the enclosing class, it is fairly protected from other classes accessing it. Using inner classes also makes it easier to read the code. You know to work on the enclosing class for GUI features, or the inner class for functionality.

Summary

Part 3 of the Dive Log tutorial reviewed Java programming concepts from Part 1 and Part 2 and introduced new ones.

User Input Objects

- Text fields and check boxes are common components used to collect user data.
- Pre plan with pencil and paper or a drawing program to decide how you want to arrange user input components. Then decide which layout manager to use.
- You can add many object to a pane by building panels that use their own layouts.

Encapsulation and Access Control

- Encapsulation works through hidden implementation and access modifiers.
- Special methods are used to access data. These methods *get* or *set* data.

Event Handling Basics

- Button and mouse clicks are examples of *events* that are fired from components.

- To handle events, a class must implement the proper interface.
- To register a component as an event listener, call the addxxxListener and pass in the proper event class that implements the promised method that defines the functionality.

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 4: Using pull-down menus, adding scrollbars to a text area, and reading from files.

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:

Submit Reset

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.

[This page was updated: 22-Jan-2002]

```
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" +
```

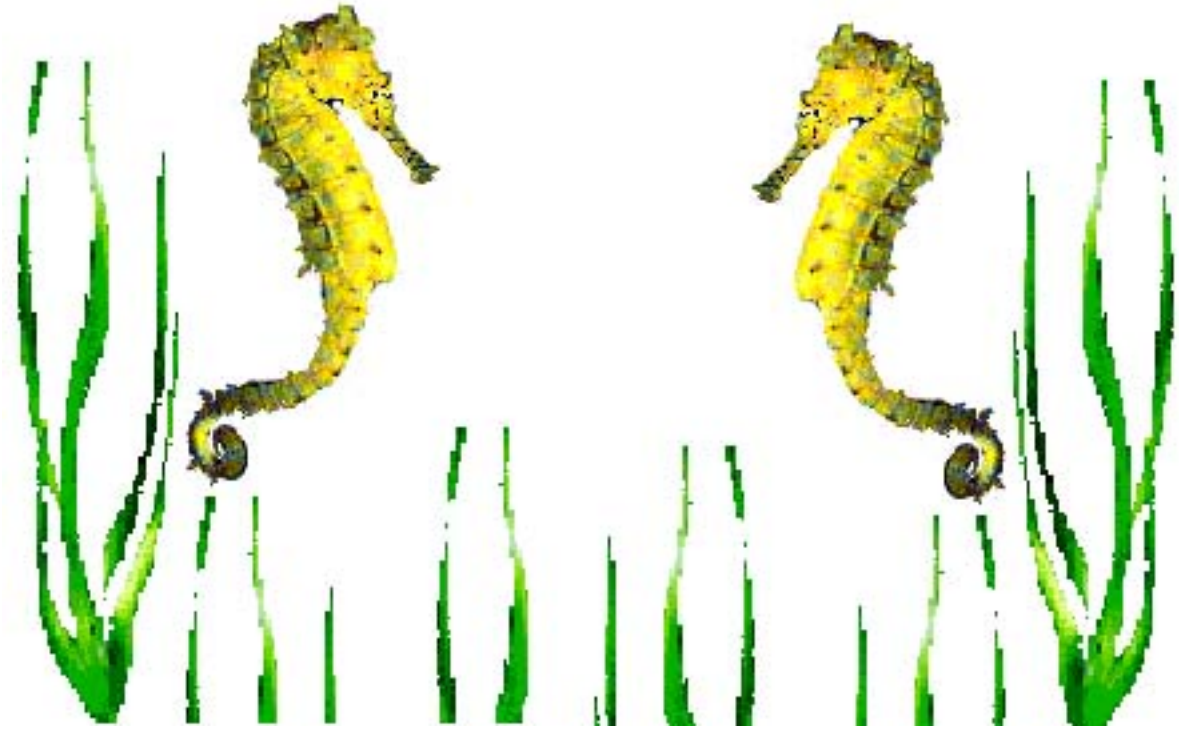
```
     " typical Graphical User Interface (GUI), featuring AWT layout "+
     "managers and Project Swing components, such as buttons, borders," +
     " text areas, menus, and more." +
     "\n\nIn addition, the dive log's functionality uses AWT event handlers" +
     ", methods to manipulate data, 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
```

```
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;

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



public class Diver extends JPanel
{ // Opens class


} // Closes class
```



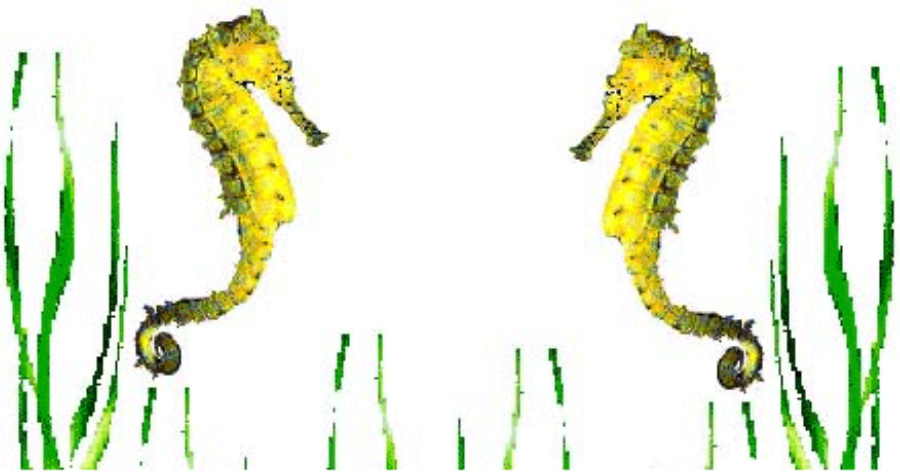
A Java(TM) Technology Dive Log

File

Welcome
Diver Data
Log Dives
Statistics
Favorite Web Site
Resources

Diver Personal Information

Name:
Street:
City:
State & Zip Code:



Training

- [ ] Open Water
- [ ] Advanced
- [ ] Underwater Photography
- [ ] Recovery & Rescue
- [ ] Wreck & Cave Diving

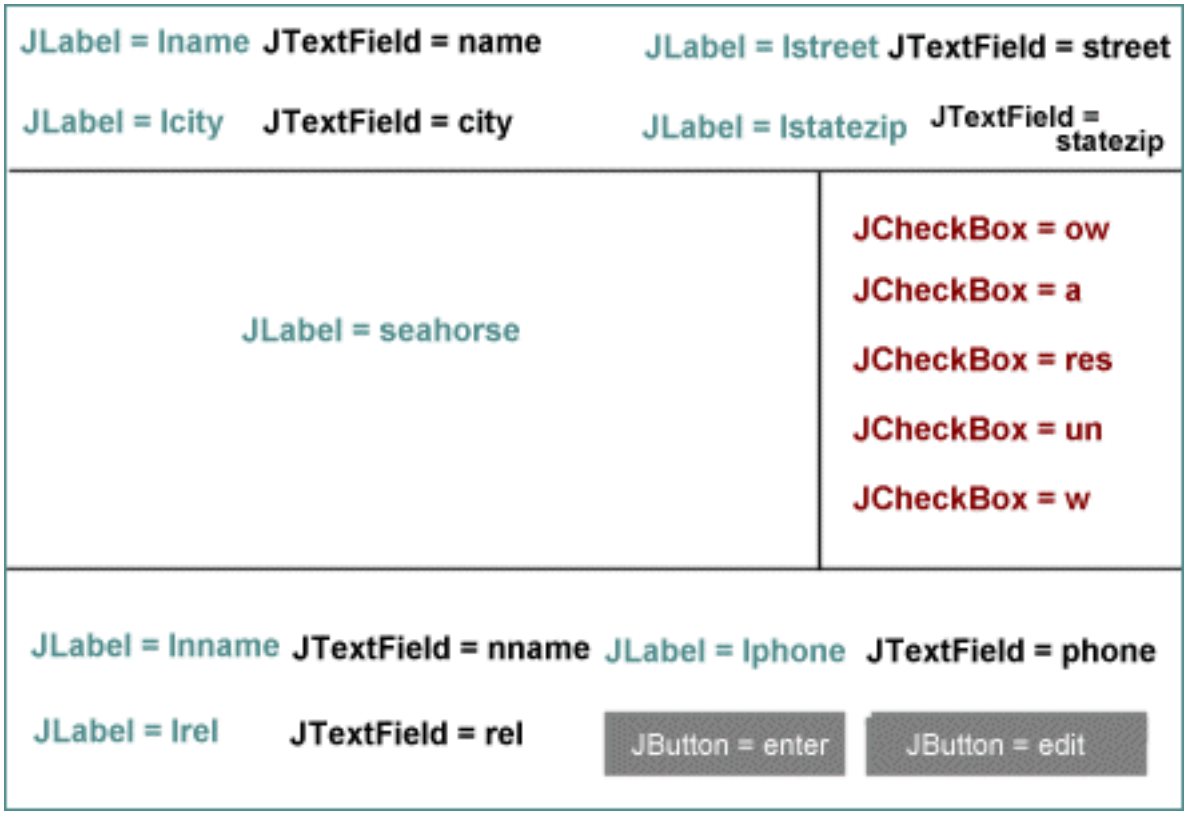
Emergency

Name:
Phone:
Relationship:

Enter Diver Data
Edit Diver Data

```
public Diver()
  { // Opens Constructor
   // Sets layout for Diver panel
   setLayout(new BorderLayout());
   // Sets background color for
   // Diver panel
   setBackground(Color.white);

   // Initializes text fields for
   // diver information
   name = new JTextField("Enter Your Name");
   street = new JTextField();
   city = new JTextField ();
   statezip = new JTextField ();

   // Initializes text fields for
   // emergency information
   nname = new JTextField();
   phone = new JTextField();
   rel = new JTextField ();

   // Initializes labels for textfields
   lname = new JLabel("Name: ");
   lstreet = new JLabel("Street: ");
   lcity = new JLabel ("City: ");
   lstatezip = new JLabel("State & Zip Code: ");
   lnname = new JLabel("Name: ");
   lphone = new JLabel("Phone: ");
   lrel = new JLabel ("Relationship: ");


   // Initializes checkboxes with titles
   ow = new JCheckBox("Open Water", true);
   a = new JCheckBox("Advanced");
   res = new JCheckBox("Recovery & Rescue");
   un = new JCheckBox("Underwater Photography");
   w = new JCheckBox("Wreck & Cave Diving");


   // Initialize buttons and image
   enter = new JButton("Enter Diver Data");
   edit = new JButton("Edit Diver Data");
   seahorse = new JLabel("",
        new ImageIcon("images/2seahorses.jpg"),
                                JLabel.CENTER);
   // Calls method to buid image panel, which
   // is defined outside of the constructor
   buildImagePanel();

   // Calls method to buid address panel, which
   // is defined outside of the constructor
   buildAddressPanel();

   // Calls method to buid emerg panel, which
   // is defined outside of the constructor
   buildEmergencyPanel();

   // Calls method to buid training panel, which
   // is defined outside of the constructor
   buildTrainingPanel();
```

```
        //The methods called above build the panels, then this
       // call to add adds each panel to the main panel's
       // border layout manager.

        add(jaddress, BorderLayout.NORTH);
        add(images, BorderLayout.CENTER);
        add(training, BorderLayout.EAST);
        add(emerg, BorderLayout.SOUTH);


    } // Ends constructor
```


JLabel = lname
JTextField = name
JLabel = lstreet
JTextField = street
JLabel = lcity
JTextField = city
JLabel = lstatezip
JTextField = statezip
JLabel = seahorse
JCheckBox = ow
JCheckBox = a
JCheckBox = res
JCheckBox = un
JCheckBox = w
JLabel = lnname
JTextField = nname
JLabel = lphone
JTextField = phone
JLabel = lrel
JTextField = rel
JButton = enter
JButton = edit

```
package divelog;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.Color.*;



public class Diver extends JPanel implements ActionListener
{ // Opens class


     // Fields for Diver Personal Information

     // Text fields for user input
     private JTextField name;
     private JTextField street;
     private JTextField city;
     private JTextField statezip;

     // Labels to go with each text field
     private JLabel lname;
     private JLabel lstreet;
     private JLabel lcity;
     private JLabel lstatezip;

     // Check boxes for types of diver training
     private JCheckBox ow;
     private JCheckBox a;
     private JCheckBox res;
     private JCheckBox un;
     private JCheckBox w;

     // Textfields for Emergency box
     private JTextField nname;
     private JTextField phone;
     private JTextField rel;

     // Textfields for Emergency Contact
     private JLabel lnname;
     private JLabel lphone;
     private JLabel lrel;

     // Buttons and image
     private JButton enter;
     private JButton edit;
     private JLabel seahorse;

     // Panels to be built and added
     // to the border layout of this
     // panel.

     private JPanel images;
     private JPanel jaddress;
     private JPanel emerg;
     private JPanel training;

     // Class to handle functionality of checkboxes
     ItemListener handler = new CheckBoxHandler();
```

```
   // Class constructor that builds the necessary
   // labels and text fields images, buttons,
   // and panels.

        public Diver()
          { // Opens Constructor
           // Sets layout for Diver panel
           setLayout(new BorderLayout());
           // Sets background color for
           // Diver panel
           setBackground(Color.white);

           // Initializes textfields
           name = new JTextField("Enter Your Name");
           street = new JTextField();
           city = new JTextField ();
           statezip = new JTextField ();
           // Initializes labels for textfields
           lname = new JLabel("Name: ");
           lstreet = new JLabel("Street: ");
           lcity = new JLabel ("City: ");
           lstatezip = new JLabel("State & Zip Code: ");
           // Initializes checkboxes with titles
           ow = new JCheckBox("Open Water", true);
           a = new JCheckBox("Advanced");
           res = new JCheckBox("Recovery & Rescue");
           un = new JCheckBox("Underwater Photography");
           w = new JCheckBox("Wreck & Cave Diving");

           // Initializes textfields for emergency panel
           nname = new JTextField();
           phone = new JTextField();
           rel = new JTextField ();
           // Initializes labels for textfields
           lnname = new JLabel("Name: ");
           lphone = new JLabel("Phone: ");
           lrel = new JLabel ("Relationship: ");
           // Initialize objects
           enter = new JButton("Enter Diver Data");
           edit = new JButton("Edit Diver Data");
           seahorse = new JLabel("",
                new ImageIcon("images/2seahorses.jpg"),
                                        JLabel.CENTER);
           // Calls method to buid image panel, which
           // is defined outside of the constructor
           buildImagePanel();

           // Calls method to buid address panel, which
           // is defined outside of the constructor
           buildAddressPanel();

           // Calls method to buid emerg panel, which
           // is defined outside of the constructor
           buildEmergencyPanel();

           // Calls method to buid training panel, which
           // is defined outside of the constructor
           buildTrainingPanel();

         //The methods called above build the panels, then this
        // call to add adds each panel to the main panel's
```

```
          // border layout manager.

           add(jaddress, BorderLayout.NORTH);
           add(images, BorderLayout.CENTER);
           add(training, BorderLayout.EAST);
           add(emerg, BorderLayout.SOUTH);


        } // Ends constructor


        // This method creates a panel called images
        private void buildImagePanel()
        { // Opens method
          // Instantiates a new JPanel object
           images = new JPanel();
          // Sets the color, layout, and adds the
          // seahorse object
           images.setLayout(new FlowLayout() );
           images.setBackground(Color.white);
           images.add(seahorse);
          } // Closes method

        private void buildAddressPanel ()
        { // Opens method
          jaddress = new JPanel();
          // Sets color and layout.
          // Adds the textfields and labels for
          // diver input.
          jaddress.setBackground(Color.white);
          jaddress.setLayout( new  GridLayout(2, 4, 20, 20) );
          //Adds each component to the panel
          jaddress.add(lname);
          jaddress.add(name);
          jaddress.add(lstreet);
          jaddress.add(street);
          jaddress.add(lcity);
          jaddress.add(city);
          jaddress.add(lstatezip);
          jaddress.add(statezip);

          // Sets a border around the panel, including
          // a title
          jaddress.setBorder(BorderFactory.createTitledBorder(
                                    "Diver Personal Information"));

          //Listeners for each text field in the
          name.addActionListener( this );
          street.addActionListener( this );
          city.addActionListener( this );
          statezip.addActionListener( this );
          nname.addActionListener( this );
          phone.addActionListener( this );
          rel.addActionListener( this );

        } // Closes method
}// Closes class
```

```
private void buildEmergencyPanel()
 { // Opens method
  //Create another panel for emergency input fields
    emerg = new JPanel();
    emerg.setLayout( new  GridLayout(2, 4, 20, 0) );
    emerg.setBackground(Color.white);

    emerg.add( lnname);
    emerg.add( nname);
    emerg.add( lphone);
    emerg.add( phone);
    emerg.add( lrel);
    emerg.add( rel);
    emerg.add(enter);
    emerg.add(edit);
   //Creates titled border around emerg panel
    emerg.setBorder(BorderFactory.createTitledBorder("Emergency"));

    //Adds an actionListener for Emergency Info
    nname.addActionListener( this );
    phone.addActionListener( this );
    rel.addActionListener( this );


    //Creates borders around the Edit and
    // Enter Diver Data buttons
    enter.setBorder(BorderFactory.createRaisedBevelBorder());
    edit.setBorder(BorderFactory.createRaisedBevelBorder());

     //Listeners for the buttons
     enter.addActionListener( this );
     edit.addActionListener( this );
    } //Closes method
```

```
package divelog;
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
import java.awt.Color.*;



public class Diver extends JPanel implements ActionListener
{ // Opens class


     // Fields for Diver Personal Information

     // Text fields for user input
     private JTextField name;
     private JTextField street;
     private JTextField city;
     private JTextField statezip;

     // Labels to go with each text field
     private JLabel lname;
     private JLabel lstreet;
     private JLabel lcity;
     private JLabel lstatezip;

     // Check boxes for types of diver training
     private JCheckBox ow;
     private JCheckBox a;
     private JCheckBox res;
     private JCheckBox un;
     private JCheckBox w;

     // Textfields for Emergency box
     private JTextField nname;
     private JTextField phone;
     private JTextField rel;

     // Textfields for Emergency Contact
     private JLabel lnname;
     private JLabel lphone;
     private JLabel lrel;

     // Buttons and image
     private JButton enter;
     private JButton edit;
     private JLabel seahorse;

     // Panels to be built and added
     // to the border layout of this
     // panel.

     private JPanel images;
     private JPanel jaddress;
     private JPanel emerg;
     private JPanel training;

     // Class to handle functionality of checkboxes
     ItemListener handler = new CheckBoxHandler();
```

```
   // Class constructor that builds the necessary
   // labels and text fields images, buttons,
   // and panels.

        public Diver()
          { // Opens Constructor
           // Sets layout for Diver panel
           setLayout(new BorderLayout());
           // Sets background color for
           // Diver panel
           setBackground(Color.white);

           // Initializes textfields
           name = new JTextField("Enter Your Name");
           street = new JTextField();
           city = new JTextField ();
           statezip = new JTextField ();
           // Initializes labels for textfields
           lname = new JLabel("Name: ");
           lstreet = new JLabel("Street: ");
           lcity = new JLabel ("City: ");
           lstatezip = new JLabel("State & Zip Code: ");
           // Initializes checkboxes with titles
           ow = new JCheckBox("Open Water", true);
           a = new JCheckBox("Advanced");
           res = new JCheckBox("Recovery & Rescue");
           un = new JCheckBox("Underwater Photography");
           w = new JCheckBox("Wreck & Cave Diving");

           // Initializes textfields for emergency panel
           nname = new JTextField();
           phone = new JTextField();
           rel = new JTextField ();
           // Initializes labels for textfields
           lnname = new JLabel("Name: ");
           lphone = new JLabel("Phone: ");
           lrel = new JLabel ("Relationship: ");
           // Initialize objects
           enter = new JButton("Enter Diver Data");
           edit = new JButton("Edit Diver Data");
           seahorse = new JLabel("",
                new ImageIcon("images/2seahorses.jpg"),
                                        JLabel.CENTER);
           // Calls method to buid image panel, which
           // is defined outside of the constructor
           buildImagePanel();

           // Calls method to buid address panel, which
           // is defined outside of the constructor
           buildAddressPanel();

           // Calls method to buid emerg panel, which
           // is defined outside of the constructor
           buildEmergencyPanel();

           // Calls method to buid training panel, which
           // is defined outside of the constructor
           buildTrainingPanel();

         //The methods called above build the panels, then this
        // call to add adds each panel to the main panel's
```

```
    // border layout manager.

     add(jaddress, BorderLayout.NORTH);
     add(images, BorderLayout.CENTER);
     add(training, BorderLayout.EAST);
     add(emerg, BorderLayout.SOUTH);


  } // Ends constructor


  // This method creates a panel called images
  private void buildImagePanel()
  { // Opens method
     images = new JPanel();
    // Sets the color, layout, and adds the
    // seahorse object
     images.setBackground(Color.white);
     images.setLayout(new FlowLayout() );
     images.add(seahorse);
    } // Closes method

  private void buildAddressPanel ()
  { // Opens method
    jaddress = new JPanel();
    // Sets color and layout.
    // Adds the textfields and labels for
    // diver input.
    jaddress.setBackground(Color.white);
    jaddress.setLayout( new  GridLayout(2, 4, 20, 20) );
    //Adds each component to the panel
    jaddress.add(lname);
    jaddress.add(name);
    jaddress.add(lstreet);
    jaddress.add(street);
    jaddress.add(lcity);
    jaddress.add(city);
    jaddress.add(lstatezip);
    jaddress.add(statezip);

    // Sets a border around the panel, including
    // a title
    jaddress.setBorder(BorderFactory.createTitledBorder(
                              "Diver Personal Information"));

    //Listeners for each text field in the
    name.addActionListener( this );
    street.addActionListener( this );
    city.addActionListener( this );
    statezip.addActionListener( this );
    nname.addActionListener( this );
    phone.addActionListener( this );
    rel.addActionListener( this );

  } // Closes method

  private void buildEmergencyPanel()
   { // Opens method
    //Create another panel for emergency input fields
      emerg = new JPanel();
      emerg.setLayout( new  GridLayout(2, 4, 20, 0) );
```

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

          emerg.add( lnname);
          emerg.add( nname);
          emerg.add( lphone);
          emerg.add( phone);
          emerg.add( lrel);
          emerg.add( rel);
          emerg.add(enter);
          emerg.add(edit);
         //Creates titled border around emerg panel
          emerg.setBorder(BorderFactory.createTitledBorder("Emergency"));

          //Adds an actionListener for Emergency Info
          nname.addActionListener( this );
          phone.addActionListener( this );
          rel.addActionListener( this );


          //Creates borders around the Edit and
          // Enter Diver Data buttons
          enter.setBorder(BorderFactory.createRaisedBevelBorder());
          edit.setBorder(BorderFactory.createRaisedBevelBorder());

           //Listeners for the buttons
           enter.addActionListener( this );
           edit.addActionListener( this );
          } //Closes method

      private void buildTrainingPanel()
      { //Opens method

        // Creates a panel for training courses

         training = new JPanel();
         training.setBackground(Color.white);
         training.setLayout(new GridLayout(5, 1, 10, 20 ) );
         //sets backgrounds of components to white
         training.add(ow).setBackground(Color.white);
         training.add(a).setBackground(Color.white);
         training.add(un).setBackground(Color.white);
         training.add(res).setBackground(Color.white);
         training.add(w).setBackground(Color.white);

        //Sets a titled border around training panel
        training.setBorder(BorderFactory.createTitledBorder("Training"));
        //Adds listeners to checkbox items
          ow.addItemListener(handler);
          a.addItemListener(handler);
          un.addItemListener(handler);
          res.addItemListener(handler);
          w.addItemListener(handler);
       } //Closes method
} // Closes class Diver
```

```
public void actionPerformed(ActionEvent evt)
  { // Opens method
  // Checks if the button clicked was the
  // Enter Diver Data button.
  // If not, moves on to next if statment.
  // Otherwise it enters this if statement
  if ((evt.getSource() == name) || (evt.getSource() == enter))
       { // Opens if statement
        //Retrives the text from the name text field and
        //assigns it to the variable nameText of
        //type String
        String nameText = name.getText();
        lname.setText("Name:    " + nameText);
        //After printing text to JLabel, hides the textfield
        name.setVisible(false);
        }// ends if statement
  if ((evt.getSource() == street) || (evt.getSource() == enter))
       { // Opens if statement
        String streetText = street.getText();
        lstreet.setText("Street:    " + streetText);
        street.setVisible(false);
         }// ends if statement
   if ((evt.getSource() == city) || (evt.getSource() == enter))
       { // Opens if statement
        String cityText = city.getText();
        lcity.setText("City:  " + cityText);
        city.setVisible(false);
             }// ends if statement
   if ((evt.getSource() == statezip) || (evt.getSource() == enter))
       { // Opens if statement
        String statezipText = statezip.getText();
        lstatezip.setText("State & Zip:     " + statezipText);
        statezip.setVisible(false);
             }// ends if statement
  if ((evt.getSource() == nname) || (evt.getSource() == enter))
       { // Opens if statement
        String relname = nname.getText();
        lnname.setText("Name:   " + relname);
        nname.setVisible(false);
              }// ends if statement
  if ((evt.getSource() == phone) || (evt.getSource() == enter))
       { // Opens if statement
        String relphone = phone.getText();
        lphone.setText("Phone:   " + relphone);
        lphone.setForeground(Color.red);
        phone.setVisible(false);
        }
  if ((evt.getSource() == rel) || (evt.getSource() == enter))
  {
        String relString = rel.getText();
        lrel.setText("Relationship:   " + relString);
        rel.setVisible(false);


    // If Edit button was clicked, set textfields to
    // visible for user to reenter information and
    // it sets the text that had been entered to
    // to an empty String, giving the appearance
    // that the text has been removed.
    } // Closes if statement
     if (evt.getSource() == edit)
```

```
        { // Opens else if

      // If edit button has been pressed
      // the following is set to visible
       name.setVisible(true);
       street.setVisible(true);
       city.setVisible(true);
       statezip.setVisible(true);

       // Relative's info

       nname.setVisible(true);
       phone.setVisible(true);
       rel.setVisible(true);

       lnname.setText("Name:   ");
       lnname.setForeground(Color.black);
       lphone.setText("Phone:   ");
       lphone.setForeground(Color.black);
       lrel.setText("Relationship:   ");
       lrel.setForeground(Color.black);

    } // Ends if

  } // Ends actionPerformed method
```

```
package divelog;
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
import java.awt.Color.*;



public class Diver extends JPanel implements ActionListener
{ // Opens class


     // Fields for Diver Personal Information

     // Text fields for user input
     private JTextField name;
     private JTextField street;
     private JTextField city;
     private JTextField statezip;

     // Labels to go with each text field
     private JLabel lname;
     private JLabel lstreet;
     private JLabel lcity;
     private JLabel lstatezip;

     // Check boxes for types of diver training
     private JCheckBox ow;
     private JCheckBox a;
     private JCheckBox res;
     private JCheckBox un;
     private JCheckBox w;

     // Textfields for Emergency box
     private JTextField nname;
     private JTextField phone;
     private JTextField rel;

     // Textfields for Emergency Contact
     private JLabel lnname;
     private JLabel lphone;
     private JLabel lrel;

     // Buttons and image
     private JButton enter;
     private JButton edit;
     private JLabel seahorse;

     // Panels to be built and added
     // to the border layout of this
     // panel.

     private JPanel images;
     private JPanel jaddress;
     private JPanel emerg;
     private JPanel training;

     // Class to handle functionality of checkboxes
     ItemListener handler = new CheckBoxHandler();
```

```
   // Class constructor that builds the necessary
   // labels and text fields images, buttons,
   // and panels.

        public Diver()
          { // Opens Constructor
           // Sets layout for Diver panel
           setLayout(new BorderLayout());
           // Sets background color for
           // Diver panel
           setBackground(Color.white);

           // Initializes textfields
           name = new JTextField("Enter Your Name");
           street = new JTextField();
           city = new JTextField ();
           statezip = new JTextField ();
           // Initializes labels for textfields
           lname = new JLabel("Name: ");
           lstreet = new JLabel("Street: ");
           lcity = new JLabel ("City: ");
           lstatezip = new JLabel("State & Zip Code: ");
           // Initializes checkboxes with titles
           ow = new JCheckBox("Open Water", true);
           a = new JCheckBox("Advanced");
           res = new JCheckBox("Recovery & Rescue");
           un = new JCheckBox("Underwater Photography");
           w = new JCheckBox("Wreck & Cave Diving");

           // Initializes textfields for emergency panel
           nname = new JTextField();
           phone = new JTextField();
           rel = new JTextField ();
           // Initializes labels for textfields
           lnname = new JLabel("Name: ");
           lphone = new JLabel("Phone: ");
           lrel = new JLabel ("Relationship: ");
           // Initialize objects
           enter = new JButton("Enter Diver Data");
           edit = new JButton("Edit Diver Data");
           seahorse = new JLabel("",
                new ImageIcon("images/2seahorses.jpg"),
                                      JLabel.CENTER);
           // Calls method to buid image panel, which
           // is defined outside of the constructor
           buildImagePanel();

           // Calls method to buid address panel, which
           // is defined outside of the constructor
           buildAddressPanel();

           // Calls method to buid emerg panel, which
           // is defined outside of the constructor
           buildEmergencyPanel();

           // Calls method to buid training panel, which
           // is defined outside of the constructor
           buildTrainingPanel();

         //The methods called above build the panels, then this
        // call to add adds each panel to the main panel's
```

```
    // border layout manager.

     add(jaddress, BorderLayout.NORTH);
     add(images, BorderLayout.CENTER);
     add(training, BorderLayout.EAST);
     add(emerg, BorderLayout.SOUTH);


  } // Ends constructor


  // This method creates a panel called images
  private void buildImagePanel()
  { // Opens method
     images = new JPanel();
    // Sets the color, layout, and adds the
    // seahorse object
     images.setBackground(Color.white);
     images.setLayout(new FlowLayout() );
     images.add(seahorse);
    } // Closes method

  private void buildAddressPanel ()
  { // Opens method
    jaddress = new JPanel();
    // Sets color and layout.
    // Adds the textfields and labels for
    // diver input.
    jaddress.setBackground(Color.white);
    jaddress.setLayout( new  GridLayout(2, 4, 20, 20) );
    //Adds each component to the panel
    jaddress.add(lname);
    jaddress.add(name);
    jaddress.add(lstreet);
    jaddress.add(street);
    jaddress.add(lcity);
    jaddress.add(city);
    jaddress.add(lstatezip);
    jaddress.add(statezip);

    // Sets a border around the panel, including
    // a title
    jaddress.setBorder(BorderFactory.createTitledBorder(
                              "Diver Personal Information"));

    //Listeners for each text field in the
    name.addActionListener( this );
    street.addActionListener( this );
    city.addActionListener( this );
    statezip.addActionListener( this );
    nname.addActionListener( this );
    phone.addActionListener( this );
    rel.addActionListener( this );

  } // Closes method

  private void buildEmergencyPanel()
   { // Opens method
    //Create another panel for emergency input fields
      emerg = new JPanel();
      emerg.setLayout( new  GridLayout(2, 4, 20, 0) );
```

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

      emerg.add( lnname);
      emerg.add( nname);
      emerg.add( lphone);
      emerg.add( phone);
      emerg.add( lrel);
      emerg.add( rel);
      emerg.add(enter);
      emerg.add(edit);
     //Creates titled border around emerg panel
      emerg.setBorder(BorderFactory.createTitledBorder("Emergency"));

      //Adds an actionListener for Emergency Info
      nname.addActionListener( this );
      phone.addActionListener( this );
      rel.addActionListener( this );


      //Creates borders around the Edit and
      // Enter Diver Data buttons
      enter.setBorder(BorderFactory.createRaisedBevelBorder());
      edit.setBorder(BorderFactory.createRaisedBevelBorder());

       //Listeners for the buttons
       enter.addActionListener( this );
       edit.addActionListener( this );
      } //Closes method

  private void buildTrainingPanel()
  { //Opens method

    // Creates a panel for training courses

     training = new JPanel();
     training.setBackground(Color.white);
     training.setLayout(new GridLayout(5, 1, 10, 20 ) );
     //sets backgrounds of components to white
     training.add(ow).setBackground(Color.white);
     training.add(a).setBackground(Color.white);
     training.add(un).setBackground(Color.white);
     training.add(res).setBackground(Color.white);
     training.add(w).setBackground(Color.white);

    //Sets a titled border around training panel
    training.setBorder(BorderFactory.createTitledBorder("Training"));
    //Adds listeners to checkbox items
      ow.addItemListener(handler);
      a.addItemListener(handler);
      un.addItemListener(handler);
      res.addItemListener(handler);
      w.addItemListener(handler);
   } //Closes method

// Prints the input into JLabels and hides text fields, or
// returns the text fields so input may be changed

public void actionPerformed(ActionEvent evt)
  { // Opens method
  // Checks if the button clicked was the
  // Enter Diver Data button.
```

```
      // If not, moves on to next if statment.
      // Otherwise it enters this if statement
      if ((evt.getSource() == name) || (evt.getSource() == enter))
           { // Opens if statement
            //Retrives the text from the name text field and
            //assigns it to the variable nameText of
            //type String
            String nameText = name.getText();
            lname.setText("Name:    " + nameText);
            //After printing text to JLabel, hides the textfield
            name.setVisible(false);
            }// ends if statement
      if ((evt.getSource() == street) || (evt.getSource() == enter))
           { // Opens if statement
            String streetText = street.getText();
            lstreet.setText("Street:    " + streetText);
            street.setVisible(false);
             }// ends if statement
       if ((evt.getSource() == city) || (evt.getSource() == enter))
           { // Opens if statement
            String cityText = city.getText();
            lcity.setText("City:  " + cityText);
            city.setVisible(false);
                 }// ends if statement
       if ((evt.getSource() == statezip) || (evt.getSource() == enter))
           { // Opens if statement
            String statezipText = statezip.getText();
            lstatezip.setText("State & Zip:     " + statezipText);
            statezip.setVisible(false);
                 }// ends if statement
      if ((evt.getSource() == nname) || (evt.getSource() == enter))
           { // Opens if statement
            String relname = nname.getText();
            lnname.setText("Name:   " + relname);
            nname.setVisible(false);
                  }// ends if statement
      if ((evt.getSource() == phone) || (evt.getSource() == enter))
           { // Opens if statement
            String relphone = phone.getText();
            lphone.setText("Phone:   " + relphone);
            lphone.setForeground(Color.red);
            phone.setVisible(false);
            }
      if ((evt.getSource() == rel) || (evt.getSource() == enter))
      {
            String relString = rel.getText();
            lrel.setText("Relationship:   " + relString);
            rel.setVisible(false);


        // If Edit button was clicked, set textfields to
        // visible for user to reenter information and
        // it sets the text that had been entered to
        // to an empty String, giving the appearance
        // that the text has been removed.
        } // Closes if statement
         if (evt.getSource() == edit)
            { // Opens else if statement

         // If edit button has been pressed
         // the following is set to visible
```

```
            name.setVisible(true);
            street.setVisible(true);
            city.setVisible(true);
            statezip.setVisible(true);

            // Relative's info

            nname.setVisible(true);
            phone.setVisible(true);
            rel.setVisible(true);

            lnname.setText("Name:   ");
            lnname.setForeground(Color.black);
            lphone.setText("Phone:   ");
            lphone.setForeground(Color.black);
            lrel.setText("Relationship:   ");
            lrel.setForeground(Color.black);

         } // Ends if statement

       } // Ends actionPerformed method

// Inner class that handles functionality for the
// checkboxes.
private class CheckBoxHandler implements ItemListener
{ // Opens inner class

     public void itemStateChanged (ItemEvent e)
       { // Opens method
           if ( e.getSource() == ow )
              if ( e.getStateChange() == ItemEvent.SELECTED )
                ow.setForeground(Color.blue);
                else
                ow.setForeground(Color.black);

           if ( e.getSource() == a )
              if ( e.getStateChange() == ItemEvent.SELECTED )
                a.setForeground(Color.blue);
                else
                a.setForeground(Color.black);

            if ( e.getSource() == un )
              if ( e.getStateChange() == ItemEvent.SELECTED )
                un.setForeground(Color.blue);
                else
                un.setForeground(Color.black);

             if ( e.getSource() == res )
              if ( e.getStateChange() == ItemEvent.SELECTED )
                res.setForeground(Color.blue);
                else
                res.setForeground(Color.black);

             if ( e.getSource() == w )
              if ( e.getStateChange() == ItemEvent.SELECTED )
                w.setForeground(Color.blue);
                else
                w.setForeground(Color.black);

            } // Closes ItemStateChanged method
    } // Closes class CheckBoxHandler
```

```
} // Closes Class Diver
```