Copyright (c) 2001 Sun Microsystems, Inc. All Rights Reserved.

Redistribution and use in source and binary forms, with or without modification,

are permitted provided that the following conditions are met:

-Redistribution of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

-Redistribution in binary form must reproduce the above copyright notice,

this list of conditions and the following disclaimer in the documentation and/or

other materials provided with the distribution.

Neither the name of Sun Microsystems, Inc. or the names of contributors may

be used to endorse or promote products derived from this software without

specific prior written permission.

This software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN MIDROSYSTEMS, INC. ("SUN") AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE THIS SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You acknowledge that this software is not designed, licensed or intended for use in the design, construction, operation or maintenance of any nuclear facility.

8/17/01

Building an Application: Introduction

by Dana Nourie, *October 2001*

Overview

What You Need

Start the Tutorial

Overview

Many Java™ Application Programming Interfaces (APIs) are used in application programming. As you learn how to create buttons or write text to files, figuring out how to fit the technologies together into a single application can be difficult and confusing. Seeing an application built from the ground up can be more useful than just reading how to make a menu bar or read a file into a Graphical User Interface (GUI).

This six-part tutorial details application development, using a single fully-featured application as an example. You'll learn how to:

- Use predefined objects and creating new objects.
- Call predefined methods and writing new methods.
- Create GUI components, such as tabbed panes, buttons, menus, text fields, and text areas.
- Use AWT layout managers and event handlers.
- Print text and images to the screen.
- Read from and writing to files.
- Open an HTML page in the application and activating the links.

You'll also learn about packages and packaging an application for distribution, and the Java technologies referred to as Java I/O, Project Swing, and AWT.

The application created for this series tutorial is a simple dive log, such as one a scuba diver might use to record dive depth, water temperature and conditions, and air consumed.

Though to a certified scuba diver, this application falls short of being a robust, detailed dive log, it serves as an example application, demonstrating features commonly used in application programming.

The tutorial begins with simple concepts and leads to more complex programming techniques, introducing new concepts and repeating programming techniques throughout.

- The first half of each series part explains how to create the GUI for that particular pane.
- The second half details the functionality of any buttons, menus, text fields, and so forth.

This tutorial is aimed at beginning programmers, or developers new to Java technology. Though programming concepts are introduced and many given great detail, this is not intended to be a comprehensive tutorial to all Java programming syntax.

Click to enlarge

What You Need

You don't need to be a certified scuba diver to follow this tutorial (though diving is a lot of fun and you might consider trying it). You also don't need a lot of experience with the Java programming language.

Familiarity with programming is helpful, and it's recommended that you at least understand what the Java platform is and how to set it up on your computer.

If Java technology is new to you, read the following articles before starting the tutorial:

About Java Technology
Introducing the Java Platform
Setting Up and Getting Started

The following software is required to compile the code in this tutorial:

- Java 2 Platform, Standard Edition (J2SE™)
  This dive log was created using J2SE version 1.3. J2SE includes the compiler and API needed for creating the Java technology dive log.
- A text editor for writing classes. Use a simple text editor. If you use a full-fledged word processor, save class files as text and create a `.java` extension. Examples are shown in the tutorial.

  Optional
  Forte™ for Java
  This is an Integrated Development Environment (IDE) that enables you to write, compile, and run code within one application. In addition, it has short cut features to use for creating buttons, toolbars, and many other widgets or components. Learning an IDE can be time-consuming, so many programmers opt for a simple text editor and use the command line for compiling.

Start the Tutorial

If a dive log is not to your liking, feel free to change the text and code to better suit your needs. Perhaps you'd rather design a diet or exercise log, or some other kind of application. You can learn a lot by taking prewritten code and changing it. In fact, that is the theme of object oriented programming: Reuse, don't start from scratch.

As each part of the tutorial is completed, the titles below will be linked. With that in mind, start the tutorial:

Part 1: Application Objects, Classes, Constructors, and Methods PDF *(October 2001)*
Part 2: Inheritance, Images, Text, and Layouts *(December 2001)*
Part 3: Receiving User Input Through Text Fields and Check Boxes, and the Basics Event Handling
Part 4: Scroll Bars, Pop-up Option Boxes, Reading From and Writing to Files
Part 5: Converting Data, Operations and Expressions, Displaying Results
Part 6: Displaying HTML and Activating Links

Application Objects

The Java™ programming language has a lot in common with every day life. Each day you use objects, such as the car you drive, the meals you eat, and Internet pages you read.

Java applications are also built of objects, such as buttons, scroll bars, menus, and text areas.

In addition, objects do something. Your car gets you from one place to another. To do that, it has to have function, or many functions. Software objects also have function.

For action to take place in Java objects, blocks of code called methods are used. Methods tell an application what to do when buttons are clicked, menus are opened, and text is typed. Methods are either predefined or are created from scratch to manipulate objects, or the data within those objects.

Setting up the Java 2 Platform

[Java 2 Platform, Standard Edition version 1.3.1]

[Setting Up and Getting Started]

[Lesson: Solving Common Compiler and Interpreter Problems]

The Dive Log application you're going to create consists of many objects. These objects are based on classes that detail how the object is defined and how it behaves.

This lesson covers the basics of using predefined classes and creating, or designing, new objects with your own classes. In addition, you'll learn how to work with objects by calling predefined methods and writing new methods to get the application to do what you want it to do.

Preparation

Before learning about the code for the Dive Log, you need to create a special directory for the files and images that build this application.

Saving Files

The Dive Log is designed from many classes, therefore many files. Keep them organized and in place.

Follow these steps...

1. Create a directory called `divelog`.
2. Under the `divelog` directory, create another directory called `images`.
   The directory stucture:
   For the Windows platform:

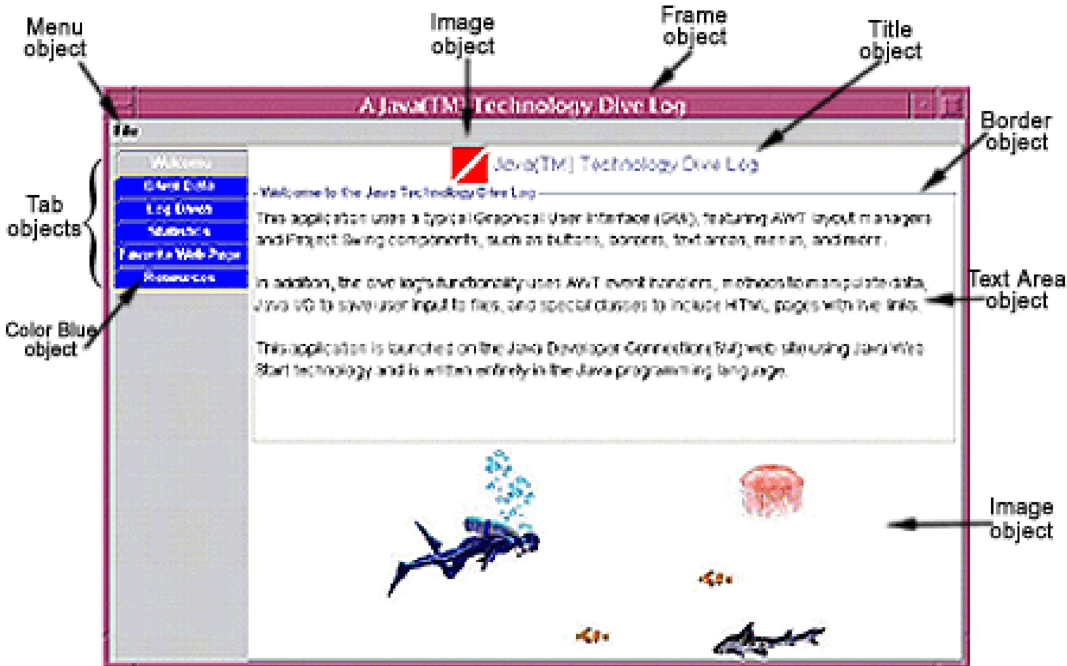
   ```
   C:\divelog\images
   ```

   For the Solaris™ environment:

   ```
   %home/usr/divelog/images
   ```

Dive Log Objects

The concept of software objects is easier to understand with a real world example. Everything you're going to create for the Dive Log application is an object, including the application itself. The Dive Log application main screen lists some of the visible objects.



Visible objects in the completed Dive Log

Each object has its own characteristics, or state. The Title object is different from the Tab objects, and the Image objects are different from the Text Area objects. But before an object comes into existence, the design must be written.

Before cars and houses are built, someone designs a blueprint. Before a batch of cookies is baked, a recipe is written. Software objects are also created with a specific design.

The design for a software object is called a class. Classes detail, or specify, exactly how an object should appear and how it is to behave. Instructions for creating software objects are carefully written into the class using variables for data and information, and methods for manipulation of that data and information.

Do you think there is a limit to how many objects make up an application?
Yes   No

Do you think there is a limit to how many objects make up an application?

There are no limits on how many objects you can create for an application.

The number of objects an application instantiates, or puts into memory, depends entirely on the requirements of the application.

The point to keep in mind about classes and objects is that the class is the plan, the object is the plan with all the details filled in and put into memory. In other words, the object is the actual button a user clicks rather than the

instructions or class that specifies button size or function.

About Classes

The classes that are a part of the Java™ J2SE™ download are complete, predefined classes you can use in your applications. These predefined classes provide features frequently used in creating applications, such as writing to and reading from files, creating graphical components like buttons and menus, and making web pages interactive.

To create an application, though, you need to define your own classes as well as using predefined classes from the Java library.

More About Objects

[Classes, Objects, and Constructors: What's the Difference?]

[Object-Oriented Programming Defined]

[What Is an Object?]

When you define a class, you are planning how the object created from that class is going to appear and behave. A class contains:

- Fields
  Fields, or variables, store data and are frequently called data members. These variables often differentiate one object from another and define attributes such as amounts, names, titles, and so forth.
- Methods
  Methods manipulate variables or objects, such as doing math operations, inputting characters or strings, printing text to the screen, adding a button to a menu bar, or simply instantiating an object.

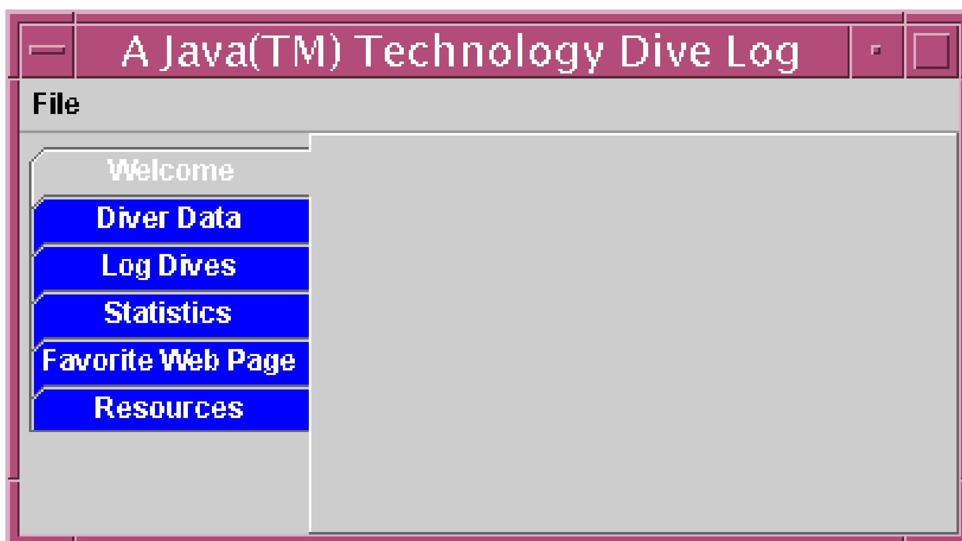
What are Keywords?

Keywords are reserved words that cannot be used as variable, method, or class names because they are used in Java™ programming syntax.

`for`, `while`, `public`, `if`, and `class` are some keywords.

More on fields and methods later. For now, look at some kinds of classes you'll write to create objects for the Dive Log.

You'll need a frame for the application. Other components of the application are organized into tabbed panes. Each of those tab objects has white text and the background color blue. Later, you'll learn what goes on each pane and how to develop those objects. For now, you'll learn about frame and tab objects.



Result of creating frame and tab objects

Designing the classes for these objects makes more sense as code is covered in detail.

You will start with designing a few objects (the frame and tabs), then build on those classes as you progress through the lessons.

1. Take a look at the first class you are going to create: DiveLog.java.
2. Note the syntax of how the names are written, paying special attention to which words start with uppercase letters and which do not.
3. Look at positions where the curly braces are placed.
4. Open your text editor to start your first class.
5. Copy and paste this line of code into your text editor:

```
package divelog;
```

6. Save the file, naming it `DiveLog.java`

Every class in the Dive Log starts with:

```
package divelog;
```

What is the purpose of the keyword `package`?
A. Group and store related classes in a container. B. Make the classes in a package accessible to the compiler. C. All of the above.

The purpose of using the `package` keyword is to group **and** store related classes in a container, and to make the classes in a package accessible to the compiler.

More About Objects

Naming Conventions

Code Conventions for the Java Programming Language

It's All in the Packaging

Creating a Package

Packages

The Java™ programming language has many predefined classes that can be used in applications. There are so many that the classes have been organized into groups called packages.

For instance, classes that support I/O (input and output) are contained within the `java.io` package, and classes for creating applets are in the `java.applet` package. Putting classes into packages organizes them conveniently for the compiler and for you.

When you create applications, put the classes for your particular application into a package. The Dive Log application stores its files in a package called `divelog`. This tells anyone looking at the code that the classes for the Dive Log application are in a directory called `divelog`, just as `java.applet` classes are in a directory called `applet`, which is in a directory under `java`. In other words, package names correspond with directory names.

package `java.applet` has the directory structure:

```
/java/

   |

    /applet
```

This `divelog` package has the directory structure:

```
C:\divelog
```

or

```
%/usr/home/divelog
```

Naming Your Class Files

As you create class files, save them in the `divelog` directory:

1. Copy or type the code as explained in the tutorial into a file.
2. Save the file with the same name as the class you are creating.
3. Name each file with the `.java` extension. A file with the class `DiveLog` defined should be called `DiveLog.java`.

When you use predefined Java API classes, the compiler needs to locate those classes. The `package` keyword is only going to map out classes for your particular application, not the Java API classes you may be using. Tell the compiler which and where the Java classes are in two ways:

- Writing out the entire package name and the class name: `java.applet.Applet`
- Using an import statement at the top of your class: `import java.applet.*;`

The second way, using import statements, tells the compiler your class includes classes from the `java.applet` package. This way, after the import statement, you need only name the class rather than having to write out the fully qualified name as in the first example.

Getting Ready for the Graphical User Interface (GUI)

The Dive Log is an application with a graphical user interface (GUI). In fact, most of the application is the interface itself. In creating graphical features such as menus, tabs, text, images, and so forth, you will become familiar with Project Swing, the Java APIs that provide many predefined classes for GUI applications. In addition, you will learn about AWT, another large set of Java APIs that provides classes for designing the layout of the graphical objects, and functionality called event handling.

Importing Packages

To make it easy to use the Swing and AWT classes, include import statements in your class.

1. Copy and paste, or type the following lines of code into your `DiveLog.java` class file:

```
import javax.swing.*;

import java.awt.*;

import java.awt.event.*;
```

2. Save the file

These import statements allow easy access of the predefined classes in those packages. In this case, the packages with classes for GUI components have been imported, so now you needn't type out the fully qualified name, but instead can directly name the class you need to use. In other words, import statements give you a shortcut to class names in other packages. You'll see examples of this throughout the tutorial.

Class Definition

After package names and import statements, you are ready to define the class. The first class to define is the framework for the application itself, the `DiveLog` class:

```
public class DiveLog

{
```

This line tells the compiler:

- The class is `public`, meaning it's accessible from any other class in any package.
- That it is a class, hence the keyword `class`.
- That the class' name is `DiveLog`.

Notice this class name begins with an uppercase letter. Naming convention calls for the first letter of a class name to be in uppercase, and the first letter of method names to be in lowercase. Using this naming scheme makes it easier to read code because class names differ from method and variable names.

The { begins the class definition. At the end of the class you must have a closing }. Curly braces signify the beginning and ending of classes and method bodies and statements.

Missing or extra curly braces are one of the most common causes of compilation errors. It's a good idea to create curly braces in pairs, then insert your code between them, such as in this method:

Commenting Code

Commenting code is important to provide insight for others, and also to provide reminders for you. Comments may be inserted into source in the following formats, which tell the compiler to ignore the text:

```
// This is a single line comment.

/* This format is often used for

multiline comments. */
```

First:

```
public void actionPerformed(ActionEvent e)

 {



 }
```

Second:

```
public void actionPerformed(ActionEvent e)

 {

     System.exit(0);

 }
```

Follow these steps...

1. Open the `DiveLog.java` file in your text editor.
2. Type or copy and paste the class header and curly braces into your file after a few lines down from the import statements:

```
public class DiveLog

{
```

}

3. Save the file.

Declaring Variables

Objects contain data, and variables are holders of that data, or rather references to the data. The idea behind using a variable, a reference to the data, rather than the data itself is so you can manipulate the data.

For instance, in the example below `birthYear` and `currentYear` are given values, while `age` represents `birthYear` subtracted from `currentYear`:

```
birthYear = 1981;

currentYear = 2001;

age = currentYear - birthYear;
```

In this example, the variable `age` has a value of `20`. Change the value of either `birthYear` or `currentYear` and the value of `age` changes accordingly.

Variables don't have to be hardcoded. The variable `birthYear` can instead be assigned to a text field for a user to enter a number. Then the `currentYear` variable could be hardcoded, declared with the current year, such as `2001`. This kind of variable use is what makes applications dynamic. While one user types in `1962` as the value of the `birthYear`, another user types in `1982` as the `birthYear`, a different value for the `age` is calculated according to the user input.

Using variables in this way to represent data, a programmer doesn't have to know what the value is going to be in advance. Bank applications are likely to have variables representing customers names, addresses, account numbers, checking account amounts, and so forth, along with hardcoded variable values such as interest rates and bank fees.

In the Java programming language, variable **types** must be declared with the variable name. In other words, the compiler has to know if the variable is going to represent a `String`, an `int`, or some other type. For the example above to compile, the code must be rewritten:

```
int birthYear = 1981;

int currentYear = 2001;

int age = currentYear - birthYear;
```

Look closely at the code above, then see if you can answer the question below:

Which variable declaration is written incorrectly?

```
A. String lang = "Java Programming";
B. double amount = 80.11
C. int size = 8;
```

Which variable declaration is written incorrectly?

The statement `double amount = 80.11` is incorrect because it is missing a semicolon at the end of the declaration. All statements, including declarations, must end in a semicolon. The statement should appear as:

```
double amount = 80.11;
```

More about Variables and Java Documentation

Variable Definition and Assignment

Variables

Java 2 Platform Documentation

Variable Types

The Java™ programming language is sometimes called a strongly typed language because you must declare variables as a specific type before using them. A variable can be one of two basic types:

- Primitive type
- Reference type

Primitive Type

Primitive, or simple, types are the only types that are not objects.

The following are primitives:

- Integers, such as `byte, short, int`, and `long`
- Floating-point numbers, such as `double` and `float`
- Characters representing letters and numbers called `char`
- Boolean, which is a type that represents `true` or `false` values

In other words, primitives are what you commonly call numbers, single characters, or true or false. Numbers can also be represented as a `String`, but if you declare a number as a `String`, you cannot do mathematic operations on it, and it becomes an object of type `String`.

Also, whenever numbers are entered at the command line or in a GUI, the entry is accepted as a `String` object and must be converted to do calculations. Special classes called wrappers make this conversion and are covered later in this tutorial.

Because primitive types are not objects, declaring them is a one-step process:

```
int month;  //declaration without a value

int month = 10; //declaration with an assigned value of 10
```

Reference Type

Reference types frequently refer to predefined objects, such as classes that are a part of the J2SE library. Reference types may also refer to classes specifically designed by the programmer to go with an application. The term *class type* is often used synonymously with *reference type*.

As an example:

```
Font monoFont = new Font("Courier", Font.PLAIN, 12);
```

What are Parameters?

Parameters appear between ( ) following a method or class name, specifying the type of value that can be passed in:

```
methodName(type variableName, type variableName2)
```

The parameter is the information you pass to a new class to build an object, or to a method to use.

The reference variable name `monoFont` refers to the `Font` class, a predefined class in the Java library. To declare and instantiate, or create a `Font` object, the keyword `new` is used. In this case, information is also included, such as the font face, style type, and point size. The included information is called parameters. In other words, a `Font` object with specific details is created and put into memory.

Many of the predefined classes provided in the Java library are created this way. If you want to instantiate a predefined class in your application, use the Java API documentation to find out exactly how to call the class from within your application. The documentation makes more sense when you understand constructors and methods, which is discussed in detail later in this tutorial.

In `DiveLog.java`, two variables are declared right after the opening curly brace.

Follow these steps...

1. Open the `DiveLog.java` file in your text editor.
2. After the opening curly brace, type or copy and paste these two declarations:

```
private JFrame dlframe; //Not assigned yet.

private JTabbedPane tabbedPane; //Not assigned yet.
```

3. Save the file

`JFrame` and `JTabbedPane` are reference types. In this case, the variables `dlframe` and `tabbedPane` refer to classes in the `javax.swing` package. In the declarations above, no object has been created yet. For now, this code just tells the compiler to reserve some memory for these two variables. Later in the code, objects of the `JFrame` and `JTabbedPane` classes are instantiated, or created, to build a frame for the Dive Log application, with a tabbed pane included.

Access Attributes

In the snippet of code above, notice the words `private`. This is an access attribute that tells the compiler if other packages or classes have access to this variable. In this case, the variables `dlframe` and `tabbedPane` have been declared `private`, which makes them accessible only from within this class and not accessible at all from other classes or packages.

Access attributes are also used with classes, as in this `DiveLog` class, which has been declared `public`, making it accessible to any class or package, and access attributes are used with methods.

The following table lists the access attributes and the access privileges they allow:

Attribute	Access
If none is provided	Default or package, meaning from any class in the same package
`public`	Any class or package
`private`	Only from within that particular class
`protected`	Any class in that package only

You'll see more about variables and access attributes as you create classes for the Dive Log. The important point to keep in mind is that the variables declared so far can be accessed only from the `DiveLog.java` class, and as yet they have not been assigned a value. They've only been declared as a reference type, specific to the `JFrame` and `JTabbedPane` classes. Until the key word `new` is used to instantiate them, these are not yet objects in memory.

Instantiating the `JFrame` and `JTabbedPane` objects occurs within a method. As mentioned earlier, methods make something happen. Methods instantiate objects, perform math calculations, and more.

Constructing Objects

One important type of method to understand is a special method called a constructor. The main purpose of a constructor is to set the initial state of an object when the object is created, or instantiated. A class details the data an object contains and can work with, methods it uses to work with that data, and a constructor tells how the object is to be built.

Constructors look like other methods with a few differences, which are described in the next section.

Use the `DiveLog` constructor as a guide to answer the following question:

```
public DiveLog()

{

. . . // These dots mean that code has been

      // omitted for brevity.

}
```

Which constructor is written incorrectly?
A. `public Button(){}` B. `public Button(String mode){}` C. A and B are both correct.

Which constructor is written incorrectly?

Both `public Button(){}` and `public Button(String mode){}` are correct. Constructors can be written with empty `()`, or a constructor can require parameters in the `()`, such as B, a `Button` constructor that takes a `String` parameter. In fact, a class can have several constructors. This is called constructor overloading.

More about Constructors

Classes, Objects, and Constructors: What's the Difference?

Providing Constructors for Your Classes

Constructor Details

Many classes in the Java™ library have a constructor. To create an object of that class in one of your own classes, for most classes, you call the class's constructor.

For instance, if you want to create a button for your application, the Java library has a predefined `Button` class you can use. A look at the documentation for `Button` shows that this class has two possible constructors you can call:

- `Button()`
  Creates a button without a label
- `Button(String label)`
  Creates a button with a text label

Having more than one constructor is called constructor overloading. Each constructor must have the same name as the class that defines it, but it can have as many constructors as might be necessary. The difference is in the parameter list that is defined in the `()`. The compiler knows which constructor is being called simply by the type of information being provided.

In this case, if you provide a `String` as a parameter when calling the constructor as in `Button("Edit")`, the compiler knows you are calling the second constructor.

A constructor can require different types in the parameter list, but you should put them in the order you want them added.

For instance:

```
Customer(String name, String address, int age)
```

Instantiate a `Customer` object by calling the `Customer` class constructor with the `new` keyword:

```
Customer cust1 = new Customer(

         "Jane Smith", "555 Street", 30);
```

Customer defines the reference variable, and the variable is called cust1. A Customer object is built by calling the Customer class constructor with the keyword new.

In other words, the constructor builds an object exactly the way you want. In this case, it creates a Customer object with personal information.

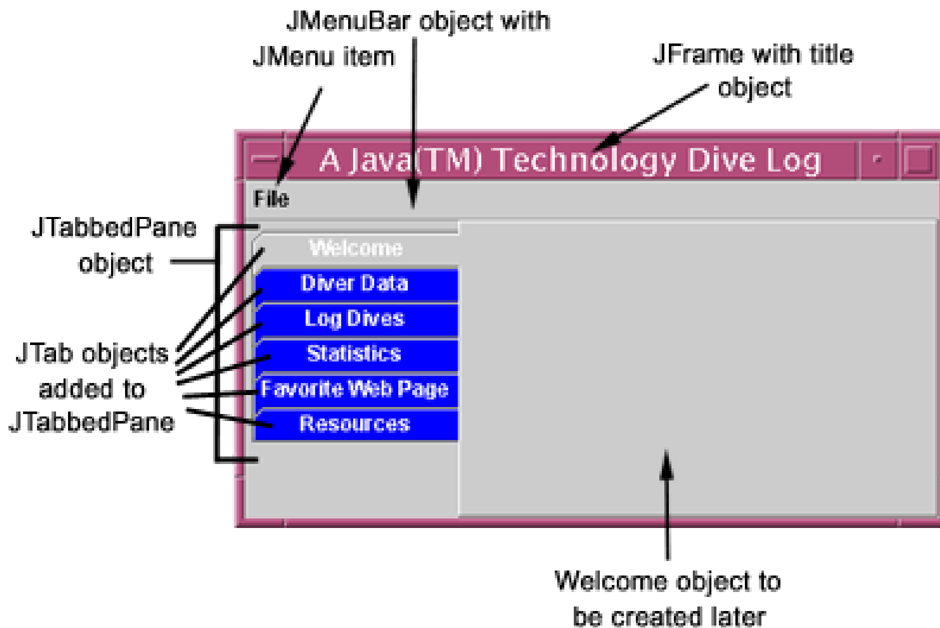
Upon starting an application, certain GUI components must appear for the user to use the application. This concept may make it seem like everything for the entire application must be written into the constructor, but it doesn't.

The DiveLog constructor defines just a few GUI components, and it can have some of those class definitions call other classes. Otherwise, the main class would get big and unwieldy. Also, in object oriented programming, the objective is to build objects that work with other objects to make it easy to add or remove components without having to change a lot of code.

When writing constructors, include only the elements an object must have on initialization.

You'll see these concepts at work as you build this Dive Log application.

The image below points out the objects that are created within the DiveLog constructor:



Object created from the DiveLog constructor

The DiveLog class constructor begins as follows:

```
public DiveLog()

  { // Opens constructor



   // Creates a frame object to add to

   // the application GUI components.
```

```
   dlframe = new JFrame(

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

Notice this class constructor has the same name as the class itself and does not require any parameters, which `()` shows. The opening { begins the body of the constructor.

Recall at the beginning of `DiveLog.java`, the variable `dlframe` was declared, but not yet assigned a value. Now, within the constructor, the `JFrame` constructor is assigned to the variable `dlframe`. A `JFrame` is simply a predefined Java class that builds a frame with a border, a minimize and maximize button, and a close box. A `JFrame` is a Swing container that can hold other containers and components. `DiveLog` initializes a `JFrame` as follows:

```
   dlframe = new JFrame(

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

If you read the `JFrame` class documentation under the heading Constructor Summary, you'll discover several constructors available for building a `JFrame` object:

- The first is the default constructor, which creates a `JFrame` that's invisible and doesn't have any other features.
- The second constructor creates a `JFrame` in the specified `GraphicsConfiguration` of a screen device and a blank title.
- The third constructor accepts a `String` as an argument, and creates a `JFrame` with the title provided. This is the constructor called in the `DiveLog` constructor.
- The last constructor creates a `JFrame` with the specified title and the specified `GraphicsConfiguration` of a screen device.

Though a `JFrame` object is initialized with the title for the frame and assigned to the variable `dlframe`, it still needs a bit more work.

The following snippet of code has some advanced concepts involved, so it will be explained fully at a later time. This method insures that the Dive Log application can shut down fully and properly.

```
// Closes from title bar

// and from menu

dlframe.addWindowListener(new WindowAdapter()

  {

    public void windowClosing(WindowEvent e)

    {

    System.exit(0);
```

```
    }
});
```

1. Open the `DiveLog.java` class file.
2. Type, or cut and paste the following lines of code:

```
public DiveLog()
      {

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

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

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

```
        }// Ends constructor
```

3. Save the file.

Next, the `JTabbedPane` is instantiated. At the top of the `DiveLog` class, the variable `tabbedPane` was declared as type `JTabbedPane`, but not assigned. Here, in the constructor, the keyword `new` is used to create the `JTabbedPane` object, passing in the necessary parameters to position the tabs on the left side of the window pane, and the variable is assigned to the `JTabbedPane` object:

```
// Tabbed pane with panels for components



tabbedPane =

   new JTabbedPane(SwingConstants.LEFT);
```

`JTabbedPane` is another handy Java class that creates tabs in an application and has two available constructors to call. The `DiveLog` constructor calls the `JTabbedPane` class constructor that makes it possible to create tabs at the `TOP, BOTTOM, LEFT`, or `RIGHT`. Simply type in your preference when calling the constructor. In the `DiveLog` case, the tabs are set to the left of the screen. You can place them wherever you like.

Calling Predefined Methods

`JFrame` inherits the predefined methods from the classes `Frame, Window, Container, Component`, and `Object`. You'll read about inheritance later.

Recall that methods provide action in applications. A class has at least one method, but generally many more. You can see what methods are available for use in the `JFrame` class by reading the `JFrame` class documentation and scanning the methods listed in the inherited classes. Click on the method names to find out what information you need to provide when calling the methods.

Methods, like constructors, often need information passed into them through the parameter list.

Call these predefined methods simply by using the dot operator with the object variable name as follows:

```
tabbedPane.setBackground(Color.blue);

tabbedPane.setForeground(Color.white);
```

The methods `setBackground` and `setForeground` are called and used on the `tabbedPane` object, setting the background and foreground colors. You can use these methods on any GUI component to change background and foreground colors, simply by naming the object through its variable, then passing in the `Color` class as a parameter, along with the color you want to use.

1. Open the DiveLog.java class file
2. Type, or cut and paste the following lines of code into the file just after the last section you added:

```
// Tabbed pane with panels for Jcomponents

// Instantiate JTabbedPane with keyword new

tabbedPane =

   new JTabbedPane(SwingConstants.LEFT);

// Calls method to set color

tabbedPane.setBackground(Color.blue);

tabbedPane.setForeground(Color.white);
```

3. Save the file.

Predefined methods are easy to use. When you decide you want to create an object of a certain class, then look that class up in the documentation and see what methods are available. Create an instance of that class using the `new` operator and assign a variable, then call the method by using the dot operator as demonstrated above. You'll see many more examples of this as you progress through the Dive Log.

At this point, the `JTabbedPane` class constructor has been called and the object instantiated, but the individual tabs have not been added.

How are the individual tabs added to the `JTabbedPane`?
A. Assign a variable and using the keyword `new`. B. Call a special method of the `JTabbedPane` class. C. None of the above.

How are the individual tabs are added to the `JTabbedPane`?

The answer is B. The clue in this question was the word *added*. The `TabbedPane` class has an `addTab` method that takes parameters. By calling the `addTab` method and providing the necessary information, you can add as many tabs to the `TabbedPane` object as you need.

Calling Methods

You could call the `addTab` method within the constructor for each tab you want to add to the `JTabbedPane` object, but that clutters the constructor body. Constructing the tabs contained within a method designed for that purpose is a

cleaner way of doing it.

So far you have begun writing a constructor and calling predefined methods from the Java™ library, using the object variable name and the dot operator.

To call a method of your design from within the constructor, simply type the method's name and any required parameters:

```
populateTabbedPane();
```

More about Methods

Details of a Method Declaration

Passing Information into a Method

The Method Body

Methods

Note the method name

- Begins with a lowercase letter
- Uses uppercase letters
- Gives an indication of its function to make reading the code easier

Since the `populateTabbedPane` method is called in the `DiveLog` constructor, you must define a `populateTabbedPane` method outside the constructor, which is covered in the next section.

Follow these steps...

1. Open `DiveLog.java` in a text editor
2. Type or paste the following line of code:

```
populateTabbedPane();
```

By now your `DiveLog.java` class file should look like this example.

Defining Methods

A method is a group of programming statements that is given a name. The statements can contain other method calls, simple print statements, or complex operations. When a method is called, the flow of control transfers to that method. Each statement in the method body is executed, one at a time, then control returns to the location where the call was made.

A method's header gives the compiler important information:

- access modifier (optional)
  Tells the compiler whether other classes or packages have access to this method
- return type (required)
  If the method returns an object or primitive data, the type must be listed here. If the method doesn't return a value, then it is type `void`.
- method name (required)
  Like classes, methods need a name. Generally methods begin with a lowercase letter.
- parameter list or () (required)

Return Types

Some methods require a return type. For instance, if you create an `addNumbers` method to add numbers, then return the sum, the return type might be an `int`, `double`, or `float`. For example:

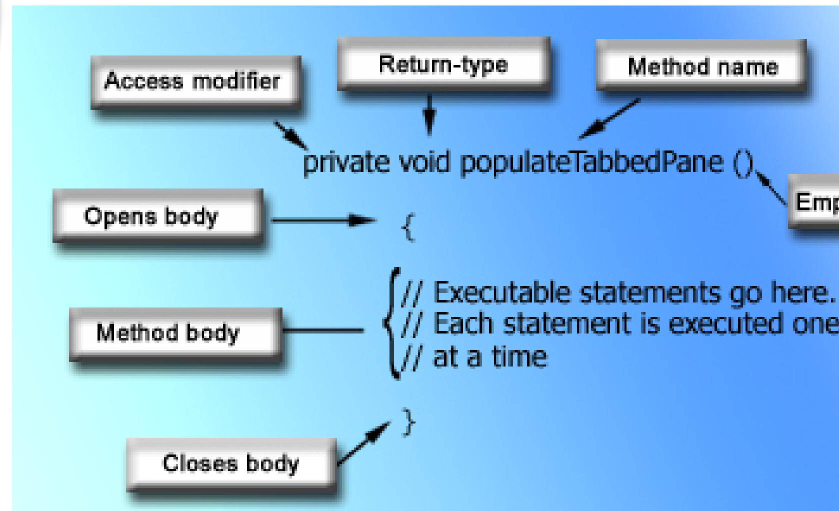
```
public int addNum(int a, int b, int c)
    {
       int sum;
       sum = a + b + c;
       return sum;
     }
```

The return type is `int`.

If no parameters are passed into the method, then empty () are shown. Otherwise the type of information must be detailed in the parameter list.

The `tabbedPane` method is declared `private` because no other class in the `divelog` package, or any other package, needs to or should have access to this method. This method is used only to add the individual tabs to the `JTabbedPane` object called `tabbedPane`.

The return type of the `tabbedPane` method is `void` because it doesn't return a value. This method simply provides a function for building a part of the GUI.



Method header and body

In the `DiveLog` class, the method created to populate the `tabbedPane` object is called `populateTabbedPane` because it is descriptive, making it easy to guess what the method does.

Calling Methods within Methods

As mentioned earlier, tabs are added to the `tabbedPane` object by calling a predefined method from the `JTabbedPane` class. So in this `populateTabbedPane` method, to add tabs, or rather populate the `tabbedPane` object with tabs, you need only call the `addTab` method and provide parameters for each tab.

There are three versions of the `addTab` method:

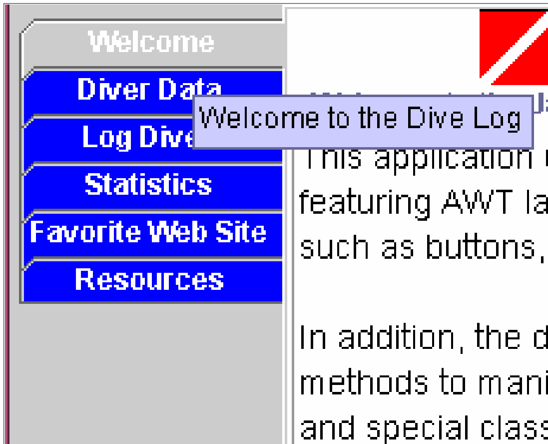
- Tab with title and a component
- Tab with title, icon, and a component
- Tab with title, icon, component, and a string tip

For the DiveLog, use the last:

```
tabbedPane.addTab(
                 "Welcome",
                  null,
```

```
            new Welcome(),

            "Welcome to the Dive Log");
```



Tab with string tip displayed

The last `addTab` method signature creates a tab with a title, a component, and a string tip. Include the title by typing text enclosed in quote marks. Since an icon is not provided on the tabs, `null` takes place of an icon image name.

Next, a component, or more specifically, a new class called `Welcome` is instantiated. The `Welcome` class displays the content for this tab, and is covered fully in the next part of this Dive Log tutorial. Lastly, text for a string tip is provided. This text appears when a user mouses over the tab.

Each tab initializes a new class that contains the code for that particular tabbed page:

1. Open `DiveLog.java` in a text editor
2. Type or paste the following line of code after the `}//Ends Constructor`:

```
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
```

3. Save the file.

Tabs that initialize other classes provide organization for the application and prevent classes from becoming overly long and confusing.

What happens if you do not create classes called `Welcome.java, Diver.java, Dives.java`, and so forth?
A. Nothing since there isn't a class to put into memory yet. B. A compilation error because the compiler can't find the named classes. C. None of the above.

What will happen if you do not create classes yet called Welcome, Diver, Dives, and so forth?

You get a compilation error if the compiler cannot locate every class that is called in the code. To prevent this type of error, create classes that can be instantiated. For now, they don't need to do anything other than exist in memory.

Creating Placeholder Classes

The individual classes initialized with each tab are covered in the next installments of the Dive Log tutorial. The `Welcome` class is the next class to be covered in detail.

In the meantime, though, so you can compile the `DiveLog` class, create empty classes that do nothing but initialize placeholders. For now, these classes consist only of package and import statements, the class header, and the closing and opening curly braces. Each class is saved in its own file.

More on Creating Objects

Creating Objects

Learn How To Store Data in Objects

Beginning Java Objects

Creating Class Instances

1. Open your text editor.
2. Copy and paste, or type the following class, in a new file:

```
package divelog;

/**

 * This class creates the content on the

 * Welcome tabbed pane in the Dive Log

 * application.

 * @version 1.0

*/

//import for buttons, labels, and images

import javax.swing.*;

//import for layout manager

import java.awt.*;


public class Welcome extends JPanel

{ //Opens class




  }//Closes class
```

3. Save this file as `Welcome.java` in the `divelog` directory.
4. Create classes like this `Welcome` class with the titles `Diver.java`, `Dives.java`, `Statistics.java`, `WebSite.java`, and `Resources.java`.

As you work through the tutorial, you'll be instructed to open these files later and fill them in as each class is covered.

Creating Additional GUI Objects

After the DiveLog constructor calls populateTabbedPane, program flow returns to the constructor. Next, a menu object needs to be created. Like the tabs, creating and pulling the objects together to create a menu is best done within a method. Using the typical naming conventions, this method is called buildMenu.

This method is private to prevent other packages and classes from accessing it, and it is void because its only purpose is to build a menu with a menu item called Exit, and to exit cleanly when Exit is selected. The name suggests what the method does, and the parentheses are empty because no parameters are required.

Step 3 below shows you how to create application menus using the JMenuBar, JMenu, and JMenuItem classes. You instantiate a new JMenuBar and assign it to mb. Next, you instantiate a JMenu object with the variable menu, and pass in the String "File" to name a new menu. Lastly, instantiate a menu item object, including the parameter "Exit" and assigning it to item.

Follow these steps...

1. Open your text editor
2. Open the DiveLog.java file
3. Copy and paste, or type the following code in after the closing constructor curly brace:

```
// Method header

private void buildMenu()

  {

  // Instantiates JMenuBar, JMenu,

  // and JMenuItem.

     JMenuBar mb = new JMenuBar();

     JMenu menu = new JMenu("File");

     JMenuItem item = new JMenuItem("Exit");
```

Note: To make the menu functional, an addActionListener is assigned to the JMenuItem. ActionListener concepts are a bit advanced and are discussed in detail in another part of the Dive Log tutorial.

After the buildMenu is called in the DiveLog constructor, the code completes the construction of the DiveLog object. Next the code needs to get the content pane and adds the tabbedPane object. The frame you have created for the application is a container, and in this case is the top-level container. All Swing containers have another container within, which is the content pane, where you place all child containers and components.

1. Type or cut and paste the code:

```
      //Closes the application from the Exit

      //menu item.

      item.addActionListener(new ActionListener()

           {

             public void actionPerformed(ActionEvent e)

             {

                System.exit(0);

              }



    });// Ends buildMenu method
```

   Pull the menu objects together, using the predefined `add` method, passing in the variable reference.

2. Add the following:

```
//Adds the item to the menu object

menu.add(item);

//Adds the menu object with item

//onto the menu bar

mb.add(menu);

//Sets the menu bar in the frame

dlframe.setJMenuBar(mb);
```

3. Save the file.

Though it may seem as though the objects created so far have been added to the frame object, they haven't. Instead

objects must be added to a frame's container, the content pane.

Adding Objects to the Frame

The content pane manages the interior of a Swing frame, and components are added to the frame's content pane rather than directly to the frame. To do this, simply send a message to the `dlframe` object with the dot operator, calling the `getContentPane` method and call the `add` method to add the `tabbedPane` object to the content pane. You won't need to add anything else to the content pane, since the tabs instantiate classes, so those are added with the creation of the tabs, which in turn have already been added to the `tabbedPane` object.

More on Project Swing

Fundamentals of JFC/Swing: Part I

Fundamentals of JFC/Swing: Part II

Trail: Creating a GUI with JFC/Swing

Project Swing and Java™ 2D Graphics

The `pack` method causes the Window to be sized to fit the preferred size and layouts of its subcomponents, and the `setSize` method sets a size for the frame. Even with size and color established, you still won't see the application without setting the initialized `JFrame` component to visible. It might seem odd to do this, but frequently you need to hide components. For instance, once a user has entered information to a text area, you hide the text area component, which is no longer needed, then display the text to the screen, or provide a different component.

This `JFrame` for the Dive Log should be visible all the time. This done by calling `setVisible` to `true` with the following:

```
dlframe.setVisible(true);
```

To hide a component, change the `true` to `false`.

Follow these steps...

1. Open the DiveLog.java file in your text editor.
2. Copy and paste, or type this code before the closing } of the constructor:

```
dlframe.getContentPane().add(tabbedPane);

dlframe.pack();

dlframe.setSize(765, 690);

dlframe.setBackground(Color.white);

dlframe.setVisible(true);
```

Your code should look like this.

The `DiveLog` constructor is finished and defines how the `DiveLog` object looks and behaves when it is initialized:

- The frame that has a set size, background color, and is visible.
- The content pane contains a `tabbedPane` object.
- The `tabbedPane` object contains tabs with titles, string tips, and initializes separate classes that contain the details of those pages. (For now those classes are empty.)

With the instructions about how the `DiveLog` object is written out within the constructor, only one method remains:

```
public static void main(String[] args)

    {

        DiveLog dl = new DiveLog();

    }
```

What purpose does a `main` method serve?
A. Serves as an entry point for any application. B. Calls the other methods required to run the application. C. All of the above.

What purpose does a `main` method serve?

`DiveLog` is the main class that launches the application because this class contains the `main` method. The `main` method is the entry point for any application and calls constructors necessary to application initialization.

When you run the interpreter with the `java` command, the interpreter searches through the class entered at the command line. When `main` is found, it calls the methods of that class and the other classes to run the application.

In this case, the `DiveLog` constructor is called and runs as instructed. The constructor builds the frame, adds the tab and menu objects to the content pane, and initializes the empty classes you created.

Reviewing Concepts

[Runtime Environments and Class Path Settings]

[Comment Syntax]

[Compiling the Program]

[API Documentation]

Follow these steps...

1. Open `DiveLog.java` in your text editor.
2. Add the main method as shown below:

```
public static void main(String[] args)
  {
      DiveLog dl = new DiveLog();
  }
```

You should have seven classes in the `divelog` directory, including the completed `DiveLog.java` class.

Compiling Code and Running Application

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

Follow these steps...

1. Compile `DiveLog.java` as follows:

   On a Windows platform:

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

   In the Solaris™ operating environment:

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

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

2. Run the Dive Log with the following:

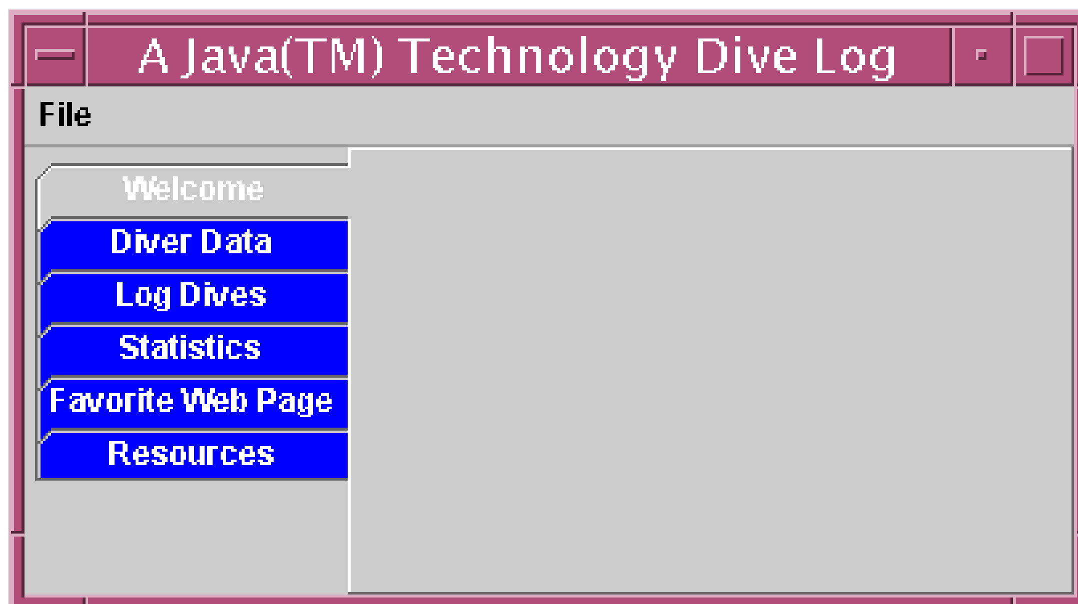
   Windows:

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

   Solaris:

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

The application should look similar to the image below:



Result of completed classes

When you mouse over each tab in your application, you get a string tip. In addition, you get a blank page beside the tabs. Each of these pages are filled as you work through the tutorial and develop the classes that, for now, remain empty.

Summary

Part 1 of the Dive Log provided an overview of many Java programming concepts:

Objects and Classes

- Classes define fields, methods, and constructors.
- Instantiate some objects by calling the constructor with the keyword `new`.
- Variable types are either reference types, or primitive types.

Creating and Calling Methods

- Define methods with an access attribute, a return type, and a list of parameters or empty parentheses.
- Call designed methods by name within the constructor.
- Call predefined methods using the reference variable and the dot operator.
- The `main` method is the entry point for an application.

Files and Packages

- Save application files in a package that relates to where the files are organized.
- Import the necessary Java API to prevent having to type fully qualified class names.

The `DiveLog` class served as an introductory example to Java programming as well as the main class to the Dive Log application. It is not a comprehensive guide to the Java programming language, but instead is an example of a Java application that teaches basic 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 2: Inheritance, Images, Text, and Layouts next.

About the Author

Dana Nourie is a JDC staff writer. She enjoys exploring the Java platform, especially creating interactive web applications using servlets and JavaServer Pages technologies, such as the JDC Quizzes and Learning Paths and Step-by-Step pages in the New to Java Programming Center. She is also a certified scuba diver and is looking for the Pacific Cold Water Seahorse.

Troubleshooting Guide

This page describes the most common problems for compiling and running packaged applications, and the solutions to these problems.

Problems Compiling

Problem 1: Compiling Only the `DiveLog.java` File

Because the Dive Log consists of multiple files, it is packaged using the `package` keyword. This tells the compiler where to locate the class files for the application and the packages the Dive Log uses. Unlike small one class applications, you can't just use the command `javac DiveLog.java`. If you do, you get an error that looks something like this:

```
DiveLog.java:60: cannot resolve symbol
symbol  : class Welcome
```

```
location: class divelog.DiveLog
                                new Welcome(),
                                    ^
DiveLog.java:66: cannot resolve symbol
symbol  : class Diver
location: class divelog.DiveLog
                                new Diver(),
                                    ^
DiveLog.java:71: cannot resolve symbol
symbol  : class Dives
location: class divelog.DiveLog
                                new Dives(),
                                    ^
DiveLog.java:76: cannot resolve symbol
symbol  : class Statistics
location: class divelog.DiveLog
                                new Statistics(),
                                    ^
DiveLog.java:81: cannot resolve symbol
symbol  : class WebSite
location: class divelog.DiveLog
                                new WebSite(),
                                    ^
DiveLog.java:85: cannot resolve symbol
symbol  : class Resources
location: class divelog.DiveLog
                                new Resources(),
                                    ^
6 errors
```

Solution to Problem 1

To compile a packaged application you must include the classpath which points to, but does not include, the directory where the files live. If your Dive Log files are in the following directory:

```
C:\Applications\divelog
```

Step 1: `cd` to the `divelog` directory.
Step 2: Compile the Dive Log application with the following command:

```
javac -classpath C:\Applications\ DiveLog.java
```

Note the space after `C:\Applications\`. Because you're in the `divelog` directory, the compiler knows where to find `DiveLog.java`. The `divelog` package name in the file itself tells the compiler where to look for any other classes that you created as a part of that package.

Problem 2: Leaving off the `.java` extension

If you leave off the `.java` extension for the `DiveLog.java` file, you get this error:

```
invalid argument: divelog.DiveLog
```

Solution to Problem 2
Always include the `.java` extension, so the file name reads `DiveLog.java` when compiling.

Problem 3: Misplaced Curly Braces

Misplaced curly braces are the most common cause of compilation errors. It is easy to lose track of which braces go with certain blocks of code, especially nested blocks that contain `if/else` statements or similar constructs.

Leaving out or having an extra curly brace in your code can lead to a variety of compilation errors that give little indication of the real problem. At other times, the compiler may highlight a brace, saying you need to include another.

Solution to Problem 3

Comment braces as you create them. Mark the opening brace with a comment, such as `{// Opens class`, or `{//Opens buildMenu method`. Do the same with all closing braces, such as `}// Closes class`, or `}//Closes buildMenu method`. This helps keep track of all braces and to ensure you don't have extra or missing braces. Example code with comments.

Problem 4: Calling Methods from the Wrong Place

The Dive Log application calls all its methods from with the constructor or from within the `main` method. If you try to call a method outside the constructor or some other method, you get a compile error.

Solution to Problem 4

Defining and calling methods can be tricky, so to simplify, only the `DiveLog.java` file contains a `main` method, and all the classes contain a constructor that calls the methods. The methods are called from within the constructor, and they are defined outside of the constructor.

Step 1: Check the curly braces to be certain there aren't extra or missing braces.
Step 2: If the method is being called outside the constructor, move it inside the constructor.
Step 3: Make certain the the methods are clearly defined outside of the method or constructor calling that method.

Problems Running the Application

Running the Dive Log application is similar to compiling, but troubleshooting can be more difficult. If your classes compile without error, but the application does not run correctly, determining the problem can be frustrating and difficult. Below are some of the more common problems.

Problem 1: Trying to Run Only `DiveLog.java`

If you run the `DiveLog` class and this error appeared on the command line:

```
Exception in thread "main" java.lang.NoClassDefFoundError: DiveLog/java
```

You probably tried to run the application with:

```
java DiveLog
```

For a single class application that command normally works. For a multi-class, packaged application it does not.

Solution to Problem 1

The rules for compiling the Dive Log apply to running it as well.

To run a packaged application you must include the classpath which points to, but does not include, the directory where the classes live. Do not include the package name with the dot, though. If your Dive Log files are in the following directory:

```
C:\Applications\divelog
```

Step 1: `cd` to the `divelog` directory.
Step 2: Run the command to compile the Dive Log application with the following:

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

Note the command to run is slightly different from the compile command:

`javac` to compile
`java` to run

Also, note the space after `C:\Applications\`

Problem 2: Naming the Directory Incorrectly

Similar to problem 1, the application needs to know where the class files are to be found. If you leave off the directory name in front of the file name, such as:

```
java -classpath C:\Applications\ DiveLog.java
```

you get an error similar to:

```
Exception in thread "main" java.lang.NoClassDefFoundError: DiveLog/java
```

You also get this error if you include the directory where the class files live in the classpath. Remember, you must `cd` *into* the `divelog` directory, then name the directory as a part of the class name to be run:

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

If you still have problems compiling or running the Dive Log application after reading this page, use the Reader Feedback form. Include the command you used, and the error it generated. Problems and solutions

will be added to this page to help all readers.

```
package divelog;

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


public class DiveLog
{ //Opens DiveLog class

  private JTabbedPane tabbedPane;
  private JFrame dlframe;



      public DiveLog()
      { //Opens DiveLog constructor

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

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

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


         // Tabbed pane with panels for Jcomponents

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

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

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

         dlframe.getContentPane().add(tabbedPane);

         dlframe.pack();
         dlframe.setSize(765, 690);
         dlframe.setBackground(Color.white);
         dlframe.setVisible(true);
         } // Ends class constructor

         private void populateTabbedPane()
           { // Opens populateTabbedPane method definition
           // 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()
     { // Opens buildMenu method definition
       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()
       { // Opens addActionListener method
         public void actionPerformed(ActionEvent e)
         { // Opens actionPerformed method
            System.exit(0);
          } // Closes actionPerformed method

         }); // Closes addActionListener method

         menu.add(item);
         mb.add(menu);
         dlframe.setJMenuBar(mb);
         }// Closes buildMenu method


 // main method and entry point for app
 public static void main(String[] args)
 { // Opens main method

  DiveLog dl = new DiveLog();
```

```
     } // Closes main method


} //Ends class DiveLog
```

```
package divelog;

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


public class DiveLog
{ // Opens class

  private JTabbedPane tabbedPane;
  private JFrame dlframe;



      public DiveLog()
      { // Opens constructor

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

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

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


         // 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();


         }// Ends constructor




 } //Ends class
```

```
package divelog;

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


public class DiveLog
{ // Opens class

  private JTabbedPane tabbedPane;
  private JFrame dlframe;



      public DiveLog()
      { // Opens contructor

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

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

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


         // 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();

         dlframe.getContentPane().add(tabbedPane);

         dlframe.pack();
         dlframe.setSize(765, 690);
         dlframe.setBackground(Color.white);
         dlframe.setVisible(true);

         } //Ends the constructor

         private void populateTabbedPane()
           { // Opens populateTabbedPane method
           // 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()
{ // Opens buildMenu method
  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()
  { // Opens addActionListener method
    public void actionPerformed(ActionEvent e)
    { // Opens actionPerformed method
       System.exit(0);
     } // Closes actionPerformed method

    }); // Closes addActionListener method

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

    } //Ends the buildMenu method
```

```
} //Ends class
```

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

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


public class Welcome extends JPanel
{ // Opens class


} // Closes class
```

```
package divelog;

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



public class Diver extends JPanel
{ // Opens class


} // Closes class
```

```
package divelog;

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


public class Dives extends JPanel
{ // Opens class


} // Closes class
```

```
package divelog;

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


public class Statistics extends JPanel
{ // Opens class


} // Closes class
```

```
package divelog;

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



public class WebSite extends JPanel
{ // Opens class


} // Closes class
```

```
package divelog;

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


public class Resources extends JPanel
{ // Opens class


} // Closes class
```