Essentials of the Java™ Programming Language: A Hands-On Guide, Part 1

by Monica Pawlan

[CONTENTS] [NEXT>>]

If you are new to programming in the Java™ language, have some experience with other languages, and are familiar with things like displaying text or graphics or performing simple calculations, this tutorial could be for you. It walks through how to use the Java® 2 Platform software to create and run three common types of programs written for the Java platform—applications, applets, and servlets.

You will learn how applications, applets, and servlets are similar and different, how to build a basic user interface that handles simple end user input, how to read data from and write data to files and databases, and how to send and receive data over the network. This tutorial is not comprehensive, but instead takes you on a straight and uncomplicated path through the more common programming features available in the Java platform.

If you have no programming experience at all, you might still find this tutorial useful; but you also might want to take an introductory programming course or read Teach Yourself Java 2 Online in Web Time before you proceed.

Contents

- More Information

Lesson 6: File Access and Permissions

- File Access by Applications
- Exception Handling
- File Access by Applets
- Granting Applets Permission
- Restricting Applications
- File Access by Servlets
- Appending
- More Information

Lesson 7: Database Access and Permissions

- Database Setup
- Create Database Table
- Database Access by Applications
  - Establishing a Database Connection
  - Final and Private Variables
  - Writing and Reading Data
- Database Access by Applets
  - JDBC Driver
  - JDBC-ODBC Bridge with ODBC Driver
- Database Access by Servlets
- More Information

Lesson 8: Remote Method Invocation

- About the Example
  - Program Behavior
  - File Summary
  - Compile the Example
  - Start the RMI Registry
  - Run the RemoteServer Server Object
  - Run the RMIClient1 Program
  - Run the RMIClient2 Program
- RemoteSend Class
- Send Interface
- RMIClient1 Class
- RMIClient2 Class
- More Information

In Closing

Reader Feedback

Tell us what you think of this training book.

[Duke] ○ Very worth reading ○ Worth reading ○ Not worth reading

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

Submit Reset

[TOP]

Print Button

[This page was updated: 6-Apr-2000]

For more information on Java technology
and other software from Sun Microsystems, call:
(800) 786-7638
Outside the U.S. and Canada, dial your country's
AT&T Direct Access Number first.

Sun Microsyste

Java™ Programming Language Basics, Part 1

Lesson 1: Compiling and Running A Simple Program

[<<BACK] [CONTENTS] [NEXT>>]

The computer age is here to stay. Households and businesses all over the world use computers in one way or another because computers help individuals and businesses perform a wide range of tasks with speed, accuracy, and efficiency. Computers can perform all kinds of tasks ranging from running an animated 3D graphics application with background sound to calculating the number of vacation days you have coming to handling the payroll for a Fortune 500 company.

When you want a computer to perform tasks, you write a program. A program is a sequence of instructions that define tasks for the computer to execute. This lesson explains how to write, compile, and run a simple program written in the Java™ language (Java program) that tells your computer to print a one-line string of text on the console.

But before you can write and compile programs, you need to understand what the Java platform is, and set your computer up to run the programs.

A Word About the Java Platform

The Java platform consists of the Java application programming interfaces (APIs) and the Java[1] virtual machine (JVM).

Java APIs are libraries of compiled code that you can use in your programs. They let you add ready-made and customizable functionality to save you programming time.

The simple program in this lesson uses a Java API to print a line of text to the console. The console printing capability is provided in the API ready for you to use; you supply the text to be printed.

Java programs are run (or interpreted) by another program called the Java VM. If you are familiar with Visual Basic or another interpreted language, this concept is probably familiar to you. Rather than running directly on the native operating system, the program is interpreted by the Java VM for the native operating system. This means that any computer system with the Java VM installed can run Java programs regardless of the computer system on which the applications were originally developed.

For example, a Java program developed on a Personal Computer (PC) with the Windows NT operating system should run equally well without modification on a Sun Ultra workstation with the Solaris operating system, and vice versa.

Setting Up Your Computer

Before you can write and run the simple Java program in this lesson, you need to install the Java platform on your computer system.

The Java platform is available free of charge from the java.sun.com web site. You can choose between the Java® 2 Platform software for Windows 95/98/NT or for Solaris. The download page contains the information you need to install and configure the Java platform for writing and running Java programs.

Note: Make sure you have the Java platform installed and configured for your system before you try to write and run the simple program presented next.

Writing a Program

The easiest way to write a simple program is with a text editor. So, using the text editor of your choice, create a text file with the following text, and be sure to name the text file `ExampleProgram.java`. Java programs are case sensitive, so if you type the code in yourself, pay particular attention to the capitalization.

```
//A Very Simple Example
class ExampleProgram {
  public static void main(String[] args){
    System.out.println("I'm a Simple Program");
  }
}
```

Here is the ExampleProgram.java source code file if you do not want to type the program text in yourself.

Compiling the Program

A program has to be converted to a form the Java VM can understand so any computer with a Java VM can interpret and run the program. Compiling a Java program means taking the programmer-readable text in your program file (also called source code) and converting it to bytecodes, which are platform-independent instructions for the Java VM.

The Java compiler is invoked at the command line on Unix and DOS shell operating systems as follows:

```
javac ExampleProgram.java
```

Note: Part of the configuration process for setting up the Java platform is setting the class path. The class path can be set using either the `-classpath` option with the `javac` compiler command and `java`

interpreter command, or by setting the CLASSPATH environment variable. You need to set the class path to point to the directory where the `ExampleProgram` class is so the compiler and interpreter commands can find it. See Java 2 SDK Tools for more information.

Interpreting and Running the Program

Once your program successfully compiles into Java bytecodes, you can interpret and run applications on any Java VM, or interpret and run applets in any Web browser with a Java VM built in such as Netscape or Internet Explorer. Interpreting and running a Java program means invoking the Java VM byte code interpreter, which converts the Java byte codes to platform-dependent machine codes so your computer can understand and run the program.

The Java interpreter is invoked at the command line on Unix and DOS shell operating systems as follows:

```
java ExampleProgram
```

At the command line, you should see:

```
I'm a Simple Program
```

Here is how the entire sequence looks in a terminal window:

Common Compiler and Interpreter Problems

If you have trouble compiling or running the simple example in this lesson, refer to the Common Compiler and Interpreter Problems lesson in The Java Tutorial for troubleshooting help.

Code Comments

Code comments are placed in source files to describe what is happening in the code to someone who might be reading the file, to comment-out lines of code to isolate the source of a problem for debugging purposes, or to generate API documentation. To these ends, the Java language supports three kinds of comments: double slashes, C-style, and doc comments.

Double Slashes

Double slashes (`//`) are used in the C++ programming language, and tell the compiler to treat everything from the slashes to the end of the line as text.

```
//A Very Simple Example
class ExampleProgram {
  public static void main(String[] args){
    System.out.println("I'm a Simple Program");
  }
}
```

C-Style Comments

Instead of double slashes, you can use C-style comments (`/* */`) to enclose one or more lines of code to be treated as text.

```
/* These are
C-style comments
*/
class ExampleProgram {
  public static void main(String[] args){
    System.out.println("I'm a Simple Program");
  }
}
```

Doc Comments

To generate documentation for your program, use the doc comments (`/** */`) to enclose lines of text for the `javadoc` tool to find. The `javadoc` tool locates the doc comments embedded in source files and uses those comments to generate API documentation.

```
/** This class displays a text string at
*  the console.
*/
```

```
class ExampleProgram {
  public static void main(String[] args){
    System.out.println("I'm a Simple Program");
  }
}
```

With one simple class, there is no reason to generate API documentation. API documentation makes sense when you have an application made up of a number of complex classes that need documentation. The tool generates HTML files (Web pages) that describe the class structures and contain the text enclosed by doc comments. The javadoc Home Page has more information on the `javadoc` command and its output.

API Documentation

The Java platform installation includes API Documentation, which describes the APIs available for you to use in your programs. The files are stored in a doc directory beneath the directory where you installed the platform. For example, if the platform is installed in `/usr/local/java/jdk1.2`, the API Documentation is in `/usr/local/java/jdk1.2/doc/api`.

More Information

See Java 2 SDK Tools for more information on setting the class path and using the `javac`, and `java` commands.

See Common Compiler and Interpreter Problems lesson in The Java Tutorial for troubleshooting help.

The javadoc Home Page has more information on the `javadoc` command and its output.

You can also view the API Documentation for the Java 2 Platform on the java.sun.com site.

_____

[1] As used on this web site, the terms "Java virtual machine" or "JVM" mean a virtual machine for the Java platform.

[TOP]

Print Button

[This page was updated: 31-Mar-2000]

Java™ Programming Language Basics, Part 1

Lesson 2: Building Applications

[<<BACK] [CONTENTS] [NEXT>>]

All programs written in the Java™ language (Java programs) are built from classes. Because all classes have the same structure and share common elements, all Java programs are very similar.

This lesson describes the structure and elements of a simple application created from one class. The next lesson covers the same material for applets.

Application Structure and Elements

An application is created from classes. A `class` is similar to a RECORD in the Pascal language or a `struct` in the C language in that it stores related data in *fields*, where the fields can be different types. So you could, for example, store a text string in one field, an integer in another field, and a floating point in a third field. The difference between a class and a RECORD or `struct` is that a class also defines the *methods* to work on the data.

For example, a very simple class might store a string of text and define one method to set the string and another method to get the string and print it to the console. Methods that work on the data are called *accessor* methods.

Every application needs one class with a `main` method. This class is the entry point for the program, and is the class name passed to the `java` interpreter command to run the application.

The code in the `main` method executes first when the program starts, and is the control point from which the controller class accessor methods are called to work on the data.

Here, again, is the example program from Lesson 1. It has no fields or accessor methods, but because it is the only class in the program, it has a `main` method.

```
class ExampleProgram {
   public static void main(String[] args){
     System.out.println("I'm a Simple Program");
   }
}
```

The `public static void` keywords mean the Java[1] virtual machine (JVM) interpreter can call the program's `main` method to start the program (public) without creating an instance of the class (static), and the program does not return data to the Java VM interpreter (void) when it ends.

An instance of a class is an executable copy of the class While the class describes the data and behavior, you need a class instance to acquire and work on data. The diagram at the left shows three instances of the `ExampleProgram` class by the names: `FirstInstance`, `SecondInstance` and

ThirdInstance.

The main method is static to give the Java VM interpreter a way to start the class without creating an instance of the control class first. Instances of the control class are created in the main method after the program starts.

The main method for the simple example does not create an instance of the ExampleProgram class because none is needed. The ExampleProgram class has no other methods or fields, so no class instance is needed to access them from the main method. The Java platform lets you execute a class without creating an instance of that class as long as its static methods do not call any non-static methods or fields.

The ExampleProgram class just calls println, which is a static method in the System class. The java.lang.System class, among other things, provides functionality to send text to the terminal window where the program was started. It has all static fields and methods.

The static fields and methods of a class can be called by another program without creating an instance of the class. So, just as the Java VM interpreter command could call the static main method in the ExampleProgram class without creating an instance of the ExampleProgram class, the ExampleProgram class can call the static println method in the System class, without creating an instance of the System class.

However, a program must create an instance of a class to access its non-static fields and methods. Accessing static and non-static fields and methods is discussed further with several examples in the next section.

Fields and Methods

The LessonTwoA.java program alters the simple example to store the text string in a static field called text. The text field is static so its data can be accessed directly without creating an *instance* of the LessonTwoA class.

```
class LessonTwoA {
   static String text = "I'm a Simple Program";
   public static void main(String[] args){
```

```
        System.out.println(text);
    }
}
```

The LessonTwoB.java and LessonTwoC.java programs add a `getText` method to the program to retrieve and print the text.

The LessonTwoB.java program accesses the non-static `text` field with the non-static `getText` method. Non-static methods and fields are called instance methods and fields. This approach requires that an instance of the `LessonTwoB` class be created in the `main` method. To keep things interesting, this example includes a static text field and a non-static instance method (`getStaticText`) to retrieve it.

Note: The field and method return values are all type `String`.

```
class LessonTwoB {

   String text = "I'm a Simple Program";
   static String text2 = "I'm static text";

   String getText(){
        return text;
   }

   String getStaticText(){
        return text2;
   }

   public static void main(String[] args){
        LessonTwoB progInstance = new LessonTwoB();
        String retrievedText = progInstance.getText();
        String retrievedStaticText =
                  progInstance.getStaticText();
        System.out.println(retrievedText);
        System.out.println(retrievedStaticText);
   }
}
```

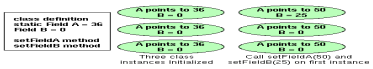
The LessonTwoC.java program accesses the static `text` field with the static `getText` method. Static methods and fields are called class methods and fields. This approach allows the program to

call the static `getText` method directly without creating an instance of the `LessonTwoC` class.

```
class LessonTwoC {

   static String text = "I'm a Simple Program";

//Accessor method
   static String getText(){
        return text;
   }

   public static void main(String[] args){
        String retrievedText = getText();
        System.out.println(retrievedText);
   }
}
```

So, class methods can operate only on class fields, and instance methods can operate on class and instance fields.

You might wonder what the difference means. In short, there is only one copy of the data stored or set in a class field but each instance has its own copy of the data stored or set in an instance field.



The figure above shows three class instances with one static field and one instance field. At runtime, there is one copy of the value for static Field A and each instance points to the one copy. When setFieldA(50) is called on the first instance, the value of the one copy changes from 36 to 50 and all three instances point to the new value. But, when setFieldB(25) is called on the first instance, the value for Field B changes from 0 to 25 for the first instance only because each instance has its own copy of Field B.

See Understanding Instance and Class Members lesson in The Java tutorial for a thorough discussion of this topic.

Constructors

Classes have a special method called a *constructor* that is called when a class instance is created. The class constructor always has the same name as the class and no return type. The LessonTwoD program converts the LessonTwoB program to use a constructor to initialize the text string.

Note: If you do not write your own constructor, the compiler adds an empty constructor, which calls the no-arguments constructor of its parent class. The empty constructor is called the default constructor. The default constructor initializes all non-initialized fields and variables to zero.

```
class LessonTwoD {

   String text;

//Constructor
   LessonTwoD(){
     text = "I'm a Simple Program";
   }

//Accessor method
   String getText(){
        return text;
   }

   public static void main(String[] args){
        LessonTwoD progInst = new LessonTwoD();
        String retrievedText = progInst.getText();
        System.out.println(retrievedText);
   }
}
```

To Summarize

A simple program that prints a short text string to the console would probably do everything in the main method and do away

with the constructor, `text` field, and `getText` method. But, this lesson used a very simple program to show you the structure and elements in a basic Java program.

More Information

See Understanding Instance and Class Members lesson in The Java tutorial for a thorough discussion of this topic.

_____

[1] As used on this web site, the terms "Java virtual machine" or "JVM" mean a virtual machine for the Java platform.

[TOP]

Print Button

[This page was updated: 31-Mar-2000]

Java™ Programming Language Basics, Part 1

Lesson 3: Building Applets

[<<BACK] [CONTENTS] [NEXT>>]

Like applications, applets are created from classes. However, applets do not have a `main` method as an entry point, but instead, have several methods to control specific aspects of applet execution.

This lesson converts an application from Lesson 2 to an applet and describes the structure and elements of an applet.

Application to Applet

The following code is the applet equivalent to the `LessonTwoB` application from Lesson 2. The figure below shows how the running applet looks. The structure and elements of the applet code are discussed after the section on how to run the applet just below.


```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;

public class SimpleApplet extends Applet{

  String text = "I'm a simple applet";

  public void init() {
        text = "I'm a simple applet";
        setBackground(Color.cyan);
  }
  public void start() {
        System.out.println("starting...");
  }
  public void stop() {
        System.out.println("stopping...");
  }
  public void destroy() {
        System.out.println("preparing to unload...");
  }
  public void paint(Graphics g){
        System.out.println("Paint");
        g.setColor(Color.blue);
        g.drawRect(0, 0,
                   getSize().width -1,
                   getSize().height -1);
        g.setColor(Color.red);
        g.drawString(text, 15, 25);
  }
}
```

The `SimpleApplet` class is declared `public` so the program that runs the applet (a browser or `appletviewer`), which is not local

to the program can access it.

Run the Applet

To see the applet in action, you need an HTML file with the Applet tag as follows:

```
<HTML>
<BODY>
<APPLET CODE=SimpleApplet.class WIDTH=200
HEIGHT=100>
</APPLET>
</BODY>
</HTML>
```

The easiest way to run the applet is with appletviewer shown below where `simpleApplet.html` is a file that contains the above HTML code:

```
appletviewer simpleApplet.html
```

Note: To run an applet written with Java™ 2 APIs in a browser, the browser must be enabled for the Java 2 Platform. If your browser is not enabled for the Java 2 Platform, you have to use appletviewer to run the applet or install Java Plug-in. Java Plug-in lets you run applets on web pages under the 1.2 version of the Java VM instead of the web browser's default Java VM.

Applet Structure and Elements

The Java API `Applet` class provides what you need to design the appearance and manage the behavior of an applet. This class provides a graphical user interface (GUI) component called a `Panel` and a number of methods. To create an applet, you extend (or subclass) the `Applet` class and implement the appearance and behavior you want.

The applet's appearance is created by drawing onto the `Panel` or by attaching other GUI components such as push buttons, scrollbars, or text areas to the `Panel`. The applet's behavior is defined by implementing the methods.

Extending a Class

Most classes of any complexity extend other classes. To extend another class means to write a new class that can use the fields and methods defined in the class being extended. The class being extended is the parent class, and the class doing the extending is the child class. Another way to say this is the child class inherits the fields and methods of its parent or chain of parents. Child classes either call or override inherited methods. This is called single inheritance.

The `SimpleApplet` class extends `Applet` class, which extends the `Panel` class, which extends the `Container` class. The `Container` class extends `Object`, which is the parent of all Java API classes.

The `Applet` class provides the `init`, `start`, `stop`, `destroy`, and `paint` methods you saw in the example applet. The `SimpleApplet` class overrides these methods to do what the `SimpleApplet` class needs them to do. The `Applet` class provides no functionality for these methods.

However, the `Applet` class does provide functionality for the `setBackground` method,which is called in the `init` method. The call to `setBackground` is an example of calling a method inherited from a parent class in contrast to overriding a method inherited from a parent class.

You might wonder why the Java language provides methods without implementations. It is to provide conventions for everyone to use for consistency across Java APIs. If everyone wrote their own method to start an applet, for example, but gave it a different name such as `begin` or `go`, the applet code would not be interoperable with other programs and browsers, or portable across multiple platforms. For example, Netscape and Internet Explorer know how to look for the `init` and `start` methods.

Behavior

An applet is controlled by the software that runs it. Usually, the underlying software is a browser, but it can also be

appletviewer as you saw in the example. The underlying software controls the applet by calling the methods the applet inherits from the Applet class.

The init Method: The init method is called when the applet is first created and loaded by the underlying software. This method performs one-time operations the applet needs for its operation such as creating the user interface or setting the font. In the example, the init method initializes the text string and sets the background color.

The start Method: The start method is called when the applet is visited such as when the end user goes to a web page with an applet on it. The example prints a string to the console to tell you the applet is starting. In a more complex applet, the start method would do things required at the start of the applet such as begin animation or play sounds.

After the start method executes, the event thread calls the paint method to draw to the applet's Panel. A thread is a single sequential flow of control within the applet, and every applet can run in multiple threads. Applet drawing methods are always called from a dedicated drawing and event-handling thread.

The stop and destroy Methods: The stop method stops the applet when the applet is no longer on the screen such as when the end user goes to another web page. The example prints a string to the console to tell you the applet is stopping. In a more complex applet, this method should do things like stop animation or sounds.

The destroy method is called when the browser exits. Your applet should implement this method to do final cleanup such as stop live threads.

Appearance

The Panel provided in the Applet class inherits a paint method from its parent Container class. To draw something onto the Applet's Panel, you implement the paint method to do the drawing.

The Graphics object passed to the paint method defines a *graphics context* for drawing on the Panel. The Graphics object

has methods for graphical operations such as setting drawing colors, and drawing graphics, images, and text.

The `paint` method for the `SimpleApplet` draws the *I'm a simple applet* string in red inside a blue rectangle.

```
   public void paint(Graphics g){
        System.out.println("Paint");
//Set drawing color to blue
        g.setColor(Color.blue);
//Specify the x, y, width and height for a rectangle
        g.drawRect(0, 0,
                   getSize().width -1,
                   getSize().height -1);
//Set drawing color to red
        g.setColor(Color.red);
//Draw the text string at the (15, 25) x-y location
        g.drawString(text, 15, 25);
   }
```

Packages

The applet code also has three `import` statements at the top. Applications of any size and all applets use `import` statements to access ready-made Java API classes in *packages*. This is true whether the Java API classes come in the Java platform download, from a third-party, or are classes you write yourself and store in a directory separate from the program. At compile time, a program uses `import` statements to locate and reference compiled Java API classes stored in packages elsewhere on the local or networked system. A compiled class in one package can have the same name as a compiled class in another package. The package name differentiates the two classes.

The examples in Lessons 1 and 2 did not need a package declaration to call the `System.out.println` Java API class because the `System` class is in the `java.lang` package that is included by default. You never need an `import java.lang.*` statement to use the compiled classes in that package.

More Information

You can find more information on applets in the Writing Applets trail in The Java Tutorial.

[TOP]

Print Button

[This page was updated: 31-Mar-2000]

Products & APIs | Developer Connection | Docs & Training | Online Support
Community Discussion | Industry News | Solutions Marketplace | Case Studies

Glossary - Applets - Tutorial - Employment - Business & Licensing - Java Store - Java in the Real World

FAQ | Feedback | Map | A-Z Index

For more information on Java technology
and other software from Sun Microsystems, call:
(800) 786-7638
Outside the U.S. and Canada, dial your country's
AT&T Direct Access Number first.

Sun Microsyste

Java™ Programming Language Basics, Part 1

Lesson 4: Building A User Interface

[<<BACK] [CONTENTS] [NEXT>>]

In the last lesson you saw how the `Applet` class provides a `Panel` component so you can design the applet's user interface. This lesson expands the basic application from Lessons 1 and 2 to give it a user interface using the Java™ Foundation Classes (JFC) Project Swing APIs that handle user events.

Project Swing APIs

In contrast to the applet in Lesson 3 where the user interface is attached to a panel object nested in a top-level browser, the Project Swing application in this lesson attaches its user interface to a panel object nested in a top-level frame object. A frame object is a top-level window that provides a title, banner, and methods to manage the

appearance and behavior of the window.

The Project Swing code that follows builds this simple application. The window on the left appears when you start the application, and the window on the right appears when you click the button. Click again and you are back to the original window on the left.

When Application Starts When Button Clicked

Import Statements

Here is the SwingUI.java code. At the top, you have four lines of import statements. The lines indicate exactly which Java™ API classes the program uses. You could replace four of these lines with this one line: `import java.awt.*;`, to import the entire `awt` package, but doing that increases compilation overhead than importing exactly the classes you need and no others.

```
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;
```

Class Declaration

The class declaration comes next and indicates the top-level frame for the application's user interface is a `JFrame` that implements the `ActionListener` interface.

```
class SwingUI extends JFrame
        implements ActionListener{
```

The `JFrame` class extends the `Frame` class that is part of the Abstract Window Toolkit (AWT) APIs. Project Swing extends the AWT with a full set of GUI components and services, pluggable look and feel capabilities, and assistive technology support. For a more detailed introduction to Project Swing, see the Swing Connection, and Fundamentals of Swing, Part 1.

The Java APIs provide classes and interfaces for you to use. An interface defines a set of methods, but does not implement them. The rest of the SwingUI class declaration indicates that this class will implement the ActionListener interface. This means the SwingUI class must implement all methods defined in the ActionListener interface. Fortunately, there is only one, actionPerformed, which is discussed below.

Instance Variables

These next lines declare the Project Swing component classes the SwingUI class uses. These are instance variables that can be accessed by any method in the instantiated class. In this example, they are built in the SwingUI constructor and accessed in the actionPerformed method implementation. The private boolean instance variable is visible only to the SwingUI class and is used in the actionPerformedmethod to find out whether or not the button has been clicked.

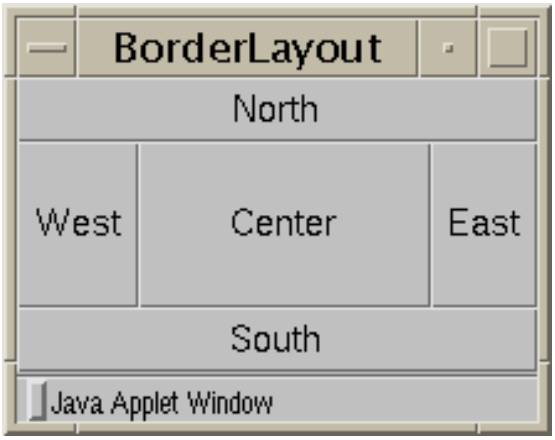
```
JLabel text, clicked;
JButton button, clickButton;
JPanel panel;
private boolean _clickMeMode = true;
```

Constructor

The constructor (shown below) creates the user interface components and JPanel object, adds the components to the JPanel object, adds the panel to the frame, and makes the JButton components event listeners. The JFrame object is created in the main method when the program starts.

```
SwingUI(){
  text = new JLabel("I'm a Simple Program");
  clicked = new JLabel("Button Clicked");

  button = new JButton("Click Me");
//Add button as an event listener
  button.addActionListener(this);

  clickButton = new JButton("Click Again");
//Add button as an event listener
  clickButton.addActionListener(this);
```

```
//Create panel
     panel = new JPanel();
//Specify layout manager and background color
     panel.setLayout(new BorderLayout(1,1));
     panel.setBackground(Color.white);
//Add label and button to panel
     getContentPane().add(panel);
     panel.add(BorderLayout.CENTER, text);
     panel.add(BorderLayout.SOUTH, button);
   }
```



When the `JPanel` object is created, the layout manager and background color are specified. The layout manager in use determines how user interface components are arranged on the display area.

The code uses the `BorderLayout` layout manager, which arranges user interface components in the five areas shown at left. To add a component, specify the area (north, south, east, west, or center).

```
//Create panel
     panel = new JPanel();
//Specify layout manager and background color
     panel.setLayout(new BorderLayout(1,1));
     panel.setBackground(Color.white);
//Add label and button to panel
     getContentPane().add(panel);
     panel.add(BorderLayout.CENTER, text);
     panel.add(BorderLayout.SOUTH, button);
   }
```

To find out about some of the other available layout managers and how to use them, see the JDC article Exploring the AWT Layout Managers.

The call to the `getContentPane` method of the `JFrame` class is for adding the `Panel` to the `JFrame`. Components are not added directly to a `JFrame`, but to its content pane. Because the layout manager controls the layout of components, it is set on the content pane where the components reside. A content pane

provides functionality that allows different types of components to work together in Project Swing.

Action Listening

In addition to implementing the `ActionListener` interface, you have to add the event listener to the JButton components. An action listener is the SwingUI object because it implements the ActionListener interface. In this example, when the end user clicks the button, the underlying Java platform services pass the action (or event) to the actionPerformed method. In your code, you implement the actionPerformed method to take the appropriate action based on which button is clicked..

The component classes have the appropriate add methods to add action listeners to them. In the code the JButton class has an addActionListener method. The parameter passed to addActionListener is this, which means the SwingUI action listener is added to the button so button-generated actions are passed to the actionPerformed method in the SwingUI object.

```
     button = new JButton("Click Me");
//Add button as an event listener
     button.addActionListener(this);
```

Event Handling

The actionPerformed method is passed an event object that represents the action event that occurred. Next, it uses an if statement to find out which component had the event, and takes action according to its findings.

```
   public void actionPerformed(ActionEvent event){
     Object source = event.getSource();
        if (_clickMeMode) {
          text.setText("Button Clicked");
          button.setText("Click Again");
          _clickMeMode = false;
        } else {
          text.setText("I'm a Simple Program");
          button.setText("Click Me");
          _clickMeMode = true;
        }
   }
```

You can find information on event handling for the different

components in The Java Tutorial section on Event Handling.

Main Method

The `main` method creates the top-level `frame`, sets the title, and includes code that lets the end user close the window using the frame menu.

```
public static void main(String[] args){
//Create top-level frame
  SwingUI frame = new SwingUI();
  frame.setTitle("Example");
//This code lets you close the window
  WindowListener l = new WindowAdapter() {
     public void windowClosing(WindowEvent e) {
       System.exit(0);
     }
  };
    frame.addWindowListener(l);
//This code lets you see the frame
  frame.pack();
  frame.setVisible(true);
 }
}
```

The code for closing the window shows an easy way to add event handling functionality to a program. If the event listener interface you need provides more functionality than the program actually uses, use an adapter class. The Java APIs provide adapter classes for all listener interfaces with more than one method. This way, you can use the adapter class instead of the listener interface and implement only the methods you need. In the example, the WindowListener interface has 7 methods and this program needs only the windowClosing method so it makes sense to use the WindowAdapter class instead.

This code extends the WindowAdapter class and overrides the windowClosing method. The new keyword creates an anonymous instance of the extended inner class. It is anonymous because you are not assigning a name to the class and you cannot create another instance of the class without executing the code again. It is an inner class because the extended class definition is nested within the SwingUI class.

This approach takes only a few lines of code, while implementing the WindowListener interface would require 6 empty method

implementations. Be sure to add the WindowAdapter object to the frame object so the frame object will listen for window events.

```
  WindowListener l = new WindowAdapter() {
//The instantiation of object l is extended to
//include this code:
    public void windowClosing(WindowEvent e){
       System.exit(0);
    }
  };
  frame.addWindowListener(l);
```

Applets Revisited

Using what you learned in Lesson 3: Building Applets and this lesson, convert the example for this lesson into an applet. Give it a try before looking at the solution.

In short, the differences between the applet and application versions are the following:

- The applet class is declared `public` so `appletviewer` can access it.
- The applet class descends from `Applet` and the application class descends from `JFrame`.
- The applet version has no `main` method.
- The application constructor is replaced in the applet by `start` and `init` methods.

- GUI components are added directly to the `Applet`; whereas, in the case of an application, GUI components are added to the content pane of its `JFrame` object.

More Information

For more information on Project Swing, see the Swing Connection, and Fundamentals of Swing, Part 1.

Also see The JFC Project Swing Tutorial: A Guide to Constructing GUIs.

To find out about some of the other available layout managers and how to use them, see the JDC article Exploring the AWT Layout Managers.

[TOP]

Print Button

[This page was updated: 31-Mar-2000]

Java™ Programming Language Basics, Part 1

Lesson 5: Writing Servlets

[<<BACK] [CONTENTS] [NEXT>>]

A servlet is an extension to a server that enhances the server's functionality. The most common use for a servlet is to extend a web server by providing dynamic web content. Web servers display documents written in HyperText Markup Language (HTML) and respond to user requests using the HyperText Transfer Protocol (HTTP). HTTP is the protocol for moving hypertext files across the internet. HTML documents contain text that has been marked up for interpretation by an HTML browser such as Netscape.

Servlets are easy to write. All you need is the Java® 2 Platform software, and JavaServer™ Web Development Kit (JWSDK). You can download a free copy of the JWSDK.

This lesson shows you how to create a very simple form that invokes a basic servlet to process end user data entered on the form.

About the Example

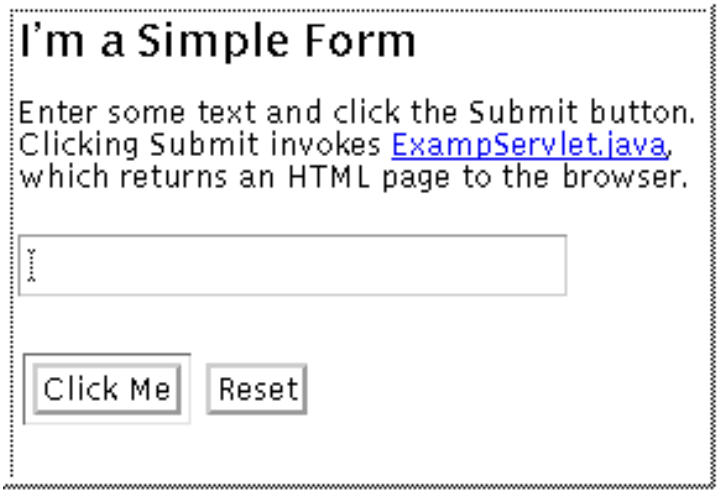
A browser accepts end user input through an HTML form. The simple form used in this lesson has one text input field for the end user to enter text and a Submit button. When the end user clicks the Submit button, the simple servlet is invoked to process

the end user input.

In this example, the simple servlet returns an HTML page that displays the text entered by the end user.

HTML Form

The HTML form is embedded in this HTML file. The diagram shows how the HTML page looks when it is opened in a browser.



The HTML file and form are similar to the simple application and applet examples in Lesson 4 so you can compare the code and learn how servlets, applets, and applications handle end user inputs.

When the user clicks the `Click Me` button, the servlet gets the entered text, and returns an HTML page with the text.

The HTML page returned to the browser by the ExampServlet.java servlet is shown below. The servlet code to retrieve the user's input and generate the HTML page follows with a discussion.

Note: To run the example, you have to put the servlet and HTML files in the correct directories for the Web server you are using. For example, with Java WebServer 1.1.1, you place the servlet in the `~/JavaWebServer1.1.1/servlets` and the HTML file in the `~/JavaWebServer1.1.1/public_html` directory.

Servlet Backend

ExampServlet.java builds an HTML page to return to the end user. This means the servlet code does not use any Project Swing or Abstract Window Toolkit (AWT) components or have event handling code. For this simple servlet, you only need to import these packages:

- `java.io` for system input and output. The `HttpServlet` class uses the `IOException` class in this package to signal that an input or output exception of some kind has occurred.
- `javax.servlet`, which contains generic (protocol-independent) servlet classes. The `HttpServlet` class uses the `ServletException` class in this package to indicate a servlet problem.
- `javax.servlet.http`, which contains HTTP servlet classes. The `HttpServlet` class is in this package.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ExampServlet extends HttpServlet {
  public void doPost(HttpServletRequest request,
         HttpServletResponse response)
        throws ServletException, IOException
  {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("<title>Example</title>" +
       "<body bgcolor=FFFFFF>");
```

```
    out.println("<h2>Button Clicked</h2>");

    String DATA = request.getParameter("DATA");

    if(DATA != null){
      out.println(DATA);
    } else {
      out.println("No text entered.");
    }

    out.println("<P>Return to
        <A HREF="../simpleHTML.html">Form</A>");
    out.close();
  }
}
```

Class and Method Declarations

All servlet classes extend the `HttpServlet` abstract class. `HttpServlet` simplifies writing HTTP servlets by providing a framework for handling the HTTP protocol. Because `HttpServlet` is `abstract`, your servlet class must extend it and override at least one of its methods. An `abstract` class is a class that contains unimplemented methods and cannot be instantiated itself.

```
public class ExampServlet extends HttpServlet {
  public void doPost(HttpServletRequest request,
         HttpServletResponse response)
         throws ServletException, IOException
  {
```

The `ExampServlet` class is declared `public` so the web server that runs the servlet, which is not local to the servlet, can access it.

The `ExampServlet` class defines a `doPost` method with the same name, return type, and parameter list as the `doPost` method in the `HttpServlet` class. By doing this, the `ExampServlet` class overrides and implements the `doPost` method in the `HttpServlet` class.

The `doPost` method performs the HTTP POST operation, which is the type of operation specified in the HTML form used for this example. The other possibility is the HTTP GET operation, in

which case you would implement the `doGet` method instead.

In short, POST requests are for sending any amount of data directly over the connection without changing the URL, and GET requests are for getting limited amounts of information appended to the URL. POST requests cannot be bookmarked or emailed and do not change the Uniform Resource Locators (URL) of the response. GET requests can be bookmarked and emailed and add information to the URL of the response.

The parameter list for the `doPost` method takes a `request` and a `response` object. The browser sends a request to the servlet and the servlet sends a response back to the browser.

The `doPost` method implementation accesses information in the `request` object to find out who made the request, what form the request data is in, and which HTTP headers were sent, and uses the `response` object to create an HTML page in response to the browser's request. The `doPost` method throws an `IOException` if there is an input or output problem when it handles the request, and a `ServletException` if the request could not be handled. These exceptions are handled in the `HttpServlet` class.

Method Implementation

The first part of the `doPost` method uses the `response` object to create an HTML page. It first sets the response content type to be `text/html`, then gets a `PrintWriter` object for formatted text output.

```
response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("<title>Example</title>" +
       "<body bgcolor=#FFFFFF>");

    out.println("<h2>Button Clicked</h2>");
```

The next line uses the `request` object to get the data from the text field on the form and store it in the DATA variable. The `getparameter` method gets the named parameter, returns `null` if the parameter was not set, and an empty string if the parameter was sent without a value.

```
    String DATA = request.getParameter("DATA");
```

The next part of the `doPost` method gets the data out of the DATA parameter and passes it to the `response` object to add to the HTML response page.

```
    if(DATA != null){
      out.println(DATA);
    } else {
      out.println("No text entered.");
    }
```

The last part of the `doPost` method creates a link to take the end user from the HTML response page back to the original form, and closes the response.

```
    out.println("<P>Return to
        <A HREF="../simpleHTML.html">Form</A>");
    out.close();
  }
```

Note: To learn how to use the other methods available in the `HttpServlet`, `HttpServletRequest`, and `HttpServletResponse` classes, see The Java Tutorial trail on Servlets.

More Information

You can find more information on servlets in the Servlets trail in The Java Tutorial.

[TOP]

Print Button

[This page was updated: 31-Mar-2000]

For more information on Java technology
and other software from Sun Microsystems, call:
(800) 786-7638
Outside the U.S. and Canada, dial your country's
AT&T Direct Access Number first.

Java™ Programming Language Basics, Part 1

Lesson 6: File Access and Permissions

[<<BACK] [CONTENTS] [NEXT>>]

So far, you have learned how to retrieve and handle a short text string entered from the keyboard into a simple graphical user interface (GUI). But programs also retrieve, handle, and store data in files and databases.

This lesson expands the examples from previous lessons to perform basic file access using the application programming interfaces (APIs) in the `java.io` package. It also shows you how to grant applets permission to access specific files, and how to restrict an application so it has access to specific files only.

File Access by Applications

The Java® 2 Platform software provides a rich range of classes for reading character or byte data into a program, and writing character or byte data out to an external file, storage device, or program. The source or destination might be on the local

computer system where the program is running or anywhere on the network.

This section shows you how to read data from and write data to a file on the local computer system. See The Java™ Tutorial trail on Reading and Writing for information on transferring data between programs, between a program and memory, and performing operations such as buffering or character encoding on data as it is read or written.

- **Reading:** A program opens an input *stream* on the file and reads the data in serially (in the order it was written to the file).
- **Writing:** A program opens an output stream on the file and writes the data out serially.

This first example converts the SwingUI.java example from Lesson 4 to accept user input through a text field. The window on the left appears when you start the FileIO application, and the window on the right appears when you click the button. When you click the button, whatever is entered into the text field is saved to a file. After that, another file is opened and read and its text is displayed in the window on the right. Click again and you are back to the original window with a blank text field ready for more input.

When Application Starts When Button Clicked

The conversion from the SwingUI.java program for Lesson 4 to the FileIO.java program for this lesson primarily involves the `constructor` and the `actionPerformed` method as described here.

Constructor and Instance Variable Changes

A `JTextfield` instance variable is added to the class so the

constructor can instantiate the object and the actionPerformed method can access the text the end user types into it.

The constructor instantiates the JTextField with a value of 20. This value tells the Java platform the number of columns to use to calculate the preferred width of the field. Lower values result in a narrower display, and likewise, higher values result in a wider display.

The text label is added to the North section of the BorderLayout so the JTextField can be added to the Center section.

Note: You can learn more about component sizing in The Java Tutorial sections on Solving Common Layout Problems and Layout Management.

```
//Instance variable for text field
JTextField textField;

FileIO(){
  text = new JLabel("Text to save to file:");
  clicked = new
        JLabel("Text retrieved from file:");

  button = new JButton("Click Me");
  button.addActionListener(this);

  clickButton = new JButton("Click Again");
  clickButton.addActionListener(this);

//Text field instantiation
  textField = new JTextField(20);

  panel = new JPanel();
  panel.setLayout(new BorderLayout());
  panel.setBackground(Color.white);
  getContentPane().add(panel);

//Adjustments to layout to add text field
  panel.add("North", text);
  panel.add("Center", textField);
  panel.add("South", button);
```

```
}
```

Method Changes

The `actionPerformed` method uses the `FileInputStream` and `FileOutputStream` classes to read data from and write data to a file. These classes handle data in byte streams, as opposed to character streams, which are shown in the applet example. A more detailed explanation of the changes to the method implementation follows the code.

```
public void actionPerformed(
                ActionEvent event){
  Object source = event.getSource();
  if(source == button){
//Variable to display text read from file
    String s = null;
    if(_clickMeMode){
      try{
//Code to write to file
        String text = textField.getText();
        byte b[] = text.getBytes();

        String outputFileName =
                System.getProperty("user.home",
          File.separatorChar + "home" +
          File.separatorChar + "monicap") +
          File.separatorChar + "text.txt";
          File outputFile = new File(outputFileName);
          FileOutputStream out = new
                FileOutputStream(outputFile);
          out.write(b);
          out.close();

//Code to read from file
        String inputFileName =
                System.getProperty("user.home",
          File.separatorChar + "home" +
          File.separatorChar + "monicap") +
          File.separatorChar + "text.txt";
          File inputFile = new File(inputFileName);
          FileInputStream in = new
                FileInputStream(inputFile);

        byte bt[] = new
                byte[(int)inputFile.length()];
        in.read(bt);
```

```
            s = new String(bt);
            in.close();
           }catch(java.io.IOException e){
            System.out.println("Cannot access text.txt");
           }
//Clear text field
      textField.setText("");
//Display text read from file
      text.setText("Text retrieved from file:");
      textField.setText(s);
      button.setText("Click Again");
      _clickMeMode = false;
    } else {
//Save text to file
      text.setText("Text to save to file:");
      textField.setText("");
      button.setText("Click Me");
      _clickMeMode = true;
    }
  }
}
```

To write the end user text to a file, the text is retrieved from the `textField` and converted to a byte array.

```
String text = textField.getText();
byte b[] = text.getBytes();
```

Next, a `File` object is created for the file to be written to and used to create a `FileOutputStream` object.

```
      String outputFileName =
                System.getProperty("user.home",
        File.separatorChar + "home" +
        File.separatorChar + "monicap") +
        File.separatorChar + "text.txt";
        File outputFile = new File(outputFileName);
        FileOutputStream out = new
                FileOutputStream(outputFile);
```

Finally, the `FileOutputStream` object writes the byte array to the `File` object and closes the output stream when the operation completes.

```
out.write(b);
```

```
out.close();
```

The code to open a file for reading is similar. To read text from a file, a `File` object is created and used to create a `FileInputStream` object.

```
String inputFileName =
          System.getProperty("user.home",
  File.separatorChar + "home" +
  File.separatorChar + "monicap") +
  File.separatorChar + "text.txt";
  File inputFile = new File(inputFileName);
  FileInputStream out = new
          FileInputStream(inputFile);
```

Next, a `byte` array is created the same size as the file into which the file contents are read.

```
byte bt[] = new byte[(int)inputFile.length()];
in.read(bt);
```

Finally, the byte array is used to construct a `String` object, which is used to create the text for the `label` component. The `FileInputStream` is closed when the operation completes.

```
String s = new String(bt);
label.setText(s);
in.close();
```

System Properties

The above code used a call to `System.getProperty` to create the pathname to the file in the user's home directory. The `System` class maintains a set of properties that define attributes of the current working environment. When the Java platform starts, system properties are initialized with information about the runtime environment including the current user, Java platform version, and the character used to separate components of a file name (`File.separatorChar`).

The call to `System.getProperty` uses the keyword `user.home` to get the user's home directory and supplies the default value `File.separatorChar + "home" + File.separatorChar + "monicap"`) in case no value is found for this key.

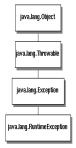
File.separatorChar

The above code used the `java.io.File.separatorChar` variable to construct the directory pathname. This variable is initialized to contain the file separator value stored in the `file.separator` system property and gives you a way to construct platform-independent pathnames.

For example, the pathname `/home/monicap/text.txt` for Unix and `\home\monicap\text.txt` for Windows are both represented as `File.separatorChar + "home" + File.separatorChar + "monicap" + File.separatorChar + "text.txt"` in a platform-independent construction.

Exception Handling

An exception is a class that descends from either `java.lang.Exception` or `java.lang.RuntimeException` that defines mild error conditions your program might encounter. Rather than letting the program terminate, you can write code to handle exceptions and continue program execution.



The file input and output code in the `actionPerformed` method is enclosed in a `try` and `catch` block to handle the `java.lang.IOException` that might be thrown by code within the block.

`java.lang.IOException` is what is called a checked exception. The Java platform requires that a method catch or specify all checked exceptions that can be thrown within the scope of a method.

Checked exceptions descend from `java.lang.Throwable`. If a checked exception is not either caught or specified, the compiler throws an error.

In the example, the `try` and `catch` block catches and handles the `java.io.IOException` checked exception. If a method does not catch a checked exception, the method must specify that it can throw the exception because an exception that can be thrown

by a method is really part of the method's public interface. Callers of the method must know about the exceptions that a method can throw so they can take appropriate actions.

However, the `actionPerformed` method already has a public interface definition that cannot be changed to specify the `java.io.IOException`, so in this case, the only thing to do is catch and handle the checked exception. Methods you define yourself can either specify exceptions or catch and handle them, while methods you override must catch and handle checked exceptions. Here is an example of a user-defined method that specifies an exception so callers of this method can catch and handle it:

```
public int aComputationMethod(int number1,
      int number2)
      throws IllegalValueException{
   //Body of method
 }
```

Note: You can find more information on this topic in The Java Tutorial trail on Handling Errors with Exceptions.

When you catch exceptions in your code, you should handle them in a way that is friendly to your end users. The exception and error classes have a `toString` method to print system error text and a `printStackTrace` method to print a stack trace, which can be very useful for debugging your application during development. But, it is probably better to deploy the program with a more user-friendly approach to handling errors.

You can provide your own application-specific error text to print to the command line, or display a dialog box with application-specific error text. Using application-specific error text that you provide will also make it much easier to internationalize the application later on because you will have access to the text.

For the example programs in this lesson, the error message for the file input and output is handled with application-specific error text that prints at the command line as follows:

```
//Do this during development
  }catch(java.io.IOException e){
```

```
   System.out.println(e.toString());
   System.out.println(e.printStackTrace());
  }

//But deploy it like this
  }catch(java.io.IOException e){
   System.out.println("Cannot access text.txt");
  }
```

If you want to make your code even more user friendly, you could separate the write and read operations and provide two `try` and `catch` blocks. The error text for the read operation could be *Cannot read text.txt*, and the error text for the write operation could be *Cannot write text.txt*.

As an exercise, change the code to handle the read and write operations separately. Give it a try before peeking at the solution.

File Access by Applets

The file access code for the FileIOAppl.java code is equivalent to the FileIO.java application, but shows how to use the APIs for handling data in character streams instead of byte streams. You can use either approach in applets or applications. In this lesson, the choice to handle data in bytes streams in the application and in character streams in the applet is purely random. In real-life programs, you would base the decision on your specific application requirements.

The changes to instance variables and the `constructor` are identical to the application code, and the changes to the `actionPerformed` method are nearly identical with these two exceptions:

- **Writing:** When the `textField` text is retrieved, it is passed directly to the `out.write` call.
- **Reading:** A character array is created to store the data read in from the input stream.

```
public void actionPerformed(ActionEvent event){
  Object source = event.getSource();
  if(source == button){
//Variable to display text read from file
    String s = null;
```

```
    if(_clickMeMode){
      try{
//Code to write to file
        String text = textField.getText();
        String outputFileName =
                System.getProperty("user.home",
          File.separatorChar + "home" +
          File.separatorChar + "monicap") +
          File.separatorChar + "text.txt";
          File outputFile = new File(outputFileName);
          FileWriter out = new
                FileWriter(outputFile);
        out.write(text);
        out.close();
//Code to read from file
        String inputFileName =
                System.getProperty("user.home",
          File.separatorChar + "home" +
          File.separatorChar + "monicap") +
          File.separatorChar + "text.txt";
        File inputFile = new File(inputFileName);
        FileReader in = new FileReader(inputFile);
        char c[] = new
        char[(char)inputFile.length()];
        in.read(c);
        s = new String(c);
        in.close();
      }catch(java.io.IOException e){
       System.out.println("Cannot access text.txt");
      }
//Clear text field
      textField.setText("");
//Display text read from file
      text.setText("Text retrieved from file:");
      textField.setText(s);
      button.setText("Click Again");
      _clickMeMode = false;
    } else {
//Save text to file
      text.setText("Text to save to file:");
      textField.setText("");
      button.setText("Click Me");
      _clickMeMode = true;
    }
  }
}
```

Granting Applets Permission

If you tried to run the applet example, you undoubtedly saw errors when you clicked the `Click Me` button. This is because the Java 2 Platform security does not permit an applet to write to and read from files without explicit permission.

An applet has no access to local system resources unless it is specifically granted the access. So for the `FileUIAppl` program to read from `text.txt` and write to `text.txt`, the applet has to be given the appropriate read or write access permission for each file.

Access permission is granted with a policy file, and appletviewer is launched with the policy file to be used for the applet being viewed.

Creating a Policy File

Policy tool is a Java 2 Platform security tool for creating policy files. The Java Tutorial trail on Controlling Applets explains how to use Policy Tool in good detail. Here is the policy file you need to run the applet. You can use Policy tool to create it or copy the text below into an ASCII file.

```
grant {
  permission java.util.PropertyPermission
        "user.home", "read";
  permission java.io.FilePermission
        "${user.home}/text.txt", "read,write";
};
```

Running an Applet with a Policy File

Assuming the policy file is named `polfile` and is in the same directory with an HTML file named `fileIO.html` that contains the HTML to run the `FileIOAppl` applet, you would run the application in appletviewer like this:

```
appletviewer -J-Djava.security.policy=polfile fileIO.html
```

Note: If your browser is enabled for the Java 2 Platform or if you have Java Plug-in installed, you can run the applet from the browser if you put the policy

file in your local home directory.

Here is the `fileIO.html` file for running the `FileIOAppl` applet:

```
<HTML>
<BODY>

<APPLET CODE=FileIOAppl.class WIDTH=200 HEIGHT=100>
</APPLET>

</BODY>
</HTML>
```

Restricting Applications

You can use the default security manager and a policy file to restrict the application's access as follows.

```
java -Djava.security.manager
     -Djava.security.policy=apppolfile  FileIO
```

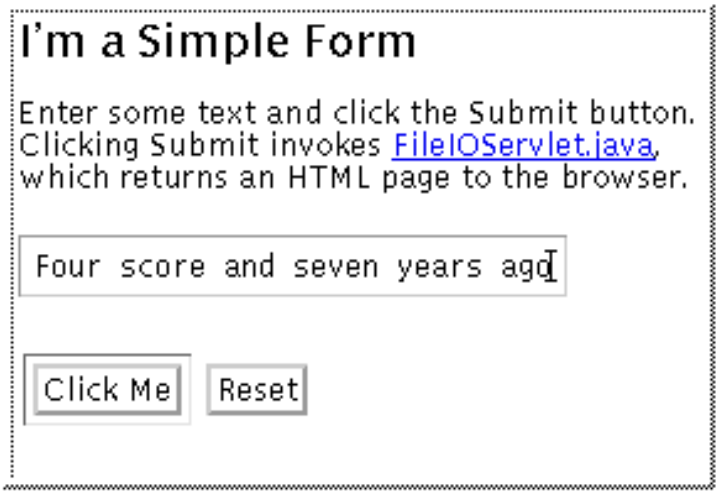
Because the application runs within the security manager, which disallows all access, the policy file needs two additional permissions. One so the security manager can access the event queue and load the user interface components, and another so the application does not display the banner warning that its window was created by another program (the security manager).

```
grant {
  permission java.awt.AWTPermission
        "accessEventQueue";
  permission java.awt.AWTPermission
        "showWindowWithoutWarningBanner";

  permission java.util.PropertyPermission
        "user.home", "read";
  permission java.io.FilePermission
        "${user.home}/text.txt", "read,write";
};
```

File Access by Servlets

Although servlets are invoked from a browser, they are under the security policy in force for the web server under which they run. When file input and output code is added to `ExampServlet.java`

from Lesson 5, FileIOServlet for this lesson executes without restriction under Java WebServer™ 1.1.1.




```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class FileIOServlet extends HttpServlet {

  public void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
  {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<title>Example<title>" +
                  "<body bgcolor=FFFFFF>");

    out.println("<h2>Button Clicked</h2>");

    String DATA = request.getParameter("DATA");

    if(DATA != null){
      out.println("<STRONG>Text from
```

```
                                form:</STRONG>");
        out.println(DATA);
      } else {
        out.println("No text entered.");
      }

      try{
//Code to write to file
        String outputFileName=
          System.getProperty("user.home",
            File.separatorChar + "home" +
            File.separatorChar + "monicap") +
            File.separatorChar + "text.txt";
        File outputFile = new File(outputFileName);
        FileWriter fout = new FileWriter(outputFile);
        fout.write(DATA);
        fout.close();

//Code to read from file
        String inputFileName =
          System.getProperty("user.home",
            File.separatorChar + "home" +
            File.separatorChar + "monicap") +
            File.separatorChar + "text.txt";
        File inputFile = new File(inputFileName);
        FileReader fin = new
          FileReader(inputFile);
        char c[] = new
          char[(char)inputFile.length()];
        int i;
        i = fin.read(c);
        String s = new String(c);
        out.println("<P>
          <STRONG>Text from file:</STRONG>");
        out.println(s);
        fin.close();
      }catch(java.io.IOException e){
       System.out.println("Cannot access text.txt");
      }

      out.println("<P>Return to
          <A HREF="../simpleHTML.html">Form</A>");
      out.close();
    }
}
```

Appending

So far the examples have shown you how to read in and write out streams of data in their entirety. But often, you want to append data to an existing file or read in only certain amounts. Using the RandomAccessFile class, alter the FileIO.java class to append to the file.

Give it a try before taking a peek at the Solution.

More Information

For more infomation on file input and output, see the Reading and Writing trail in The Java Tutorial.

You can learn more about component sizing in The Java Tutorial sections on Solving Common Layout Problems and Layout Management.

[TOP]

Java™ Programming Language Basics, Part 1

Lesson 7: Database Access and Permissions

[<<BACK] [CONTENTS] [NEXT>>]

This lesson converts the application, applet, and servlet examples from Lesson 6 to write to and read from a database using JDBC™. JDBC is the Java™ database connectivity application programming interface (API) available in the Java® 2 Platform software.

The code for this lesson is very similar to the code you saw in Lesson 6, but additional steps (beyond converting the file access code to database access code) include setting up the environment, creating a database table, and connecting to the database. Creating a database table is a database administration task that is not part of your program code. However, establishing a database connection and the resulting database access are.

As in Lesson 6, the applet needs appropriate permissions to connect to the database. Which permissions it needs varies with the type of driver used to make the database connection.

Database Setup

You need access to a database if you want to run the examples in this lesson. You can install a database on your machine or perhaps you have access to a database at work. Either way, you need a database driver and any relevant environment settings so your program can load the driver and locate the database. The program will also need database login information in the form of a user name and password.

A database driver is software that lets a program establish a connection with a database. If you do not have the right driver for the database to which you want to connect, your program will be unable to establish the connection.

Drivers either come with the database or are available from the Web. If you install your own database, consult the documentation for the driver for information on installation and any other environment settings you need for your platform. If you are using a database at work, consult your database administrator for this information.

To show you two ways to do it, the application example uses the `jdbc` driver, the applet examples use the `jdbc` and `jdbc.odbc` drivers, and the servlet example uses the `jdbc.odbc` driver. All examples connect to an `OracleOCI7.3.4` database.

Connections to other databases will involve similar steps and code. Be sure to consult your documentation or system administrator if you need help connecting to the database.

Create Database Table

Once you have access to a database, create a table in it for the examples in this lesson. You need a table with one text field for storing character data.

```
TABLE DBA (
        TEXT            varchar2(100),
        primary key (TEXT)
)
```

Database Access by Applications

This example converts the FileIO program from Lesson 6 to write data to and read data from a database. The top window below appears when you start the Dba application, and the window beneath it appears when you click the `Click Me` button.

When you click the `Click Me` button, whatever is entered into the text field is saved to the database. After that, the data is retrieved from the database and displayed in the window shown on the bottom. If you write data to the table more than once, everything written is read and displayed in the window shown on the bottom, so you might have to enlarge the window to see the entire list of table items.

When Application Starts

After Writing Orange and Apple to Database

The database access application needs code to establish the database connection and do the database read and write operations.

Establishing a Database Connection

The JDBC `DriverManager` class can handle multiple database drivers, and initiates all database communication. To load the driver and connect to the database, the application needs a

Connection object and Strings that represent the _driver and _url.

The _url string is in the form of a Uniform Resource Locator (URL). It consists of the URL, Oracle subprotcol, and Oracle data source in the form jdbc:oracle:thin, the database login username, the password, plus machine, port, and protocol information.

```
private Connection c;

final static private String _driver =
  "oracle.jdbc.driver.OracleDriver";

final static private String _url =
 "jdbc:oracle:thin:username/password@(description=(
 address_list=(address=(protocol=tcp)
 (host=developer)(port=1521)))
 (source_route=yes)(connect_data=(sid=jdcsid)))";
```

The actionPerformed method calls the Class.forName (_driver) method to load the driver, and the DriverManager.getConnection method to establish the connection. The Exception Handling section in Lesson 6 describes try and catch blocks. The only thing different here is that this block uses two catch statements because two different errors are possible.

The call to Class.forName(_driver); throws java.lang.ClassNotFoundException, and the call to c = DriverManager.getConnection(_url); throws java.sql.SQLException. In the case of either error, the application tells the user what is wrong and exits because the program cannot operate in any meaningful way without a database driver or connection.

```
public void actionPerformed(ActionEvent event){
  try{
//Load the driver
     Class.forName(_driver);
//Establish database connection
     c = DriverManager.getConnection(_url);
   }catch (java.lang.ClassNotFoundException e){
     System.out.println("Cannot find driver class");
     System.exit(1);
   }catch (java.sql.SQLException e){
```

```
      System.out.println("Cannot get connection");
      System.exit(1);
    }
```

Final and Private Variables

The member variables used to establish the database connection above are declared `private`, and two of those variables are also declared `final`.

final: A `final` variable contains a constant value that can never change once it is initialized. In the example, the user name, and password are `final` variables because you would not want to allow an instance of this or any other class to change this information.

private: A `private` variable can only be used (accessed) by the class in which it is declared. No other class can read or change `private` variables. In the example, the database driver, user name, and password variables are `private` to prevent an outside class from accessing them and jeopardizing the database connection, or compromising the secret user name and password information. You can find more information on this in the Objects and Classs lesson in The Java Tutorial

Writing and Reading Data

In the write operation, a `Statement` object is created from the `Connection`. The `Statement` object has methods for executing SQL queries and updates. Next, a `String` object that contains the SQL update for the write operation is constructed and passed to the `executeUpdate` method of the `Statement` object.

```
Object source = event.getSource();
if(source == button){
 JTextArea displayText = new JTextArea();

 try{
//Code to write to database
   String theText = textField.getText();
   Statement stmt = c.createStatement();
   String updateString = "INSERT INTO dba VALUES
                          ('" + theText + "')";
   int count = stmt.executeUpdate(updateString);
```

SQL commands are `String` objects, and therefore, follow the rules of `String` construction where the string is enclosed in double quotes (" ") and variable data is appended with a plus (+). The variable `theText` has single and double quotes to tell the database the SQL string has variable rather than literal data.

In the read operation, a `ResultSet` object is created from the `executeQuery` method of the `Statement` object. The `ResultSet` contains the data returned by the query. To retrieve the data returned, the code iterates through the `ResultSet`, retrieves the data, and appends the data to the text area, `displayText`.

```
//Code to read from database
    ResultSet results = stmt.executeQuery(
        "SELECT TEXT FROM dba ");
    while(results.next()){
      String s = results.getString("TEXT");
      displayText.append(s + "\n");
    }
    stmt.close();
  } catch(java.sql.SQLException e){
    System.out.println(e.toString());
  }
//Display text read from database
  panel.removeAll();
  panel.add("North", clicked);
  panel.add("Center", displayText);
  panel.add("South", clickButton);
  panel.validate();
  panel.repaint();
}
```

Database Access by Applets

The applet version of the example is like the application code described above except for the standard differences between applications and applets described in the Structure and Elements section of Lesson 3.

However, if you run the applet without a policy file, you get a stack trace indicating permission errors. The Granting Applets Permission section in Lesson 6 introduced you to policy files and how to launch an applet with the permission it needs. The Lesson 6 applet example provided the policy file and told you how to launch the applet with it. This lesson shows you how to read the

stack trace to determine the permissions you need in a policy file.

To keep things interesting, this lesson has two versions of the database access applet: one uses the JDBC driver, and the other uses the the JDBC-ODBC bridge with an Open DataBase Connectivity (ODBC) driver.

Both applets do the same operations to the same database table using different drivers. Each applet has its own policy file with different permission lists and has different requirements for locating the database driver

JDBC Driver

The JDBC driver is used from a program written exclusively in the Java language (Java program). It converts JDBC calls directly into the protocol used by the DBMS. This type of driver is available from the DBMS vendor and is usually packaged with the DBMS software.

Starting the Applet: To successfully run, the DbaAppl.java applet needs an available database driver and a policy file. This section walks through the steps to get everything set up. Here is the `DbaAppl.html` file for running the `DbaAppl` applet:

```
<HTML>
<BODY>

<APPLET CODE=DbaAppl.class
  WIDTH=200
  HEIGHT=100>
</APPLET>

</BODY>
</HTML>
```

And here is how to start the applet with appletviewer:

```
  appletviewer DbaAppl.html
```

Locating the Database Driver: Assuming the driver is not available to the `DriverManager` for some reason, the following error generates when you click the `Click Me` button.

```
  cannot find driver
```

This error means the DriverManager looked for the JDBC driver in the directory where the applet HTML and class files are and could not find it. To correct this error, copy the driver to the directory where the applet files are, and if the driver is bundled in a zip file, unzip the zip file so the applet can access the driver.

Once you have the driver in place, launch the applet again.

```
appletviewer DbaAppl.html
```

Reading a Stack Trace: Assuming the driver is locally available to the applet, if the DbaAppl.java applet is launched without a policy file, the following stack trace is generated when the end user clicks the `Click Me` button.

```
java.security.AccessControlException: access denied
(java.net.SocketPermission developer resolve)
```

The first line in the above stack trace tells you access is denied. This means this stack trace was generated because the applet tried to access a system resource without the proper permission. The second line means to correct this condition you need a `SocketPermission` that gives the applet access to the machine (`developer`) where the database is located.

You can use Policy tool to create the policy file you need, or you can create it with an ASCII editor. Here is the policy file with the permission indicated by the stack trace:

```
grant {
  permission java.net.SocketPermission "developer",
    "resolve";
  "accessClassInPackage.sun.jdbc.odbc";
};
```

Run the applet again, this time with a policy file named `DbaApplPol` that has the above permission in it:

```
appletviewer -J-Djava.security.policy=DbaApplPol
                                      DbaAppl.html
```

You get a stack trace again, but this time it is a different error condition.

```
java.security.AccessControlException: access denied
```

```
  (java.net.SocketPermission
  129.144.176.176:1521 connect,resolve)
```

Now you need a `SocketPermission` that allows access to the Internet Protocol (IP) address and port on the `developer` machine where the database is located.

Here is the `DbaApplPol` policy file with the permission indicated by the stack trace added to it:

```
grant {
  permission java.net.SocketPermission "developer",
                        "resolve";
  permission java.net.SocketPermission
  "129.144.176.176:1521", "connect,resolve";
};
```

Run the applet again. If you use the above policy file with the Socket permissions indicated, it works just fine.

```
  appletviewer -J-Djava.security.policy=DbaApplPol
                                        DbaAppl.html
```

JDBC-ODBC Bridge with ODBC Driver

Open DataBase Connectivity (ODBC) is Microsoft's programming interface for accessing a large number of relational databases on numerous platforms. The JDBC-ODBC bridge is built into the Solaris and Windows versions of the Java platform so you can do two things:

1. Use ODBC from a Java program
2. Load ODBC drivers as JDBC drivers. This example uses the JDBC-ODBC bridge to load an ODBC driver to connect to the database. The applet has no ODBC code, however.

The `DriverManager` uses environment settings to locate and load the database driver. For this example, the driver file does not need to be locally accessible.

Start the Applet: Here is the `DbaOdb.html` file for running the `DbaOdbAppl` applet:

```
<HTML>
<BODY>

<APPLET CODE=DbaOdbAppl.class
  WIDTH=200
  HEIGHT=100>
</APPLET>

</BODY>
</HTML>
```

And here is how to start the applet:

```
  appletviewer DbaOdb.html
```

Reading a Stack Trace: If the DbaOdbAppl.java applet is launched without a policy file, the following stack trace is generated when the end user clicks the `Click Me` button.

```
  java.security.AccessControlException: access denied
  (java.lang.RuntimePermission
  accessClassInPackage.sun.jdbc.odbc )
```

The first line in the above stack trace tells you access is denied. This means this stack trace was generated because the applet tried to access a system resource without the proper permission. The second line means you need a `RuntimePermission` that gives the applet access to the `sun.jdbc.odbc` package. This package provides the JDBC-ODBC bridge functionality to the Java[1] virtual machine (VM).

You can use Policy tool to create the policy file you need, or you can create it with an ASCII editor. Here is the policy file with the permission indicated by the stack trace:

```
grant {
  permission java.lang.RuntimePermission
    "accessClassInPackage.sun.jdbc.odbc";
};
```

Run the applet again, this time with a policy file named `DbaOdbPol` that has the above permission in it:

```
  appletviewer -J-Djava.security.policy=DbaOdbPol
                                        DbaOdb.html
```

You get a stack trace again, but this time it is a different error condition.

```
java.security.AccessControlException:
access denied (java.lang.RuntimePermission
file.encoding read)
```

The stack trace means the applet needs read permission to the encoded (binary) file. Here is the `DbaOdbPol` policy file with the permission indicated by the stack trace added to it:

```
grant {
  permission java.lang.RuntimePermission
     "accessClassInPackage.sun.jdbc.odbc";
  permission java.util.PropertyPermission
     "file.encoding", "read";
};
```

Run the applet again. If you use the above policy file with the Runtime and Property permissions indicated, it works just fine.

```
appletviewer -J-Djava.security.policy=DbaOdbPol
                                        DbaOdb.html
```

Database Access by Servlets

As you learned in Lesson 6, servlets are under the security policy in force for the web server under which they run. When the database read and write code is added to the `FileIOServlet` from Lesson 6, the DbaServlet.java servlet for this lesson executes without restriction under Java WebServer™ 1.1.1.

The web server has to be configured to locate the database. Consult your web server documentation or database administrator for help. With Java WebServer 1.1.1, the configuration setup involves editing the startup scripts with such things as environment settings for loading the ODBC driver, and locating and connecting to the database.


```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

import java.sql.*;
import java.net.*;
import java.io.*;

public class DbaServlet extends HttpServlet {

 private Connection c;
 final static private String _driver =
        "sun.jdbc.odbc.JdbcOdbcDriver";
 final static private String _user = "username";
 final static private String _pass = "password";
 final static private String
                        _url = "jdbc:odbc:jdc";

public void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException{
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  out.println("<title>Example<title>" +
      "<body bgcolor=FFFFFF>");

  out.println("<h2>Button Clicked</h2>");
```

```
  String DATA = request.getParameter("DATA");

  if(DATA != null){
     out.println("<STRONG>Text from
                          form:</STRONG>");
     out.println(DATA);
  } else {
    out.println("No text entered.");
  }

//Establish database connection
  try{
    Class.forName (_driver);
    c = DriverManager.getConnection(_url,
                                    _user,
                                    _pass);
   } catch (Exception e) {
    e.printStackTrace();
    System.exit(1);
   }

   try{
//Code to write to database
     Statement stmt = c.createStatement();
     String updateString = "INSERT INTO dba " +
                "VALUES ('" + DATA + "')";
     int count = stmt.executeUpdate(updateString);

//Code to read from database
     ResultSet results = stmt.executeQuery(
                "SELECT TEXT FROM dba ");
     while(results.next()){
       String s = results.getString("TEXT");
       out.println("<BR>
         <STRONG>Text from database:</STRONG>");
       out.println(s);
     }
      stmt.close();
     }catch(java.sql.SQLException e){
      System.out.println(e.toString());
     }

      out.println("<P>Return to
        <A HREF="../dbaHTML.html">Form</A>");
      out.close();
  }
}
```

More Information

You can find more information on variable access settings in the Objects and Classes trail in The Java Tutorial

_____

[1] As used on this web site, the terms "Java virtual machine" or "JVM" mean a virtual machine for the Java platform.

[TOP]

Print Button

[This page was updated: 31-Mar-2000]

Java™ Programming Language Basics, Part 1

Lesson 8: Remote Method Invocation

[<<BACK] [CONTENTS] [NEXT>>]

The Java™ Remote Method Invocation (RMI) application programming interface (API) enables client and server communications over the net. Typically, client programs send requests to a server program, and the server program responds to those requests.

A common example is sharing a word processing program over a network. The word processor is installed on a server, and anyone who wants to use it starts it from his or her machine by double clicking an icon on the desktop or typing at the command line. The invocation sends a request to a server program for acess to the software, and the server program responds by making the software available to the requestor.

The RMI API lets you create a publicly accessible remote server object that enables client and server communications through simple method calls on the server object. Clients can easily communicate directly with the server object and indirectly with each other through the server object using Uniform Resource Locators (URLs) and HyperText Transfer Protocol (HTTP).

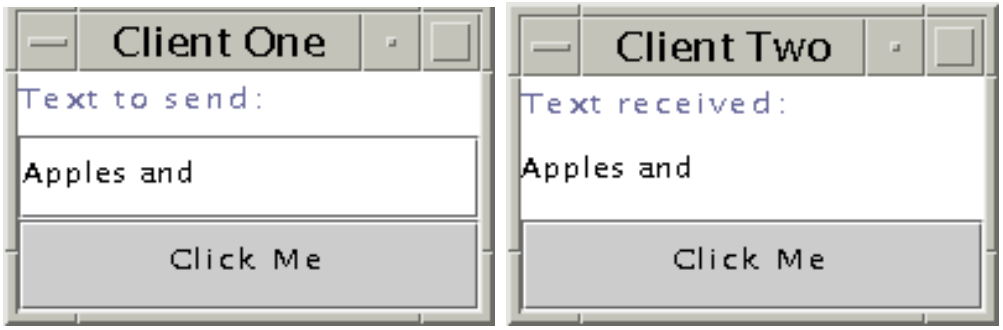
This lesson explains how to use the RMI API to establish client and server communications.

About the Example

This lesson converts the File Input and Output application from Lesson 6: File Access and Permissions to the RMI API.

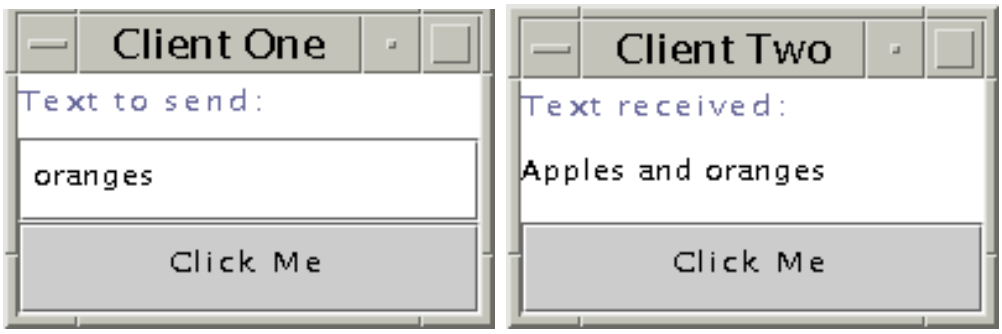
Program Behavior

The RMIClient1 program presents a simple user interface and prompts for text input. When you click the `Click Me` button, the text is sent to the RMIClient2 program by way of the remote server object. When you click the `Click Me` button on the `RMIClient2` program, the text sent from `RMIClient1` appears.



First Instance of Client 1

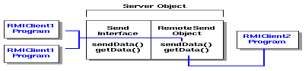
If you start a second instance of `RMIClient1` and type in some text, that text is sent to `RMIClient2` when you click the `Click Me` button. To see the text received by `RMIClient2`, click its `Click Me` button.



Second Instance of Client 1

File Summary

The example program consists of the RMIClient1 program, remote object and interface, and the RMIClient2 program as illustrated in the diagram. The corresponding source code files for these executables are described in the bullet list below.



- RMIClient1.java: Client program that calls the `sendData` method on the `RemoteServer` server object.
- RMIClient2.java: Client program that calls the `getData` method on the `RemoteServer` server object.
- RemoteServer.java: Remote server object that implements `Send.java` and the `sendData` and `getData` remote methods.
- Send.java: Remote interface that declares the `sendData` and `getData` remote server methods.

In addition, the following java.policy security policy file grants the permissions needed to run the example.

```
grant {
  permission java.net.SocketPermission
              "*:1024-65535",
              "connect,accept,resolve";
  permission java.net.SocketPermission
              "*:80", "connect";
  permission java.awt.AWTPermission
              "accessEventQueue";
  permission java.awt.AWTPermission
              "showWindowWithoutWarningBanner";
};
```

Compile the Example

These instructions assume development is in the `zelda` home directory. The server program is compiled in the home directory for user `zelda`, but copied to the `public_html` directory for user `zelda` where it runs.

Here is the command sequence for the Unix and Win32 platforms; an explanation follows.

Unix:

```
cd /home/zelda/classes
javac Send.java
javac RemoteServer.java
javac RMIClient2.java
javac RMIClient1.java
rmic -d . RemoteServer
cp RemoteServer*.class /home/zelda/public_html/classes
cp Send.class /home/zelda/public_html/classes
```

Win32:

```
cd \home\zelda\classes
javac Send.java
javac RemoteServer.java
javac RMIClient2.java
javac RMIClient1.java
rmic -d . RemoteServer
copy RemoteServer*.class \home\zelda\public_html\classes
copy Send.class \home\zelda\public_html\classes
```

The first two `javac` commands compile the `RemoteServer` and `Send` class and interface. The third `javac` command compiles the `RMIClient2` class. The last `javac` command compiles the `RMIClient1` class.

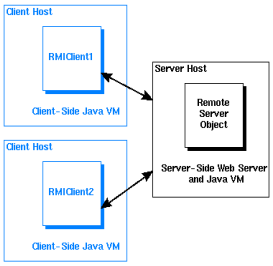
The next line runs the `rmic` command on the `RemoteServer` server class. This command produces output class files of the form `ClassName_Stub.class` and `ClassName_Skel.class`. These output classes let clients invoke methods on the `RemoteServer` server object.

The first copy command moves the `RemoteServer` class file with its associated `skel` and `stub` class files to a publicly accessible location in the `/home/zelda/public_html/classes` directory, which is on the server machine, so they can be publicly accessed and downloaded. They are placed in the `public_html` directory to be under the web server running on the server machine because these files are accessed by client programs using URLs.

The second copy command moves the `Send` class file to the same location for the same reason. The `RMIClient1` and `RMIClient2` class files are not made publicly accessible; they communicate from their client machines using URLs to access and download the remote object files in the `public_html` directory.

- `RMIClient1` is invoked from a client-side directory and uses the server-side web server and client-side Java VM to download the publicly accessible files.

- `RMIClient2` is invoked from a client-side directory and uses the server-side web server and client-side Java VM to download the publicly accessible files.



Start the RMI Registry

Before you start the client programs, you must start the RMI Registry, which is a server-side naming repository that allows remote clients to get a reference to the remote server object.

Before you start the RMI Registry, make sure the shell or window in which you run the `rmiregistry` command does not have a CLASSPATH environment variable that points to the remote object classes, including the `stub` and `skel` classes, anywhere on your system. If the RMI Registry finds these classes when it starts, it will not load them from the server-side Java VM, which will create problems when clients try to download the remote server classes.

The following commands unset the CLASSPATH and start the RMI Registry on the default 1099 port. You can specify a different port by adding the port number as follows: `rmiregistry 4444 &`. If you specify a different port number, you must specify the same port number in your server-side code as well.

Unix:

```
cd /home/zelda/public_html/classes
unsetenv CLASSPATH
rmiregistry &
```

Win32:

```
cd \home\zelda\public_html\classes
set CLASSPATH=
start rmiregistry
```

Note: You might want to set the CLASSPATH back to its original setting at this point.

Run the RemoteServer Server Object

To run the example programs, start `RemoteServer` first. If you start either `RMIClient1` or `RMIClient2` first, they will not be able to establish a connection because the remote server object is not running.

In this example, `RemoteServer` is started from the `/home/zelda/public_html/classes` directory.

The lines beginning at `java` should be all on one line with spaces where the lines break. The properties specified with the `-D` option to the `java` interpreter command are program attributes that manage the behavior of the program for this invocation.

Unix:

```
cd /home/zelda/public_html/classes
java
-Djava.rmi.server.codebase=http://kq6py/~zelda/classes
-Djava.rmi.server.hostname=kq6py.eng.sun.com
-Djava.security.policy=java.policy RemoteServer
```

Win32:

```
cd \home\zelda\public_html\classes
java -Djava.rmi.server.codebase=file:
       c:\home\zelda\public_html\classes
-Djava.rmi.server.hostname=kq6py.eng.sun.com
-Djava.security.policy=java.policy RemoteServer
```

- The `java.rmi.server.codebase` property specifies where the publicly accessible classes are located.
- The `java.rmi.server.hostname` property is the complete host name of the server where the publicly accessible classes reside.

- The `java.rmi.security.policy` property specifies the policy file with the permissions needed to run the remote server object and access the remote server classes for download.

- The class to execute (`RemoteServer`).

Run the RMIClient1 Program

Here is the command sequence for the Unix and Win32 platforms; an explanation follows.

In this example, `RMIClient1` is started from the `/home/zelda/classes` directory.

The lines beginning at `java` should be all on one line with spaces where the lines break. Properties specified with the `-D` option to the `java` interpreter command are program attributes that manage the behavior of the program for this invocation.

Unix:

```
cd /home/zelda/classes

java -Djava.rmi.server.codebase=
                          http://kq6py/~zelda/classes/
-Djava.security.policy=java.policy
                  RMIClient1 kq6py.eng.sun.com
```

Win32:

```
cd \home\zelda\classes

java -Djava.rmi.server.codebase=
                          file:c:\home\zelda\classes\
-Djava.security.policy=java.policy
                          RMIClient1 kq6py.eng.sun.com
```

- The `java.rmi.server.codebase` property specifies where the publicly accessible classes for downloading are located.

- The `java.security.policy` property specifies the policy file with the permissions needed to run the client program and access the remote server classes.

- The client program class to execute (`RMIClient1`), and the

host name of the server (Kq6py) where the remote server classes are.

Run RMIClient2

Here is the command sequence for the Unix and Win32 platforms; an explanation follows.

In this example, RMIClient2 is started from the /home/zelda/classes directory.

The lines beginning at java should be all on one line with spaces where the lines break. The properties specified with the -D option to the java interpreter command are program attributes that manage the behavior of the program for this invocation.

Unix:

```
cd /home/zelda/classes
java -Djava.rmi.server.codebase=
                          http://kq6py/~zelda/classes
-Djava.security.policy=java.policy
                          RMIClient2 kq6py.eng.sun.com
```

Win32:

```
cd \home\zelda\classes
java -Djava.rmi.server.codebase=
                file:c:\home\zelda\public_html\classes
-Djava.security.policy=java.policy
                          RMIClient2 kq6py.eng.sun.com
```

- The java.rmi.server.codebase property specifies where the publicly accessible classes are located.
- The java.rmi.server.hostname property is the complete host name of the server where the publicly accessible classes reside.
- The java.rmi.security.policy property specifies the policy file with the permissions needed to run the remote server object and access the remote server classes for download.
- The class to execute (RMIClient2).

RemoteServer Class

The RemoteServer class extends `UnicastRemoteObject` and implements the `sendData` and `getData` methods declared in the `Send` interface. These are the remotely accessible methods.

`UnicastRemoteObject` implements a number of `java.lang.Object` methods for remote objects and includes constructors and static methods to make a remote object available to receive method calls from client programs.

```
class RemoteServer extends UnicastRemoteObject
                 implements Send {

  String text;

  public RemoteServer() throws RemoteException {
    super();
  }

  public void sendData(String gotText){
    text = gotText;
  }

  public String getData(){
    return text;
  }
```

The `main` method installs the `RMISecurityManager` and opens a connection with a port on the machine where the server program runs. The security manager determines whether there is a policy file that lets downloaded code perform tasks that require permissions. The `main` method creates a name for the the `RemoteServer` object that includes the server name (`kq6py`) where the RMI Registry and remote object run, and the name, `Send`.

By default the server name uses port 1099. If you want to use a different port number, you can add it with a colon as follows: `kq6py:4444`. If you change the port here, you must start the RMI Registry with the same port number.

The `try` block creates an instance of the `RemoteServer` class and binds the name to the remote object to the RMI Registry with the `Naming.rebind(name, remoteServer);` statement.

```
  public static void main(String[] args){
    if(System.getSecurityManager() == null) {
      System.setSecurityManager(new
               RMISecurityManager());
    }
    String name = "//kq6py.eng.sun.com/Send";
    try {
      Send remoteServer = new RemoteServer();
      Naming.rebind(name, remoteServer);
      System.out.println("RemoteServer bound");
    } catch (java.rmi.RemoteException e) {
      System.out.println("Cannot create
                  remote server object");
    } catch (java.net.MalformedURLException e) {
      System.out.println("Cannot look up
                  server object");
    }
  }
}
```

Note: The `remoteServer` object is type `Send` (see instance declaration at top of class) because the interface available to clients is the `Send` interface and its methods; not the `RemoteServer` class and its methods.

Send Interface

The Send interface declares the methods implemented in the `RemoteServer` class. These are the remotely accessible methods.

```
public interface Send extends Remote {

  public void sendData(String text)
                throws RemoteException;
  public String getData() throws RemoteException;
}
```

RMIClient1 Class

The RMIClient1 class establishes a connection to the remote server program and sends data to the remote server object. The code to do these things is in the `actionPerformed` and `main`

methods.

actionPerformed Method

The `actionPerformed` method calls the `RemoteServer.sendData` method to send text to the remote server object.

```
public void actionPerformed(ActionEvent event){
   Object source = event.getSource();

   if(source == button){
//Send data over socket
      String text = textField.getText();
      try{
        send.sendData(text);
      } catch (java.rmi.RemoteException e) {
        System.out.println("Cannot send data to server");
      }
      textField.setText(new String(""));
   }
}
```

main Method

The `main` method installs the `RMISecurityManager` and creates a name to use to look up the `RemoteServer` server object. The client uses the `Naming.lookup` method to look up the `RemoteServer` object in the RMI Registry running on the server.

The security manager determines whether there is a policy file that lets downloaded code perform tasks that require permissions.

```
  RMIClient1 frame = new RMIClient1();

  if(System.getSecurityManager() == null) {
    System.setSecurityManager(new RMISecurityManager());
  }

  try {
//args[0] contains name of server where Send runs
    String name = "//" + args[0] + "/Send";
    send = ((Send) Naming.lookup(name));
  } catch (java.rmi.NotBoundException e) {
    System.out.println("Cannot look up
                  remote server object");
```

```
  } catch(java.rmi.RemoteException e){
    System.out.println("Cannot look up
                remote server object");
  } catch(java.net.MalformedURLException e) {
    System.out.println("Cannot look up
                remote server object");
  }
```

RMIClient2 Class

The RMIClient2 class establishes a connection with the remote server program and gets the data from the remote server object and displays it. The code to do this is in the `actionPerformed` and `main` methods.

actionPerformed Method

The `actionPerformed` method calls the `RemoteServer.getData` method to retrieve the data sent by the client program. This data is appended to the `TextArea` object for display to the end user on the server side.

```
public void actionPerformed(ActionEvent event) {
   Object source = event.getSource();

   if(source == button){
      try{
        String text = send.getData();
        textArea.append(text);
      } catch (java.rmi.RemoteException e) {
        System.out.println("Cannot send data
                    to server");
      }
      }
   }
}
```

main Method

The `main` method installs the `RMISecurityManager` and creates a name to use to look up the `RemoteServer` server object. The `args[0]` parameter provides the name of the server host. The client uses the `Naming.lookup` method to look up the `RemoteServer` object in the RMI Registry running on the server.

The security manager determines whether there is a policy file that lets downloaded code perform tasks that require permissions.

```
RMIClient2 frame = new RMIClient2();

if(System.getSecurityManager() == null) {
  System.setSecurityManager(new RMISecurityManager());
}

try {
  String name = "//" + args[0] + "/Send";
  send = ((Send) Naming.lookup(name));
} catch (java.rmi.NotBoundException e) {
  System.out.println("Cannot look up remote
                server object");
} catch(java.rmi.RemoteException e){
  System.out.println("Cannot look up remote
                server object");
} catch(java.net.MalformedURLException e) {
  System.out.println("Cannot look up remote
                server object");
}
```

More Information

You can find more information on the RMI API in the RMI trail of The Java Tutorial.

[TOP]

Print Button

Java™ Programming Language Basics, Part 1

In Closing

[<<BACK] [CONTENTS]

After completing this tutorial you should have a basic understanding of Java™ programming and how to use some of the more common application programming interfaces (APIs) available in the Java platform. You should also have a solid understanding of the similarities and differences between the three most common kinds of Java programs: applications, applets, and servlets.

Java Programming Language Basics, Part 2, is now available. It covers sockets, threads, cryptography, building a more complex user interface, serialization, collections, internationalization, and Java Archive (JAR) files. It also presents object-oriented concepts as they relate to the examples in Part 1 and Part 2.

You can also explore programming in the Java language on your own with the help of the articles, training materials, other documents available on the Docs & Training page.

Monica Pawlan is a staff writer on the JDC team. She has a background in 2D and 3D graphics, security, database products, and loves to explore emerging technologies. monica.pawlan@eng.sun.com

For more information on Java technology
and other software from Sun Microsystems, call:
(800) 786-7638
Outside the U.S. and Canada, dial your country's
AT&T Direct Access Number first.

```
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;

class FileIO extends JFrame implements ActionListener {
  JLabel text;
  JButton button;
  JPanel panel;
  JTextField textField;
  private boolean _clickMeMode = true;

  FileIO() { //Begin Constructor
    text = new JLabel("Text to save to file:");
    button = new JButton("Click Me");
    button.addActionListener(this);
    textField = new JTextField(30);
    panel = new JPanel();
    panel.setLayout(new BorderLayout());
    panel.setBackground(Color.white);
    getContentPane().add(panel);
    panel.add(BorderLayout.NORTH, text);
    panel.add(BorderLayout.CENTER, textField);
    panel.add(BorderLayout.SOUTH, button);
  } //End Constructor

  public void actionPerformed(ActionEvent event){
    Object source = event.getSource();
//The equals operator (==) is one of the few operators
//allowed on an object in the Java programming language
    if (source == button) {
      String s = null;
      //Write to file
      if (_clickMeMode){
        try {
         String text = textField.getText();
         byte b[] = text.getBytes();
         String outputFileName = System.getProperty("user.home",
                          File.separatorChar + "home" +
                          File.separatorChar + "zelda") +
                          File.separatorChar + "text.txt";
```

```
        FileOutputStream out = new FileOutputStream(outputFileName);
        out.write(b);
        out.close();
       } catch(java.io.IOException e) {
        System.out.println("Cannot write to text.txt");
       }
     //Read from file
        try {
        String inputFileName = System.getProperty("user.home",
                    File.separatorChar + "home" +
                    File.separatorChar + "zelda") +
                    File.separatorChar + "text.txt";
         File inputFile = new File(inputFileName);
         FileInputStream in = new FileInputStream(inputFile);
         byte bt[] = new byte[(int)inputFile.length()];
         in.read(bt);
         s = new String(bt);
         in.close();
        } catch(java.io.IOException e) {
          System.out.println("Cannot read from text.txt");
        }
       //Clear text field
         textField.setText("");
       //Display text read from file
         text.setText("Text retrieved from file:");
         textField.setText(s);
         button.setText("Click Again");
         _clickMeMode = false;
       } else {
//Save text to file
         text.setText("Text to save to file:");
         textField.setText("");
         button.setText("Click Me");
         _clickMeMode = true;
       }
   }
 }

 public static void main(String[] args){
   FileIO frame = new FileIO();
   frame.setTitle("Example");
   WindowListener l = new WindowAdapter() {
     public void windowClosing(WindowEvent e) {
```

```
      System.exit(0);
    }
   };
   frame.addWindowListener(l);
   frame.pack();
   frame.setVisible(true);
 }
}
```

```
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;

import java.io.*;
import java.net.*;

import java.rmi.*;
import java.rmi.server.*;

class RMIClient1 extends JFrame
              implements ActionListener {

   JLabel text, clicked;
   JButton button;
   JPanel panel;
   JTextField textField;
   Socket socket = null;
   PrintWriter out = null;
   static Send send;

   RMIClient1(){ //Begin Constructor
     text = new JLabel("Text to send:");
     textField = new JTextField(20);
     button = new JButton("Click Me");
     button.addActionListener(this);

     panel = new JPanel();
     panel.setLayout(new BorderLayout());
     panel.setBackground(Color.white);
     getContentPane().add(panel);
     panel.add("North", text);
     panel.add("Center", textField);
     panel.add("South", button);
   } //End Constructor

  public void actionPerformed(ActionEvent event){
     Object source = event.getSource();

     if(source == button){
//Send data over socket
```

```
      String text = textField.getText();
      try{
        send.sendData(text);
      } catch (java.rmi.RemoteException e) {
        System.out.println("Cannot send data to server");
      }
      textField.setText(new String(""));
    }
  }

  public static void main(String[] args){
      RMIClient1 frame = new RMIClient1();
      frame.setTitle("Client One");
      WindowListener l = new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                  System.exit(0);
            }
      };

      frame.addWindowListener(l);
      frame.pack();
      frame.setVisible(true);

   if(System.getSecurityManager() == null) {
     System.setSecurityManager(new RMISecurityManager());
   }

   try {
     String name = "//" + args[0] + "/Send";
     send = ((Send) Naming.lookup(name));
   } catch (java.rmi.NotBoundException e) {
     System.out.println("Cannot look up remote server object");
   } catch(java.rmi.RemoteException e){
     System.out.println("Cannot look up remote server object");
   } catch(java.net.MalformedURLException e) {
     System.out.println("Cannot look up remote server object");
   }
  }
}
```

```
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;

import java.io.*;
import java.net.*;

import java.rmi.*;
import java.rmi.server.*;

class RMIClient2 extends JFrame
             implements ActionListener {

  JLabel text, clicked;
  JButton button;
  JPanel panel;
  JTextArea textArea;
  Socket socket = null;
  PrintWriter out = null;
  static Send send;

  RMIClient2(){ //Begin Constructor
    text = new JLabel("Text received:");
    textArea = new JTextArea();
    button = new JButton("Click Me");
    button.addActionListener(this);

    panel = new JPanel();
    panel.setLayout(new BorderLayout());
    panel.setBackground(Color.white);
    getContentPane().add(panel);
    panel.add("North", text);
    panel.add("Center", textArea);
    panel.add("South", button);
  } //End Constructor

 public void actionPerformed(ActionEvent event){
    Object source = event.getSource();

    if(source == button){
      try{
```

```
        String text = send.getData();
        textArea.append(text);
      } catch (java.rmi.RemoteException e) {
        System.out.println("Cannot access data in server");
      }
    }
  }

  public static void main(String[] args){
      RMIClient2 frame = new RMIClient2();
      frame.setTitle("Client Two");
      WindowListener l = new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                  System.exit(0);
            }
      };

      frame.addWindowListener(l);
      frame.pack();
      frame.setVisible(true);

   if(System.getSecurityManager() == null) {
     System.setSecurityManager(new RMISecurityManager());
   }

   try {
     String name = "//" + args[0] + "/Send";
     send = ((Send) Naming.lookup(name));
   } catch (java.rmi.NotBoundException e) {
     System.out.println("Cannot access data in server");
   } catch(java.rmi.RemoteException e){
     System.out.println("Cannot access data in server");
   } catch(java.net.MalformedURLException e) {
     System.out.println("Cannot access data in server");
   }
  }
}
```

```
import java.awt.Font;
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;

import java.io.*;
import java.net.*;

import java.rmi.*;
import java.rmi.server.*;

class RemoteServer extends UnicastRemoteObject
                implements Send {

  private String text;

  public RemoteServer() throws RemoteException {
    super();
  }

  public void sendData(String gotText){
    text = gotText;
  }

  public String getData(){
    return text;
  }

  public static void main(String[] args){
    if(System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }

    String name = "//kq6py.eng.sun.com/Send";
    try {
      Send remoteServer = new RemoteServer();
      Naming.rebind(name, remoteServer);
      System.out.println("RemoteServer bound");
    } catch (java.rmi.RemoteException e) {
      System.out.println("Cannot create remote server object");
    } catch (java.net.MalformedURLException e) {
```

```
      System.out.println("Cannot look up server object");
    }
  }
}
```

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Send extends Remote {

  public void sendData(String text) throws RemoteException;
  public String getData() throws RemoteException;
}
```

```
grant {
    permission java.net.SocketPermission "*:1024-65535", "connect,accept,resolve";
    permission java.net.SocketPermission "*:80", "connect";
    permission java.awt.AWTPermission "accessEventQueue";
    permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
  permission java.util.PropertyPermission "user.home", "read";
  permission java.io.FilePermission "${user.home}/text.txt", "write";
  permission java.io.FilePermission "${user.home}/text2.txt", "read";

};
```

```
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;
import java.net.*;
import java.util.*;
import java.io.*;

class Dba extends JFrame implements ActionListener {

   JLabel text, clicked;
   JButton button, clickButton;
   JPanel panel;
   JTextField textField;
   private boolean _clickMeMode = true;

   private Connection c;

   final static private String _driver = "oracle.jdbc.driver.OracleDriver";
   final static private String _url = "jdbc:oracle:thin:username/password@(description=(address_list=
(address=(protocol=tcp)(host=developer)(port=1521)))(source_route=yes)(connect_data=(sid=ansid)))";


   Dba(){ //Begin Constructor
     text = new JLabel("Text to save to database:");
     button = new JButton("Click Me");
     button.addActionListener(this);
     textField = new JTextField(20);
     panel = new JPanel();
     panel.setLayout(new BorderLayout());
     panel.setBackground(Color.white);
     getContentPane().add(panel);
     panel.add(BorderLayout.NORTH, text);
     panel.add(BorderLayout.CENTER, textField);
     panel.add(BorderLayout.SOUTH, button);
   } //End Constructor

   public void actionPerformed(ActionEvent event){
     try{
// Load the Driver
        Class.forName (_driver);
```

```
// Make Connection
       c = DriverManager.getConnection(_url);
      }
      catch (java.lang.ClassNotFoundException e){
       System.out.println("Cannot find driver class");
       System.exit(1);
      }catch (java.sql.SQLException e){
       System.out.println("Cannot get connection");
       System.exit(1);
      }

     Object source = event.getSource();
     if(source == button){
       if(_clickMeMode){
         JTextArea displayText = new JTextArea();
         try{
      //Code to write to database
           String theText = textField.getText();
           Statement stmt = c.createStatement();
           String updateString = "INSERT INTO dba VALUES ('" + theText + "')";
         int count = stmt.executeUpdate(updateString);
     //Code to read from database
         ResultSet results = stmt.executeQuery("SELECT TEXT FROM dba ");
         while(results.next()){
           String s = results.getString("TEXT");
           displayText.append(s + "\n");
         }
         stmt.close();
       }catch(java.sql.SQLException e){
        System.out.println("Cannot create SQL statement");
       }

      //Display text read from database
       text.setText("Text retrieved from database:");
       button.setText("Click Again");
       _clickMeMode = false;
//Display text read from database
      } else {
       text.setText("Text to save to database:");
       textField.setText("");
       button.setText("Click Me");
       _clickMeMode = true;
      }
```

```
  }
}

  public static void main(String[] args){
    Dba frame = new Dba();
    frame.setTitle("Example");
    WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    frame.addWindowListener(l);
    frame.pack();
    frame.setVisible(true);
  }
}
```

```
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.*;
import java.applet.Applet;
import javax.swing.*;
import java.sql.*;
import java.net.*;
import java.io.*;

public class DbaAppl extends Applet implements ActionListener {

   JLabel text, clicked;
   JButton button, clickButton;
   JTextField textField;
   private boolean _clickMeMode = true;
   private Connection c;

   final static private String _driver = "oracle.jdbc.driver.OracleDriver";
   final static private String _url = "jdbc:oracle:thin:username/password@(description=(address_list=
(address=(protocol=tcp)(host=developer)(port=1521)))(source_route=yes)(connect_data=(sid=ansid)))";

   public void init(){
     setBackground(Color.white);
     text = new JLabel("Text to save to file:");
     clicked = new JLabel("Text retrieved from file:");
     button = new JButton("Click Me");
     button.addActionListener(this);
     clickButton = new JButton("Click Again");
     clickButton.addActionListener(this);
     textField = new JTextField(20);
     setLayout(new BorderLayout());
     setBackground(Color.white);
     add(BorderLayout.NORTH, text);
     add(BorderLayout.CENTER, textField);
     add(BorderLayout.SOUTH, button);
   }

   public void start(){
     System.out.println("Applet starting.");
   }

   public void stop(){
```

```
    System.out.println("Applet stopping.");
  }

  public void destroy(){
   System.out.println("Destroy method called.");
  }

  public void actionPerformed(ActionEvent event){
    try{
       Class.forName (_driver);
       c = DriverManager.getConnection(_url);
     }catch (java.lang.ClassNotFoundException e){
      System.out.println("Cannot find driver");
      System.exit(1);
     }catch (java.sql.SQLException e){
      System.out.println("Cannot get connection");
      System.exit(1);
     }

    Object source = event.getSource();
    if(source == button){
      if(_clickMeMode){
        JTextArea displayText = new JTextArea();
        try{
    //Write to database
         String theText = textField.getText();
         Statement stmt = c.createStatement();
         String updateString = "INSERT INTO dba VALUES ('" + theText + "')";
         int count = stmt.executeUpdate(updateString);
    //Read from database
         ResultSet results = stmt.executeQuery("SELECT TEXT FROM dba ");
         while(results.next()){
           String s = results.getString("TEXT");
           displayText.append(s + "\n");
         }
         stmt.close();
        }catch(java.sql.SQLException e){
        System.out.println("Cannot create SQL statement");
        System.exit(1);
        }

//Display text read from database
       text.setText("Text retrieved from file:");
```

```
        button.setText("Click Again");
        _clickMeMode = false;
//Display text read from database
      } else {
        text.setText("Text to save to file:");
        textField.setText("");
        button.setText("Click Me");
        _clickMeMode = true;
      }
    }
  }
}
```

```
import java.awt.Font;
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.*;
import java.applet.Applet;
import javax.swing.*;

import java.sql.*;
import java.net.*;
import java.io.*;

public class DbaOdbAppl extends Applet
                implements ActionListener {

   JLabel text, clicked;
   JButton button, clickButton;
   JTextField textField;
   private boolean _clickMeMode = true;
   private Connection c;
   final static private String _driver = "sun.jdbc.odbc.JdbcOdbcDriver";
   final static private String _user = "username";
   final static private String _pass = "password";
   final static private String _url = "jdbc:odbc:jdc";
   public void init(){
     text = new JLabel("Text to save to file:");
     clicked = new JLabel("Text retrieved from file:");
     button = new JButton("Click Me");
     button.addActionListener(this);
     clickButton = new JButton("Click Again");
     clickButton.addActionListener(this);
     textField = new JTextField(20);
     setLayout(new BorderLayout());
     setBackground(Color.white);
     add(BorderLayout.NORTH, text);
     add(BorderLayout.CENTER, textField);
     add(BorderLayout.SOUTH, button);
   }

   public void start(){
   }

  public void stop(){
```

```
    System.out.println("Applet stopping.");
  }

  public void destroy(){
    System.out.println("Destroy method called.");
  }

  public void actionPerformed(ActionEvent event){
    try{
     Class.forName (_driver);
     c = DriverManager.getConnection(_url, _user, _pass);
    }catch (Exception e){
     e.printStackTrace();
     System.exit(1);
    }

    Object source = event.getSource();
    if(source == button){
      if(_clickMeMode){
        JTextArea displayText = new JTextArea();
        try{
    //Write to database
         String theText = textField.getText();
         Statement stmt = c.createStatement();
         String updateString = "INSERT INTO dba VALUES ('" + theText + "')";
         int count = stmt.executeUpdate(updateString);
   //Read from database
         ResultSet results = stmt.executeQuery("SELECT TEXT FROM dba ");
         while(results.next()){
           String s = results.getString("TEXT");
           displayText.append(s + "\n");
         }
         stmt.close();
        }catch(java.sql.SQLException e){
        System.out.println("Cannot create SQL statement");
        System.exit(1);
        }

//Display text read from database
       text.setText("Text retrieved from file:");
       button.setText("Click Again");
       _clickMeMode = false;
//Display text read from database
```

```
    } else {
      text.setText("Text to save to file:");
      textField.setText("");
      button.setText("Click Me");
      _clickMeMode = true;
    }
   }
  }
}
```

```
import java.io.*; import javax.servlet.*; import javax.servlet.http.*; import java.sql.*; import java.net.*; import java.io.*; public class DbaServlet extends HttpServlet { private Connection c; final static private String _driver = "sun.jdbc.odbc.JdbcOdbcDriver"; final static private String _user = "username"; final static private String _pass = "password"; final static private String _url = "jdbc:odbc:jdc"; public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException { response.setContentType("text/html"); PrintWriter out = response.getWriter(); out.println ("" + ""); out.println("
```

Button Clicked

```
"); String DATA = request.getParameter("DATA"); if(DATA != null){ out.println("Text from form:"); out.println(DATA); } else { out.println("No text entered."); } //Establish database connection try{ Class. forName (_driver); c = DriverManager.getConnection(_url, _user,_pass); }catch (java.sql.SQLException e){ System.out.println("Cannot get connection"); System.exit(1); }catch (java.lang. ClassNotFoundException e) { System.out.println("Driver class not found"); } try{ //Code to write to database Statement stmt = c.createStatement(); String updateString = "INSERT INTO dba " + "VALUES ('" + DATA + "')"; int count = stmt.executeUpdate(updateString); //Code to read from database ResultSet results = stmt.executeQuery("SELECT TEXT FROM dba "); while(results.next()) { String s = results.getString("TEXT"); out.println("
Text from database:"); out.println(s); } stmt.close(); }catch(java.sql.SQLException e){ System.out. println("Cannot create SQL statement"); System.exit(1); } out.println("

Return to Form"); out.close(); } }
```

```
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;

class SwingUI extends JFrame
             implements ActionListener {

  JLabel text, clicked;
  JButton button, clickButton;
  JPanel panel;
  private boolean _clickMeMode = true;

  SwingUI(){ //Begin Constructor
    text = new JLabel("I'm a Simple Program");
    button = new JButton("Click Me");
    button.addActionListener(this);

    panel = new JPanel();
    panel.setLayout(new BorderLayout());
    panel.setBackground(Color.white);
    getContentPane().add(panel);
    panel.add(BorderLayout.CENTER, text);
    panel.add(BorderLayout.SOUTH, button);
  } //End Constructor

  public void actionPerformed(ActionEvent event){
      Object source = event.getSource();
      if (_clickMeMode) {
        text.setText("Button Clicked");
        button.setText("Click Again");
        _clickMeMode = false;
      } else {
        text.setText("I'm a Simple Program");
        button.setText("Click Me");
        _clickMeMode = true;
      }
  }

  public static void main(String[] args){
    SwingUI frame = new SwingUI();
    frame.setTitle("Example");
```

```
    WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };

    frame.addWindowListener(l);
    frame.pack();
    frame.setVisible(true);
  }

}
```

```
import java.awt.Font;
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.File;

class FileIOError extends JFrame
             implements ActionListener {

   JLabel text;
   JButton button;
   JPanel panel;
   JTextField textField;
   private boolean _clickMeMode = true;

   FileIOError(){ //Begin Constructor
     text = new JLabel("Text to save to file:");
     button = new JButton("Click Me");
     button.addActionListener(this);
     textField = new JTextField(20);

     panel = new JPanel();
     panel.setLayout(new BorderLayout());
     panel.setBackground(Color.white);
     getContentPane().add(panel);
     panel.add("North", text);
     panel.add("Center", textField);
     panel.add("South", button);
   } //End Constructor

   public void actionPerformed(ActionEvent event){
     Object source = event.getSource();
     if(source == button){
       if(_clickMeMode){
        JLabel label = new JLabel();

//Write to file
        try{
```

```
            String text = textField.getText();
            byte b[] = text.getBytes();

            String outputFileName = System.getProperty("user.home", File.separatorChar + "home" +
File.separatorChar + "monicap") + File.separatorChar + "text.txt";
            File outputFile = new File(outputFileName);
            FileOutputStream out = new FileOutputStream(outputFile);
            out.write(b);
            out.close();
        }catch(java.io.IOException e){
            System.out.println("Cannot write to text.txt");
        }

//Read from file
        try{
            String inputFileName = System.getProperty("user.home", File.separatorChar + "home" + File.
separatorChar + "monicap") + File.separatorChar + "text.txt";
            File inputFile = new File(inputFileName);
            FileInputStream in = new FileInputStream(inputFile);
            byte bt[] = new byte[(int)inputFile.length()];
            int i;
            i = in.read(bt);
            String s = new String(bt);
            label.setText(s);
            in.close();
        }catch(java.io.IOException e){
            System.out.println("Cannot read from text.txt");
        }
         text.setText("Text retrieved from file:");
         button.setText("Click Again");
         _clickMeMode = false;
       } else {
         text.setText("Text to save to file:");
         textField.setText("");
         button.setText("Click Me");
         _clickMeMode = true;
       }
     }
   }

  public static void main(String[] args){
    FileIO frame = new FileIO();
    frame.setTitle("Example");
```

```
    WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };

    frame.addWindowListener(l);
    frame.pack();
    frame.setVisible(true);
  }
}
```

```
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;
import java.applet.Applet;
import java.io.*;

public class FileIOAppl extends JApplet  implements ActionListener {
   JLabel text;
   JButton button;
   JPanel panel;
   JTextField textField;
   private boolean _clickMeMode = true;

   public void init(){
     getContentPane().setLayout(new BorderLayout(1, 2));
     getContentPane().setBackground(Color.white);
     text = new JLabel("Text to save to file:");
     button = new JButton("Click Me");
     button.addActionListener(this);
     textField = new JTextField(30);
     getContentPane().add(BorderLayout.NORTH, text);
     getContentPane().add(BorderLayout.CENTER, textField);
     getContentPane().add(BorderLayout.SOUTH, button);
   }
  public void start() {
     System.out.println("Applet starting.");
}

  public void stop() {
    System.out.println("Applet stopping.");
  }

  public void destroy() {
    System.out.println("Destroy method called.");
  }

  public void actionPerformed(ActionEvent event){
    Object source = event.getSource();
    if (source == button) {
      String s = null;
//Variable to display text read from file
```

```
    if (_clickMeMode) {
      try {
   //Code to write to file
        String text = textField.getText();
        String outputFileName = System.getProperty("user.home",
                          File.separatorChar + "home" +
                          File.separatorChar + "zelda") +
                          File.separatorChar + "text.txt";
         FileWriter out = new FileWriter(outputFileName);
         out.write(text);
         out.close();

    //Code to read from file
         String inputFileName = System.getProperty("user.home",
                         File.separatorChar + "home" +
                         File.separatorChar + "zelda") +
                         File.separatorChar + "text.txt";
          File inputFile = new File(inputFileName);
          FileReader in = new FileReader(inputFile);
          char c[] = new char[(int)inputFile.length()];
          in.read(c);
          s = new String(c);
          in.close();
         } catch(java.io.IOException e) {
          System.out.println("Cannot access text.txt");
         }
//Clear text field
        textField.setText("");
//Display text read from file
        text.setText("Text retrieved from file:");
        textField.setText(s);
        button.setText("Click Again");
        _clickMeMode = false;
       } else {
//Save text to file
        text.setText("Text to save to file:");
        button.setText("Click Me");
        textField.setText("");
        _clickMeMode = true;
       }
     }
   }//end action performed method
}
```

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class FileIOServlet extends HttpServlet {
   public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException
  {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<body bgcolor=FFFFFF>");
      out.println("<h2>Button Clicked</h2>");
      String data = request.getParameter("data");
      if (data != null && data.length() > 0) {
        out.println("<STRONG>Text from form:</STRONG>");
        out.println(data);
      } else {
        out.println("No text entered.");
      }
      try {
//Code to write to file
        String outputFileName=
        System.getProperty("user.home",
        File.separatorChar + "home" +
        File.separatorChar + "monicap") +
        File.separatorChar + "text.txt";
        FileWriter fout = new FileWriter(outputFileName);
        fout.write(data);
        fout.close();
//Code to read from file
        String inputFileName =
          System.getProperty("user.home",
          File.separatorChar + "home" +
          File.separatorChar + "monicap") +
          File.separatorChar + "text.txt";
        FileReader fin = new FileReader(inputFileName);
        char c[] = new char[(char)inputFileName.length()];
        fin.read(c);
        String s = new String(c);
        out.println("<P><STRONG>Text from file:</STRONG>");
        out.println(s);
```

```
        fin.close();
      } catch(java.io.IOException e) {
         System.out.println("Cannot access text.txt");
      }
      out.println("<P>Return to <A HREF=../simpleHTML.html>Form</A>");
      out.close();
  }
}
```

```
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;

class AppendIO extends JFrame implements ActionListener {
  JLabel text;
  JButton button;
  JPanel panel;
  JTextField textField;
  private boolean _clickMeMode = true;

  AppendIO() { //Begin Constructor
    text = new JLabel("Text to save to file:");
    button = new JButton("Click Me");
    button.addActionListener(this);
    textField = new JTextField(30);
    panel = new JPanel();
    panel.setLayout(new BorderLayout());
    panel.setBackground(Color.white);
    getContentPane().add(panel);
    panel.add(BorderLayout.NORTH, text);
    panel.add(BorderLayout.CENTER, textField);
    panel.add(BorderLayout.SOUTH, button);
  } //End Constructor

  public void actionPerformed(ActionEvent event){
    Object source = event.getSource();
    if (source == button){
      String s = null;
      if (_clickMeMode){
        try {
     //Write to file
        String text = textField.getText();
        byte b[] = text.getBytes();
        String outputFileName = System.getProperty("user.home",
                          File.separatorChar + "home" +
                          File.separatorChar + "zelda") +
                          File.separatorChar + "text.txt";
        File outputFile = new File(outputFileName);
        RandomAccessFile out = new RandomAccessFile(outputFile, "rw");
```

```
        out.seek(outputFile.length());
        out.write(b);

     //Write a new line (NL) to the file.
        out.writeByte('\n');
        out.close();

    //Read from file
        String inputFileName = System.getProperty("user.home",
                        File.separatorChar + "home" +
                        File.separatorChar + "zelda") +
                        File.separatorChar + "text.txt";
        File inputFile = new File(inputFileName);
        FileInputStream in = new FileInputStream(inputFile);
        byte bt[] = new byte[(int)inputFile.length()];
        in.read(bt);
        s = new String(bt);
        in.close();
       } catch(java.io.IOException e) {
        System.out.println(e.toString());
       }
//Clear text field
       textField.setText("");
//Display text read from file
        text.setText("Text retrieved from file:");
        textField.setText(s);
        button.setText("Click Again");
        _clickMeMode = false;
      } else {
//Save text to file
        text.setText("Text to save to file:");
        textField.setText("");
        button.setText("Click Me");
        _clickMeMode = true;
      }
   }
 }//end action performed method

 public static void main(String[] args) {
   JFrame frame = new AppendIO();
   frame.setTitle("Example");
   WindowListener l = new WindowAdapter() {
   public void windowClosing(WindowEvent e) {
```

```
      System.exit(0);
    }
   };
   frame.addWindowListener(l);
   frame.pack();
   frame.setVisible(true);
  }
}
```

```
<HTML>
<HEAD>
<TITLE>Example</TITLE>
</HEAD>
<BODY BGCOLOR="WHITE">

<TABLE BORDER="2" CELLPADDING="2">
<TR><TD WIDTH="275">

<H2>I'm a Simple Form</H2>

Enter some text and click the Submit button.<BR>
Clicking Submit invokes
<A HREF="./ExampServlet.java">ExampServlet.java</A>,<BR>
which returns an HTML page to the browser.

<FORM METHOD="POST" ACTION="/servlet/ExampServlet">

<INPUT TYPE="TEXT" NAME="DATA" SIZE=30>

<P>
<INPUT TYPE="SUBMIT" VALUE="Click Me">
<INPUT TYPE="RESET">
</FORM>

</TD></TR>
</TABLE>

</BODY>
</HTML>
```

import java.io.*; import javax.servlet.*; import javax.servlet.http.*; public class ExampServlet extends HttpServlet { public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException { response.setContentType("text/html"); PrintWriter out = response. getWriter(); out.println("" + ""); out.println("

Button Clicked

"); String DATA = request.getParameter("DATA"); if(DATA != null){ out.println(DATA); } else { out. println("No text entered."); } out.println("

Return to Form"); out.close(); } }

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;

public class SimpleApplet extends Applet{

  String text = "I'm a simple applet";

  public void init() {
      text = "I'm a simple applet";
      setBackground(Color.cyan);
  }

  public void start() {
      System.out.println("starting...");
  }

  public void stop() {
      System.out.println("stopping...");
  }

  public void destroy() {
      System.out.println("preparing to unload...");
  }

   public void paint(Graphics g){
      System.out.println("Paint");
      g.setColor(Color.blue);
      g.drawRect(0, 0,
                 getSize().width -1,
                 getSize().height -1);
      g.setColor(Color.red);
      g.drawString(text, 15, 25);
   }
}
```

```
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;
import java.applet.Applet;

public class ApptoAppl extends Applet
                implements ActionListener {

   JLabel text;
   JButton button;
   JPanel panel;
   private boolean _clickMeMode = true;

   public void init(){
     setLayout(new BorderLayout(1, 2));
     setBackground(Color.white);

     text = new JLabel("I'm a Simple Program");
     button = new JButton("Click Me");
     button.addActionListener(this);
     add("Center", text);
     add("South", button);
   }

  public void start(){
     System.out.println("Applet starting.");
  }

  public void stop(){
     System.out.println("Applet stopping.");
  }

  public void destroy(){
     System.out.println("Destroy method called.");
  }

   public void actionPerformed(ActionEvent event){
        Object source = event.getSource();
        if (_clickMeMode) {
          text.setText("Button Clicked");
          button.setText("Click Again");
```

```
        _clickMeMode = false;
      } else {
        text.setText("I'm a Simple Program");
        button.setText("Click Me");
        _clickMeMode = true;
      }
  }
}
```

```
//A Very Simple Example
class ExampleProgram {

   public static void main(String[] args){

       System.out.println("I'm a Simple Program");
   }
}
```

```
class LessonTwoA {
   static String text = "I'm a Simple Program";
   public static void main(String[] args){
       System.out.println(text);
   }
}
```

```
class LessonTwoB {

  String text = "I'm a Simple Program";
  static String text2 = "I'm static text";

  String getText(){
      return text;
  }

  String getStaticText(){
      return text2;
  }

  public static void main(String[] args){
      LessonTwoB progInstance = new LessonTwoB();
      String retrievedText = progInstance.getText();
      String retrievedStaticText = progInstance.getStaticText();
      System.out.println(retrievedText);
      System.out.println(retrievedStaticText);
  }
}
```

```
class LessonTwoC {

  static String text = "I'm a Simple Program";

  static String getText(){
      return text;
  }

  public static void main(String[] args){
      String retrievedText = getText();
      System.out.println(retrievedText);
  }
}
```

```
class LessonTwoD {

  String text;

  LessonTwoD(){
    text = "I'm a Simple Program";
  }

  String getText(){
      return text;
  }

  public static void main(String[] args){
      LessonTwoD progInst = new LessonTwoD();
      String retrievedText = progInst.getText();
      System.out.println(retrievedText);
  }
}
```