Essentials of the Java Programming Language

A Hands-On Guide

by Monica Pawlan

350 East Plumeria Drive
San Jose, CA 95134
USA

May 2013
Part Number TBD
v1.0

If you are new to programming in the Java programming language (Java) and have some experience with other languages, this tutorial could be for you. It walks through how to use the Java Platform software to develop a basic network application that uses common Java platform features. This tutorial is not comprehensive, but instead takes you on a straight and uncomplicated path through the more common features available in the Java platform. This tutorial is a learning tool and should be viewed as a stepping-stone for persons who find the currently available materials a little too overwhelming to start with.

To reduce your learning curve, this tutorial begins with a simple program in Lesson 1, develops the program by adding new features in every lesson, and leaves you with a general electronic commerce application, and a basic understanding of object-oriented programming concepts in Lesson 15. Unlike other more reference-style texts that give you a lot of definitions and concepts at the beginning, this tutorial takes a practical approach. New features and concepts are described when they are added to the example application.

Please note the final application is for instructional purposes only and would need more work to make it production worthy. By the time you finish this tutorial, you should have enough knowledge to comfortably go on to other Java programming language learning materials and continue your studies.

If you have no programming experience at all, you might still find this tutorial helpful; but you also might want to take an introductory programming course before you proceed.

Lessons 1 through 8 explain how applications, applets, and servlets/JavaServer Pages are similar and different, how to build a basic user interface that handles simple user input, how to read data from and write data to files and databases, and how to send and receive data over the network.

Lessons 9 through 15 walk you through socket communications, building a user interface using more components, grouping multiple data elements as one unit (collections), saving data between program invocations (serialization), and internationalizing a program. Lesson 15 concludes the series with basic object-oriented programming concepts.

This tutorial covers object-oriented concepts at the end after you have had practical experience with the language so you can relate the object-oriented concepts to your experiences.

Appendix A presents the complete and final code for this tutorial.

JavaBean Technology

> JavaBean technology, which lets you create portable program components that follow simple naming and design conventions, is not covered here. While creating a simple JavaBean component is easy, understanding JavaBeans features requires knowledge of such things as properties, serialization, events, and inheritance. When you finish these lessons, you should have the knowledge you need to go on to a good text on JavaBeans technology and continue your studies.

Acknowledgements

Many Java Developer Connection (JDC) members contributed comments and suggestions to this material when the first eight lessons appeared on the JDC website in March and April of 1999, and the last eight appeared the following July. With those suggestions and many others received from the review team at Addison-Wesley Longman, the material has evolved into an introduction to Java programming language features for persons new to the platform and unfamiliar with the terminology.

I also relied on the help of co-workers, friends, and family for whose help I am very grateful. I would like to thank my friend and co-worker, Mary Aline, for providing the French translations for the *Chapter 13, Internationalization* chapter, and my best friend and husband Jeffrey Pawlan (WA6KBL) who worked with Wolf Geihe (DJ4OA) in Germany to provide the German translations for that same chapter. I do not want to forget Stephanie Wilde, our contract editor at the JDC, who helped with copy editing on the early versions of this material posted to the JDC website. And Dana Nourie, our JDC HTML editor, who in her quest to learn Java, provided unending enthusiasm for this work and contributed to the section on how to set the `CLASSPATH` environment variable on the Windows platform.

Special thanks go to Allan Jacobs and Orson Alvarez who went through the example code and text making a number of excellent suggestions to improve them, and to Calvin Austin whose helpful suggestions at the outset made the earlier lessons more understandable and accessible to novice programmers. Finally, thanks to Danesh Forouhari who made some excellent suggestions to improve the graphics, and who encouraged me to include a short section on JavaServer Pages technology. And I cannot forget my manager, Margaret Ong, who stood behind me all of the way in this effort.

Lastly, I want to acknowledge various individual reviewers within Sun Microsystems, Inc., whose expert knowledge in their respective areas was an invaluable asset to completing the examples: Rama Roberts (object-oriented programming), Dale Green (internationalization), Alan Sommerer (JAR file format and packages), Joshua Bloch (collections), and Tony Squier (databases).

Contents

Chapter 5 Building Servlets

Chapter 6 Access and Permissions

Chapter 7 Database Access and Permissions

Chapter 8 Remote Method Invocation

Chapter 9 Socket Communications

Chapter 10 Object-Oriented Programming

Chapter 11 User Interfaces Revisited

Chapter 12 Develop the Example

Chapter 13 Internationalization

Chapter 14 Packages and JAR File Format

Appendix A Code Listings

Index

Compile and Run a Simple Program

1

If you are new to Java, you might have heard of applets, applications, servlets, and JavaServer Pages, but are not sure what they are and how they differ. Or maybe you are just curious about the basic set of application programming interfaces (APIs) available in the platform and do not want to read a lot of pages to learn the basics.

This short tutorial gives you a hands-on introduction to Java. It starts with compiling and running the simple program presented in this lesson, adds new features with explanations in each successive lesson, and introduces APIs commonly used in general programs.

This lesson covers the following topics:

About the Java Platform

Before you can write and compile programs, you need to understand what the Java platform is and configure your computer to run the programs. The Java platform consists of the Java APIs and the Java Virtual Machine (JVM).

Java APIs are libraries of compiled code that you can use in your programs. They enable you to 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 printing capability is provided in the API ready for you to use. You supply the text to be printed.

Figure 1 shows the Java platform architecture. The JVM sits on top of your native operating system. Your program sits on top of the JVM and calls compiled code from the API libraries that live within the JVM.

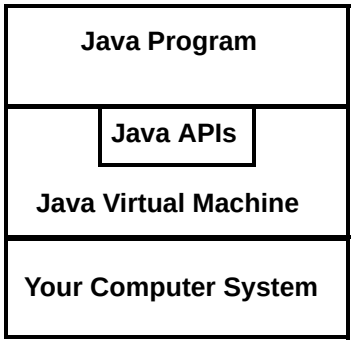


Figure 1. Java Platform Architecture

Programs written in Java are run (or interpreted) by another program called the JVM. If you have used 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 JVM for the native operating system. This means that any computer system with a JVM installed can run programs written in Java regardless of the computer system on which the applications were originally developed.

Set Up Your Computer

Before you can write and run the simple 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 *oracle.com* website. Choose the correct Java SE software for your operating system and refer to the installation instructions.

Write a Program

Use the text editor of your choice to create a text file with the following text (source code). Name the text file `ExampleProgram.java`. Programs written in Java are case sensitive.

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

Compile the Program

When you compile a Java program, the source code is converted to byte codes, which are platform-independent instructions for the JVM.

Execute the Java compiler as follows:

```
javac ExampleProgram.java
```

Run the Program

Once your program successfully compiles, you can interpret and run the program on any JVM. The JVM byte code interpreter converts the Java byte codes to platform-dependent machine codes so that you computer or browser can understand and run the program.

Execute the `java` command as follows to run the example program:

```
java ExampleProgram
```

The following commands show the entire sequence to compile and run the example program:

```
> javac ExampleProgram.java
> java ExampleProgram.java
I’m a Simple Program
```

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 accommodate these needs, Java supports three kinds of comments: double slashes, C-style, and doc comments.

Double Slashes

You can use C++-style double slashes (`//`) to 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 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 on the console. */
class ExampleProgram {
   public static void main(String[] args){
      System.out.println("I'm a Simple Program");
   }
}
```

API Documentation

The Java platform installation includes API Documentation, which describes the APIs available for you to use in your programs. By default, the files are stored in a `src.zip` file beneath the directory where you installed the platform.

Exercises

1. What is the name of the program that runs (or interprets) programs written in Java?
2. Name the interpreter command, and explain what it does.
3. Name the compiler command, and explain what it does.

Building Applications

2

All applications, applets, and servlets written in Java have almost the same structure and share many common elements. They also have some differences. This lesson describes the structure and elements common to applications.

This lesson covers the following topics:

Application Structure and Elements

You crate an application from classes. A class defines class fields to store the data, and class methods to work on the data. A `class` is similar to a `struct` in the C and C++ languages in that it can store related data of different types, but the big difference between a class and a `struct` is that a class also defines accessor methods to work on its data. The C and C++ languages separate functions from the `struct` that defines the data.

Every application needs one class with a `main` method. The class with the `main` method 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.

The `ExampleProgram.java` code from *Chapter 1* has no fields or accessor methods. Because `ExampleProgram` 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");
   }
}
```

In the above code, the `public static void` keywords mean the JVM interpreter can call the program `main` method to start the program (`public`) without creating an instance of the class (`static`), and the program does not return data to the JVM interpreter (`void`) when it ends.

An instance of a class has data members and methods as defined by that class. While the class describes the data and methods to work on the data, a class instance acquires and works on the data.

Figure 2 shows three instances of the `StoreData` class by the names: `FirstInstance`, `SecondInstance` and `ThirdInstance`. While class instances share the same definition (class), they are separate from each other in that each instance can acquire and work on different data.

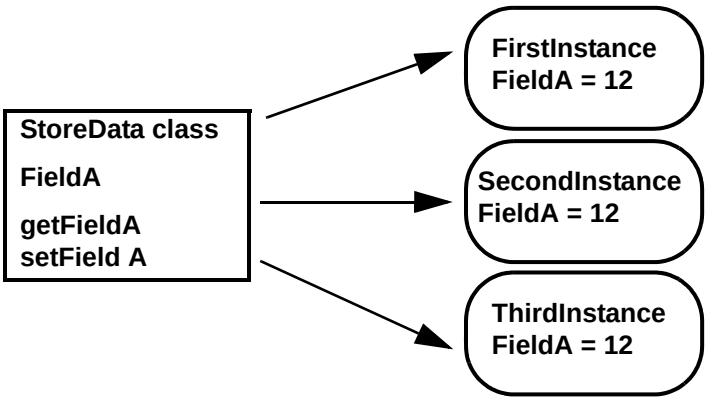


Figure 2. Class Instances

It is not always necessary to create a class instance to call methods and access fields in a class. An instance of the `ExampleProgram` class is never created because it has no fields to access and only the one `static main` method to call. The `main` method for `ExampleProgram` 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 Java platform lets a program call a method in a class without creating an instance of that class as long as the method being called is `static`. Just as the JVM interpreter command can call the `static main` method in the `ExampleProgram` class without creating an instance of it, the `ExampleProgram` class can call the `static println` method in the `System` class without creating an instance of the `System` class.

As you explore Java, you will come across library classes such as `System`, `Math`, or `Color` that contain all or some `static` methods and fields, and you might find that `static` methods and fields can make sense when you write your own classes.

For example, the `Color` class provides ready access to common colors such as red, blue, and magenta through its `static` fields, and you can get custom colors by creating a `Color` class instance and passing specific values to the `Color` class constructor. For more information on constructors, see *Constructors* .

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 that its data can be accessed directly by the `static println` method without creating an instance of the `LessonTwoA` class.

```
class LessonTwoA {
   //Static field added
   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` 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.

The example also includes a `static` text field and a non-static instance method to retrieve it. A non-static method can access both `static` and non-static fields.

```
class LessonTwoB {
   //Static and non-static fields
   String text = “I’m a Simple Program”;
   static String text2 = “I’m static text”;

   //Methods to access data in the fields
   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` 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();
      .out.println(retrievedText);
   }
}
```

Class methods can operate only on class fields, but instance methods can operate on class and instance fields. The difference is that there is only one copy of the data stored in a class field, but each instance has its own copy of the data stored in an instance field.

For example, the following `ExampleClass` class definition has one static field, one instance field, and two accessor methods to set the value for each field.

```
class ExampleClass {
   static FieldA = 36;
   FieldB=0;
   return text;
```

```
    private void setFieldA (value){
       FieldA = value;
    }
    private void setFieldB (value) {
       FieldB = value;
    }

    public static void main(String[] args){
       // Do something
    }
}
```

If another class creates two instances of `ExampleClass`, then, `FieldA` has the value 36 and `FieldB` has the value 0 for both instances. *Figure 3* shows the following:

- If another class calls `setFieldA` on the first instance of `ExampleClass` with a value of 25, then the `FieldA` value in both instances changes to 25.
- If another class calls `setFieldB` on the first instance of `ExampleClass` with a value of 50, then the `FieldB` value in the first instance changes to 25, but the `FieldB` value in the other instances remains 0.

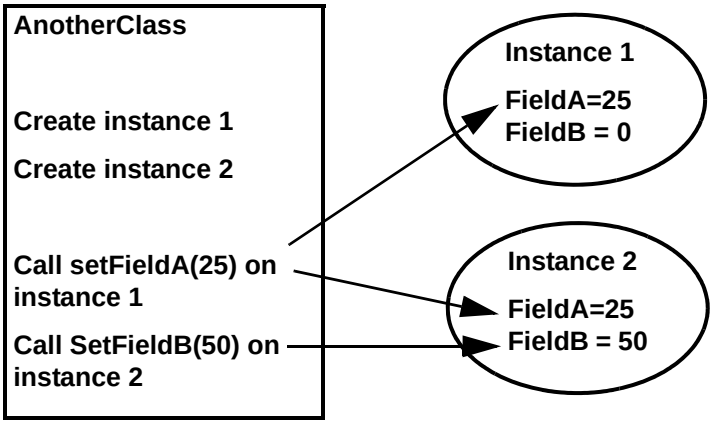


Figure 3. Change Class and Instance Field Values

Constructors

A constructor is a special method that prepares the new instance for use by initializing the instance fields. The constructor always has the same name as the class and no return type.

If you do not write your own constructor, the compiler adds an empty constructor. The empty constructor is called the default constructor and initializes all non-initialized fields and variables to zero. A constructor might or might not have parameters depending on whether the class provides its own initialization data or gets it from the calling method.

Figure 4 shows the constructor, accessor methods, and main method.

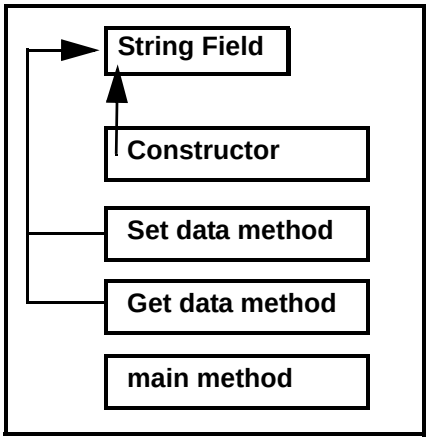


Figure 4. Constructor

The `LessonTwoD` program converts the `LessonTwoB` program to use a constructor without parameters to initialize the text string field.

```
   class LessonTwoD{
      String text;

      //Constructor
      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);
   }
}
```

The `LessonTwoE` program passes the string to be printed to the constructor as a parameter:

```
class LessonTwoE{
   String text;

   //Constructor
   LessonTwoE(String message){
   text = message;
}
   String getText(){
   return text;
}
   public static void main(String[] args){
      LessonTwoE progInst = new LessonTwoE("I'm a simple program");
      String retrievedText = progInst.getText();
      System.out.println(retrievedText);
   }
}
```

Exercises

1. An application must have one class with which kind of method?
2. What is the difference between class and instance fields?
3. What are accessor methods?

Building Applets

3

Like applications, you create applets from classes. However, applets do not have a `main` method as an entry point, do have several methods to control specific aspects of applet execution, and while applications run in the JVM installed on a computer system, applets run in the JVM installed in a web browser. You can also run an applet in a special tool for testing applets called `appletviewer`.

This lesson converts one of the applications from *Chapter 2, Building Applications* to an applet, describes the structure and elements common to applets, and shows you how to use the `appletviewer` tool.

This lesson covers the following topics:

- *Application to Applet*
- *Run the Applet*
- *Applet Structure and Elements*
- *Packages*
- *Exercises*

Application to Applet

The following code is the applet equivalent to the `LessonTwoB` example in *Chapter 2*. *Figure 5* shows how the running applet looks. See *Run the Applet* for information on the structure and elements of the applet code.

Figure 5. A Simple Applet

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

//Make applet class public
public class SimpleApplet extends Applet{
   String text = "I'm a simple applet";

   public void init() {
      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.drawRect(0, 0, getSize().width -1, getSize().height -1);
      g.setColor(Color.red);
      g.drawString(text, 15, 25);
   }
}
```

The `SimpleApplet` class is `public` so that the program that runs the applet (a browser or the `appletviewer` tool), which is not local to the program, can execute it.

Make sure to compile the applet:

```
javac SimpleApplet.java
```

Run the Applet

To execute the applet, create an HTML file with the Applet tag as follows:

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

An easy way to run the applet is to use the `appletviewer` tool. The following `appletviewer` command executes the `simpleApplet.html` file, which contains the above HTML code:

```
appletviewer simpleApplet.html
```

Applet Structure and Elements

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

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

Extend a Class

Most classes of any complexity extend other classes. To extend another class means to take the data and behavior from the parent class and add more data and/or behavior to the child class. In the C++ language, this is called subclassing.

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 their inherited methods. Java allows only single inheritance where a child class is limited to one parent.

Figure 6 shows the class hierarchy for the `SimpleApplet` class. The `Object` class is the parent of all Java classes not explicitly extended from any other class.

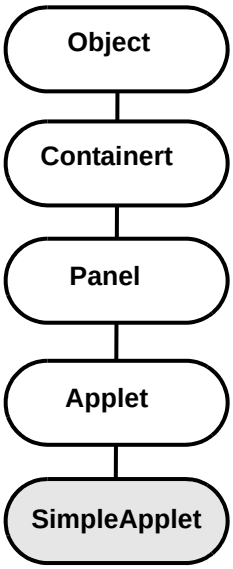


Figure 6. Extending the Applet Class

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

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

You might wonder why Java 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, tools, and browsers, or portable across multiple platforms. For example, both 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 the `appletviewer` tool as you saw in the example. The underlying software controls the applet by calling the methods the applet inherits from the `Applet` class. You do not have to implement all of these methods. You implement only the methods you need.

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 to function 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.

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

The start Method

The `start` method is called when the applet is visited such as when the 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.

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

After the `start` method executes, the platform 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 is made up of 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 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.

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

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

Appearance

The `Applet` class is a `Panel` component that inherits a `paint` method from its parent `Container` class. To draw onto the applet's `Panel`, 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 that explicitly import the `Applet`, `Graphics`, and `Color` classes in the `java.applet` and `java.awt` API library packages for use in the applet. Applications of any size and all applets access ready-made Java API classes organized into *packages* located elsewhere on the local or networked system. 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.

There are two ways to access these ready-made libraries: `import` statements, which you saw in the code in *Application to Applet* , and full package names. The following code rewrites the example applet to use full package names instead of `import` statements. 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. For example `java.lang.String` and `mypackage.String` reference two completely different classes.

```
public class SimpleApplet extends java.applet.Applet{
   String text = "I'm a simple applet";

   public void init() {
      text = "I'm a simple applet";
      setBackground(java.awt.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(java.awt.Graphics g){
      System.out.println("Paint");
      g.setColor(Color.blue);
      g.drawRect(0, 0,
      getSize().width -1,
      getSize().height -1);
      g.setColor(java.awt.Color.red);
      g.drawString(text, 15, 25);
   }
}
```

Note: The examples do not need a package declaration to call `System.out.println` because the `System` class is in the default `java.lang` package.

Exercises

1. What are some differences between applications and applets?
2. Name the applet methods that control the applet's behavior.
3. Describe two ways to access API library classes organized into packages from your programs.

Building a User Interface

4

This lesson adds a user interface to the `LessonTwoD` application from *Chapter 2, Building Applications*. The user interface is built with the Java Foundation Classes (JFC) Project Swing (Project Swing) APIs. Project Swing APIs provide a wide-range of classes for building friendly and interesting user interfaces and handling action events from user inputs such as mouse clicks and keyboard presses.

This is a very basic introduction to Project Swing that is developed more in *Chapter 11, User Interfaces Revisited*.

This lesson covers the following topics:

- *Project Swing APIs*
- *Import Statements*
- *Class Declaration*
- *Instance Variables*
- *Constructor*
- *Action Listening*
- *Event Handling*
- *Main Method*
- *Exercises: Applets Revisited*
- *Applet and Application Differences*

Project Swing APIs

The Project Swing API provides the building blocks (components) for creating interesting and friendly user interfaces. You can choose from basic controls such as buttons and checkboxes, components that contain other components such as frames and panels, and information displays such as labels and text areas.

When you build a user interface, you place basic components and information displays inside container components. If the user interface has many elements, then place container components within other container components. Ultimately, every applet and application has a top-level container to hold all of its user interface components.

An applet's top-level container is a browser window, and an application's top-level container is a frame. A frame component is a window that provides a title, banner, and methods to manage the appearance and behavior of the window. An applet relies on the browser for this type of functionality. An applet can have only one top-level panel, but an application can have many top-level panels.

The Project Swing code for this lesson builds the simple application in *Figure 7*. The frame (window) on the left appears when you start the application, and the frame on the right appears when you click the button. Click again to go back to the original frame on the left.

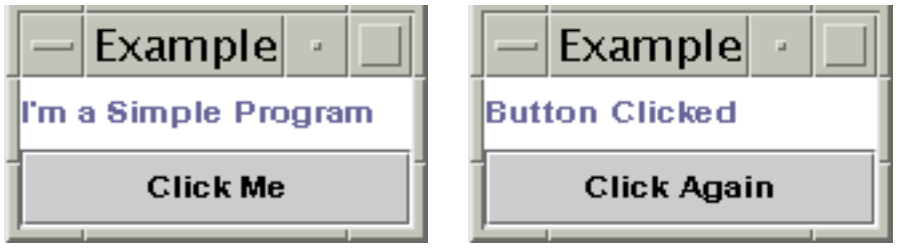


Figure 7. Project Swing Application

Import Statements

The import statements in the `SwingUI.java` code indicate which Java API packages and classes the program uses. The first two lines import specific classes in the Abstract Window Toolkit (`awt`) package, and the third line imports the `event` package within the `awt` package.

Your code is clearer to someone else reading it when you import exactly the classes and packages you need and no others. But, if you use a lot of classes in one package, it is probably easier to import an entire package including its subpackages as shown by the fourth `import javax.swing` statement.

The Abstract Window Toolkit (AWT) is an API library that provides classes for building a user interface and handling action events. However, Project Swing extends the AWT with a full set of GUI components and services, pluggable look and feel capabilities, and assistive technology support. Project Swing components include Java-language versions of the AWT components such as buttons and labels, and a rich set of higher-level components such as list boxes and tabbed panes. Because of the enhanced functionality and capabilities in the Project Swing class libraries, this tutorial focuses on the Project Swing APIs.

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

//Class Declaration
class SwingUI extends JFrame implements ActionListener {

//Instance variables
   JLabel text, clicked;
   JButton button;
   JPanel panel;
   private boolean _clickMeMode = true;

//Constructor
   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
```

```
//Event handling
   public void actionPerformed(ActionEvent event){
      Object source = event.getSource();
         if (source == button) {
            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;
            }
         }
   }

//main method
   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);
   }
}
```

Class Declaration

The class declaration indicates that the top-level frame for the application's user interface extends a `JFrame` class that implements the `ActionListener` interface. The Project Swing `JFrame` class extends the `Frame` class, which is part of the AWT APIs. Project Swing component classes have the same name as their AWT counterparts prefixed with the letter `J`.

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

The `ActionListener` interface, like all other interfaces in Java, defines a set of methods, but does not implement their behavior. Instead, you provide the interface method implementations for the class that implements the interface.

The Java platform lets a class extend only one class, which in this case is `JFrame`, but lets it implement any number of interfaces. In this example, the `SwingUI` class implements the `ActionListener` interface only.

When a program class implements an interface, it must provide behavior for all methods defined in that interface. The `ActionListener` interface has only one method, the `actionPerformed` method. So, the `SwingUI` class must implement the `actionPerformed` method, which is covered in *Event Handling* .

Instance Variables

These next lines in the `SwingUI` class declare the Project Swing component classes the `SwingUI` class uses. These are instance variables (or fields) 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` variable is an instance variable that is only accessible to the `SwingUI` class. It is used in the `actionPerformed` method to find out whether or not the `button` has been clicked.

```
JLabel text;
JButton button;
JPanel panel;
//Start out waiting to be clicked
private boolean _clickMeMode = true;
```

Constructor

The constructor creates the user interface components, adds the components to the `JPanel` object, adds the panel to the frame, and makes the `JButton` components action listeners, which is covered in *Action Listening* . The `JFrame` object is created in the `main` method when the program starts.

```
SwingUI(){
   text = new JLabel(“I’m a Simple Program”);
   button = new JButton(“Click Me”);

//Add button as an event listener
    button.addActionListener(this);

//Create panel
   panel = new JPanel();
```

```
//Specify layout manager and background color
   panel.setLayout(new BorderLayout());
   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 in the display area. The code uses the `BorderLayout` layout manager, which arranges user interface components in the five areas shown in *Figure 8*.

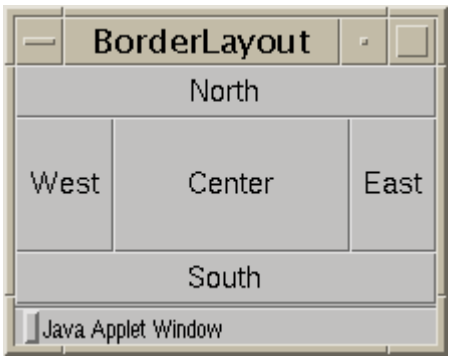


Figure 8. Border Layout

To add a component to the layout, specify the area with the `static` fields provided in the `BorderLayout` class. The next code segment adds components to the panel in the center and south regions of the border layout. Components are added to the content pane where the components reside so that the layout manager can control the component layout and provide functionality that allows different types of components to work together. The call to the `getContentPane` method of the `JFrame` class adds the Panel to the `JFrame` container.

```
//Create panel
   panel = new JPanel();

//Specify layout manager and background color
   panel.setLayout(new BorderLayout());
   panel.setBackground(Color.white);

//Add label and button to panel
   getContentPane().add(panel);
   panel.add(BorderLayout.CENTER, text);
   panel.add(BorderLayout.SOUTH, button);
}
```

Action Listening

In addition to implementing the `ActionListener` interface, you have to add the action listener to the `JButton` components. In this example the action listener is the `SwingUI` object because its class implements the `ActionListener` interface.

What this means in this example is that the `SwingUI` object listens for action events. When a button click action event occurs, the Java platform services pass the button click action to the `actionPerformed` method implemented in the `SwingUI` class. The `actionPerformed` implementation described in Event Handling on page 35 handles the action event.

Component classes have the appropriate `add` methods to add action listeners to them. The `JButton` class has an `addActionListener` method, and the parameter passed to `addActionListener` is `this`, meaning that the `SwingUI` action listener is added to the button and the Java platform services pass any button-generated actions to the `actionPerformed` method in the `SwingUI` class.

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

Event Handling

The Java platform passes an event object to the `actionPerformed` method. The event object represents an action event that occurred. The `actionPerformed` method has an `if` statement to determine whether the `button` component fired the action event and to test the `_clickMeMode` variable to find out the state of the `button` component.

If the `button` component is waiting to be clicked, the label and button text change to reflect that the button was just clicked. If the `button` component has been clicked, the label and button text change to invite another click.

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

Main Method

The `main` method creates the top-level `frame`, sets the title, and includes code that lets the 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 to close the window uses an adapter class. If the event listener interface you need provides more functionality than the program actually uses, you can use an adapter class. The Java APIs provide adapter classes for all listener interfaces with more than one method. 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 seven methods and this program needs only the `windowClosing` method so it makes sense to use the `WindowAdapter` class instead of the `WindowListener` interface.

The next code segment extends the `WindowAdapter` class and overrides the `windowClosing` method. The `new` keyword creates an anonymous instance of the extended inner class. Anonymous means that you do not assign 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. Implementing the `WindowListener` interface would require six empty method implementations. Remember to add the `WindowAdapter` object to the `frame` object so the `frame` object listens 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);
```

Exercises: Applets Revisited

Using what you learned in *Chapter 3* and in this lesson, convert the example code for this lesson into the applet shown in *Figure 9*. A solution that uses the `JApplet` follows the figure.

You could also use the `Applet` class, which was the class used in *Chapter 3*. The `JApplet` class is the Project Swing class equivalent for creating applets, and the applet code for this exercise is almost identical except the `JApplet` class requires calls to `getContentPane()` to set the layout and color and to add components to the panel, which you do not need if you use the `Applet` class.

Figure 9. Applet Version of Application

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

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

   public void init() {
      getContentPane().setLayout(new BorderLayout());
      getContentPane().setBackground(Color.white);
      text = new JLabel("I'm a Simple Program");
      button = new JButton("Click Me");
      button.addActionListener(this);
      getContentPane().add(BorderLayout.CENTER, text);
      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){
          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;
          }
       }
    }
}
```

Applet and Application Differences

The differences between the applet and application versions of the example are as follows:

- The applet class is `public` so another program such as `appletviewer` can access it.
- The applet class extends `Applet`. The application class extends `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`. User interface components are added to the content plane of an application `JFrame` object.

Building Servlets

5

Like applications and applets, you use classes to build servlets. But servlets are different from applications and applets in that the purpose of a servlet is to extend a server program to enhance its functionality. One very common use for servlets is to extend a web server by providing dynamic web content.

This lesson shows how to create a very simple browser-based HTML form that executes a basic servlet to process user data that is entered onto the form. The example is the servlet version of the applet and application examples studied so far. This lesson concludes with how to convert the servlet to a JavaServer Page (JSP).

Servlets are easy to write. All you need is Tomcat, which is the combined Java Server pages and Servlets reference implementation. You can download a free copy of Tomcat form the java.sun.com website.

This lesson covers the following topics:

About the Example

Web servers respond to browser requests with the HyperText Transfer Protocol (HTTP). HTTP is the protocol for moving hypertext files across the internet, and HyperText Markup Language (HTML) documents contain text that has been marked up for interpretation by an HTML browser such as FireFox.

A browser accepts user input through an HTML form. The simple form used in this lesson has one text input field for the user to enter text and a Submit button. When the user clicks the Submit button, the simple servlet executes and processes the user input. In this example, the simple servlet returns an HTML page that displays the text entered by the user.

Figure 10 shows the flow of data between the browser and servlet for this example.

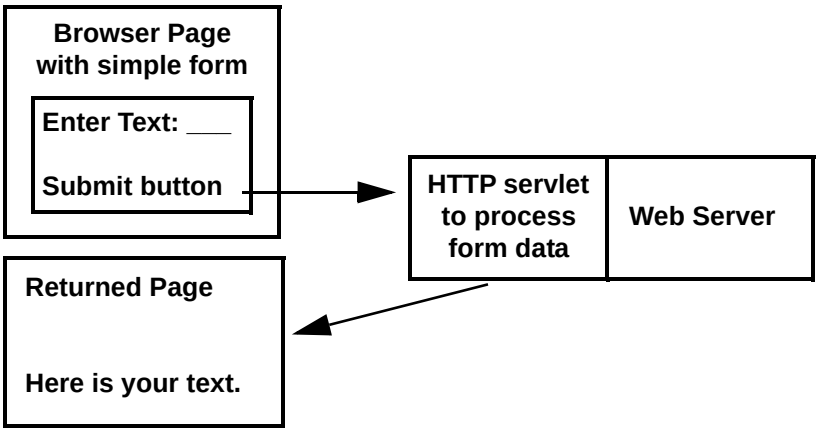


Figure 10. Returning an HTML Pages

HTML Form

Figure 11 shows the HTML form for the example. The following code for HTML form has an `ACTION` parameter that is shown in bold where you specify the location of the servlet.

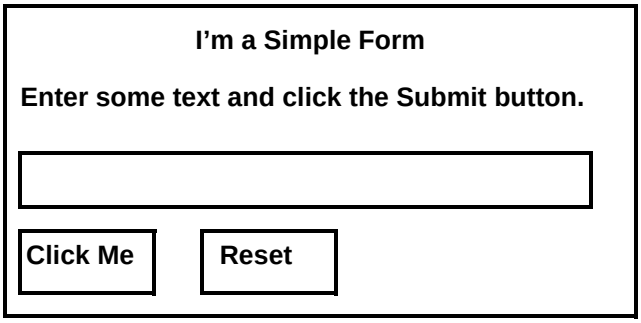


Figure 11. Simple HTML Form

```
<HTML>
<HEAD>
<TITLE>Example</TITLE>
</HEAD>
<BODY BGCOLOR="WHITE"><BR>
<H2>I'm a Simple Form</H2>
Enter some text and click the Submit button.<BR>
<FORM METHOD="POST" ACTION="/servlet/ExampServlet">
<INPUT TYPE="TEXT" NAME="DATA" SIZE=30>
<P>
<INPUT TYPE="SUBMIT" VALUE="Click Me">
<INPUT TYPE="RESET">
</FORM>
</BODY>
</HTML>
```

When the user clicks the `Click Me` button on the form, the servlet gets the text entered and returns an HTML page with the text. *Figure 12* shows an example page returned by `ExampServlet.java`. See *Servlet Code* for a description of the servlet code that retrieves the user input and generates this HTML page.

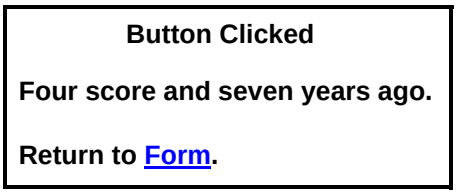


Figure 12. HTML Page Returned to Browser

To run the example, put the servlet and HTML files in the correct directories for the web server as indicated by the web server documentation or your administrator.

Servlet Code

The `ExampServlet` program builds an HTML page to return to the user. The servlet code does not use any Project Swing or AWT components or have action event handling code. For this simple servlet, you need to import only the following 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(“<body bgcolor=FFFFFF>”);
      out.println(“<h2>Button Clicked<h2>”);
      String data = request.getParameter(“data”);

      if(data != null && data.length() > 0){
         out.println(data);
      } else {
         out.println(“No text entered.”);
      }
      out.println(“<P>Return to <A HREF=../exampServlet.htm>Form</A>”);
      out.close();
   }
}
```

Class and Method Declarations

All HTML servlet classes extend the `HttpServlet` abstract class. 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. You extend the abstract class and implement the methods you need so all HTTP servlets use a common framework to handle the HTTP protocol.

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

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

GET requests might pass parameters to a URL by appending them to the URL. A POST request might pass additional data to a URL by directly sending it to the server separate from the URL. A POST request cannot be bookmarked or emailed and does not change the URL of the response. A GET request can be bookmarked, emailed, and can add information to the URL of the response.

The parameter list for the doPost method takes a request and 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. It also uses the response object to create an HTML page in response to the browser 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(“<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 is not set, and returns an empty string if the parameter has no 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 && data.length() > 0){
   out.println(data);
}else {
   out.println(“No text entered.”);
}
```

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

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

JavaServer Pages Technology

JavaServer Pages (JSP) let you put segments of servlet code directly within a static HTML or eXtensible Markup Language (XML) page. When the JSP page executes, the application server creates, compiles, loads, and runs a background servlet to execute the servlet code segments and return an HTML page.

A JSP page looks like an HTML or XML page with servlet code segments embedded between various forms of leading (<%) and closing (%>) JSP tags. There are no `HttpServlet` methods such as `doGet` and `doPost`. Instead, the code that would normally be in those methods is embedded directly in the JSP page with scriptlet tags.

HTML Form

It is straightforward to convert the servlet example to a JSP page. First, change the `ACTION` parameter in the HTML form to invoke the JSP page instead of the servlet as shown below. Note that the JSP page is not in the servlets directory, but in the same directory with the HTML page.

```
<HTML>
<BODY BGCOLOR="WHITE">
<TITLE>Example</TITLE>
<TABLE><TR><TD WIDTH="275">

<H2>I’m a Simple Form</H2>
Enter some text and click the Submit button.<BR>

<FORM METHOD="POST" ACTION="ExampJsp.jsp">
<INPUT TYPE="TEXT" NAME="data" SIZE=30>
<P>
<INPUT TYPE="SUBMIT" VALUE="Click Me">
<INPUT TYPE="RESET">
</FORM>
</TD></TR></TABLE>
</BODY>
</HTML>
```

JSP Page

The following JSP page (`ExampJSP.jsp`) is equivalent to `ExampServlet`. It starts with the usual `HTML`, `HEAD`, `TITLE`, and `BODY` tags and concludes with closing the `BODY` and `HTML` tags. In between are two types of JSP tags: directives and script lets.

JSP directives are instructions that are processed by the JSP engine when the JSP page is translated to a servlet. The directives used in this example tell the JSP engine to include certain packages and classes. Directives are enclosed by the <%@ and %> directive tags.

JSP scriptlets let you embed Java code segments into the JSP page. The embedded code is inserted directly into the servlet that executes when the page is requested. Scriptlets are enclosed in the <% and %> scriptlet tags.

A scriptlet can use the following predefined variables: `request`, `response`, `out`, and `in`. This means that you can use these variables without declaring them. For example, the `PrintWriter out = response.getWriter()` line used in the servlet code to create the `out` object is not needed in a JSP page.

```
<HTML>
<HEAD>
<TITLE>Example JSP Page</TITLE>
</HEAD>

<BODY>
<%@ page import="java.io.*" %>
<%@ page import="javax.servlet.*" %>
<%@ page import="javax.servlet.http.*" %>

<%
   response.setContentType("text/html");
   out.println("<body bgcolor=FFFFFF>");
   out.println("<h2>Button Clicked<h2>");
   String data = request.getParameter("data");

   if(data != null && data.length() > 0){
      out.println(data);
   } else {
      out.println("No text entered.");
   }
   out.println("<P>Return to <A HREF=../exampJsp.html>Form</A>");
   out.close();
%>
</BODY>
</HTML>
```

Other JSP tags you can use are comments (<%-- comment %>), declarations (<%! String data %>), expressions (<%= request.getParameter %>), and JSP-Specific tags. The following code shows the JSP page converted to use the comment and declaration tags.

```
<HTML>
<HEAD>
```

```
<TITLE>Example JSP Page</TITLE>
</HEAD>

<%-- Import Statements --%>
<%@ page import="java.io.*" %>
<%@ page import="javax.servlet.*" %>
<%@ page import="javax.servlet.http.*" %>

<%-- Declaration --%>
<%! String data; %>

<%
   response.setContentType("text/html");
   out.println("<body bgcolor=FFFFFF>");
   out.println("<h2>Button Clicked<h2>");
   data = request.getParameter("data");

   if(data != null && data.length() > 0){
      out.println(data);
   }else {
      out.println("No text entered.");
   }
   out.println("<P>Return to <A HREF=../exampJsp.html>Form</A>");
   out.close();
%>
</BODY>
</HTML>
```

Exercises

- What is the function of a servlet?
- What does the `request` object do?
- What does the `response` object do?
- How is a JSP page different from a servlet?

Access and Permissions

6

So far, you have learned how to retrieve and handle a short text string entered from the keyboard into a simple UI or HTML form. But programs also retrieve, handle, and store data in files and databases.

This lesson expands the applet, application, and servlet examples from the previous lessons to perform basic file access using the APIs in the `java.io` package. It also shows you how to grant applets and servlets permission to access specific files, and how to restrict an application so it has access to specific files only. You learn how to perform similar operations on a database in *Chapter 7, Database Access and Permissions*.

This lesson covers the following topics:

- *File Access by Applications*
- *File Access by Applets*
- *Restrict Applications*
- *File Access by Servlets*
- *Exercises*
- *Code for This Lesson*

File Access by Applications

The Java platform provides a rich range of classes for reading character data (alphanumeric) or byte data (a unit consisting of a combination of eight 1's and 0's) 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. See *Code for This Lesson* for full source code listings.

This section shows you how to read data from and write data to a file on the local computer system.

- **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 *Chapter 4, Building a User Interface* to accept user input through a text field and then save it to a file.

The window on the left appears when you start the `FileIO` application. When you click the button, whatever is entered into the text field is saved to a file. After that, the file is opened, read, and its text 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.

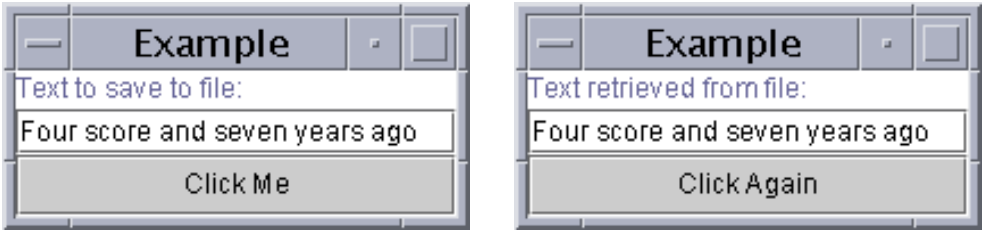


Figure 13. Click the Button

The conversion from the `SwingUI` program from *Chapter 4* to the `FileIO` program for this lesson primarily involves the `constructor` and the `actionPerformed` method as described in the next sections.

Constructor and Instance Variable Changes

The `constructor` instantiates the `textField` with a value of 30. This value tells the Java platform the number of columns to use to calculate the preferred width of the text field object. Lower values result in a narrower display, and likewise, higher values result in a wider display.

Next, the `text` label object is added to the `North` section of the `BorderLayout` and the `textField` object is added to the `Center` section.

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

```
//Constructor
   FileIO(){
      text = new JLabel(“Text to save to file:”);
      button = new JButton(“Click Me”);
      button.addActionListener(this);
//Text field instantiation
      textField = new JTextField(30);
      panel = new JPanel();
      panel.setLayout(new BorderLayout());
      panel.setBackground(Color.white);
      getContentPane().add(panel);
//Adjustments to layout to add text field
      panel.add(BorderLayout.NORTH, text);
      panel.add(BorderLayout.CENTER, textField);
      panel.add(BorderLayout.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 instead of character streams. Character streams are used in the applet example. A more detailed explanation of the changes to the method implementation comes after the code.

```
public void actionPerformed( ActionEvent event){
   Object source = event.getSource();
   if(source == button){
      String s = null;
      //Variable to display text read from file
      if (_clickMeMode){
         FileInputStream in=null;
         FileOutputStream out=null;
      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 + “zelda”) +
                  File.separatorChar + “text.txt”;
            out = new FileOutputStream(outputFileName);
            out.write(b);
            out.close();
```

```
      //Clear text field
            textField.setText("");
      //Code to read from file
            String inputFileName = System.getProperty(“user.home”,
                  File.separatorChar + “home” +
                  File.separatorChar + “zelda”) +
                  File.separatorChar + “text.txt”;
                  File.separatorChar + "text.txt";
            File inputFile = new File(inputFileName);
            in = new FileInputStream(inputFile);
            byte bt[] = new byte[(int)inputFile.length()];
            in.read(bt);
            String s = new String(bt);
            in.close();
         } catch(java.io.IOException e){
            System.out.println(“Cannot access text.txt”);
         } finally {
            try {
               in.close();
               out.close();
            } catch(java.io.IOException e) {
               System.out.println("Cannot close");
            }
         }
         //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 input 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 `FileOutputStream` object is created to open an output stream on the `text.txt` file.

```
String outputFileName = System.getProperty(“user.home”,
                        File.separatorChar + “home” +
                        File.separatorChar + “zelda”) +
                        File.separatorChar + “text.txt”;
out = new FileOutputStream(outputFileName);
```

Finally, the `FileOutputStream` object writes the byte array to the `text.txt` file and closes the output stream when the write 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 + "zelda") +
                        File.separatorChar + "text.txt";
File inputFile = new File(inputFileName);
in = new FileInputStream(inputFile);
```

Next, a byte array is created that is the same length as the file into which the text is stored.

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

Finally, the byte array is used to construct a `String` object, which contains the retrieved text displayed in the `textField` component. The `FileInputStream` is closed when the operation completes.

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

You might think you could simplify this code by not creating the `File` object and just passing the `inputFileName` `String` object directly to the `FileInputStream` constructor. The problem is the `FileInputStream` object reads a byte stream, and a byte stream is created to a specified size. In this example, the byte stream needs to be the same size as the file, and that size varies depending with the text written to the file. The File class has a length method that lets you get this value so the byte stream can be created to the correct size each time.

System Properties

The previous 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 + "zelda"` in case no value is found for this key.

File.separatorChar

The 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 provides a way to construct platform-independent pathnames.

For example, the pathname `/home/zelda/text.txt` for UNIX and `\home\zelda\text.txt` for Windows are written as `File.separatorChar + "home" + File.separatorChar + "zelda" + File.separatorChar + "text.txt"` in a platform-independent construction.

Exception Handling

Java includes classes that represent conditions that can be thrown by a program during execution. Throwable classes can be divided into error and exception conditions and descend from the `java.lang.Exception` and `java.lang.Error`. classes shown in *Figure 14*. An Exception subclass indicates throwable exceptions that a typical application would want to catch, and an `Error` subclass indicates a serious throwable error that a typical application would not catch.

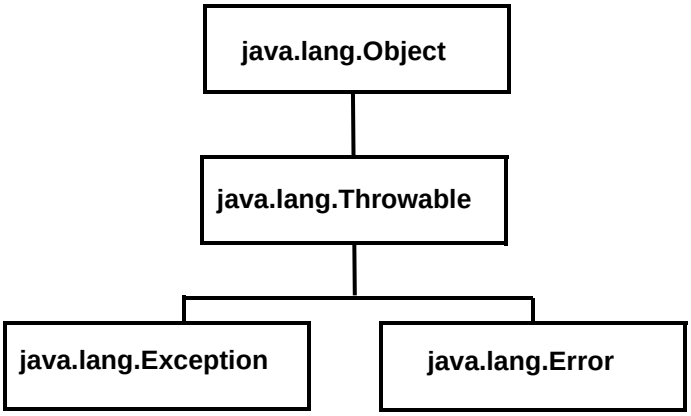


Figure 14. Exception Classes

All exceptions except `java.lang.RuntimeException` and its subclasses are called checked exceptions. The Java platform requires that a method catch or specify all checked exceptions that can be thrown within the scope of a method. If a checked exception is not either caught or specified, the compiler throws an error.

In the `FileIO` example, the `actionPerformed` method has file input and output code that could throw a `java.lang.IOException` checked exception in the event the file cannot be created for the write operation or opened for the read operation. To handle these possible exception situations, the file input and output code in the `actionPerformed` method is enclosed in a `try` and `catch` block.

```
try {
   //Write to file
   //Read from file
} catch (java.lang.IOException e) {
   //Do something if read or write fails
}
```

If a method does not catch a checked exception, then the method must specify that it can throw the checked exception because a checked exception that can be thrown by a method is part of the method's public interface. Callers of the method must know about the checked exceptions a method might throw so they can take appropriate actions to handle the exception.

While it is true that checked exceptions must be either caught or specified, sometimes you do not have a choice. For example, the `actionPerformed` method already has a public interface definition that cannot be changed to specify the `java.io.IOException`. If you add a throws clause to the `actionPerformed` method, you will get a compiler error. So in this case, the only thing you can do is catch and handle the checked exception.

However, methods you define yourself can either specify exceptions or catch and handle them. Here is an example of a user-defined method that specifies an exception. In this case the method implementation does not have to catch and handle `IllegalValueException`, but callers of this method must catch and handle `IllegalValueException`. The decision to throw an exception or catch and handle it in a user-defined method depends on the method behavior. Sometimes it makes more sense for the method to handle its own exceptions and sometimes it makes more sense to give that responsibility to the callers.

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

Whenever you catch exceptions, you should handle them in a way that is friendly to your 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 problems.

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 on page 173

later because you will have access to the text. *Display Data in a Dialog Box* explains how to display a dialog box with the text you want.

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

The example uses a `finally` block for closing the `in` and `out` objects. The `finally` block is the final step in a `try` and `catch` block and contains clean-up code for closing files or releasing other system resources. Statements in the `finally` block are executed no matter what happens in the `try` block. So, even if an error occurs in the `try` block, you can be sure the `Statement` and `ResultSet` objects will be closed to release the memory they were using.

```
} finally {
   try {
      in.close();
      out.close();
   } catch(java.io.IOException e) {
      System.out.println("Cannot close");
   }
}
```

File Access by Applets

The file access code for the `FileIOAppl` applet is equivalent to the `FileIO` 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 byte streams in the application and character streams in the applet is arbitrary. In your programs, 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 exceptions:

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

> **Note:** See *Grant Applets Permission* before you run the applet.

```
public void actionPerformed(ActionEvent event){
   Object source = event.getSource();
   if (source == button){
   //Variable to display text read from file
      String s = null;
      if (_clickMeMode){
         FileReader in=null;
         FileWriter out=null;
         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";
            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);
            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”
         } finally {
            try {
               in.close();
               out.close();
            } catch(java.io.IOException e) {
                  System.out.println("Cannot close");
            }
         }
         //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");
            _clickMeMode = true;
        }
    }
}
```

Grant Applets Permission

If you tried to run the applet example in a directory other than in your home directory, you undoubtedly saw errors when you clicked the `Click Me` button. This is because Java platform security imposes restrictions on applets. An applet cannot access local system resources such as files without explicit permission. In the example for this lesson, the applet cannot write to or read from the `text.txt` file without explicit permission.

Java platform security is enforced by the default security manager, which disallows all potentially threatening access operations unless the applet executes with a policy file that specifically grants the needed access. So for the `FileUIAppl` program to write to and read from the `text.txt` file, the applet has to execute with a policy file that grants the appropriate read and write access to the `text.txt` file.

Creating a Policy File

Policy tool is a Java platform security tool for creating policy files. The policy file you need to run the applet appears below. You can use `policy` tool to create it (type `policytool` at the command line) or copy the following text into an ASCII file. The advantage to using Policy tool is that you can avoid typos and syntax errors that make the policy file ineffective.

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

Run 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 the `appletviewer` tool like this:

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

If your browser is enabled for the Java 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 and rename it `java.policy` for Windows and `.java.policy` for UNIX.

This is an HTML file to run the `FileIOAppl` applet:

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

Restrict Applications

Normally, applications do not run under the default security manager, but you can launch an application with special command-line options and a policy file to achieve the same kind of restriction you get with applets. This is how to do it:

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

Because the application runs within the security manager, which disallows all access, the policy file needs two additional permissions over those required to run the applet. The `accessEventQueue` permission lets the security manager access the event queue where action events are stored and load the user interface components. The `showWindowWIthoutWarningBanner` permission lets the application execute without displaying the banner warning that its window was created by 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. The `FileIOServlet program` writes to and reads from the `text.txt` file without restriction under Java WebServer 1.1.1.

Exercises

Error Handling: If you want to make the code for this lesson easier to read, 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. The `FileIO` class shows the solution.

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` class to append to the file. If you need help, see the `AppendIO` class on page 64.

Code for This Lesson

- *FileIO Program*
- *FileIOAppl Program*
- *FileIOServlet Program*
- *AppendIO Program*

FileIO Program

```
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.awt.Color;
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 Java
   if (source == button) {
      String s = null;
      //Write to file
      if (_clickMeMode){
         FileInputStream in=null;
         FileOutputStream out=null;
         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");
         } finally {
            try {
            in.close();
            out.close();
            } catch(java.io.IOException e) {
               System.out.println("Cannot close");
            }
         }
         //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);
   }
}
```

FileIOAppl Program

```
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;
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) {
         FileReader in=null;
         FileWriter out=null;
         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");
         } finally {
            try {
               in.close();
               out.close();
            } catch(java.io.IOException e) {
               System.out.println("Cannot close");
            }
         }
         //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
}
```

FileIOServlet Program

```
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");
      FileReader fin = null;
      FileWriter fout = null;

      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";
         fout = new FileWriter(outputFileName);
         fout.write(data);
         //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);
         fin = new FileReader(inputFile);
         char c[] = new char[30];
         fin.read(c);
         String s = new String(c);
         out.println("<P><STRONG>Text from file:</STRONG>");
         out.println(s);
      } catch(java.io.IOException e) {
         System.out.println("Cannot access text.txt");
      } finally {
         try {
         fout.close();
         fin.close();
         } catch(java.io.IOException e) {
            System.out.println("Cannot close");
      }
   }
      out.println("<P>Return to <A HREF=../simpleHTML.html>Form</A>");
      out.close();
   }
}
```

AppendIO Program

```
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){
         RandomAccessFile out = null;
         FileInputStream in = null
         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);
            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 + "text2.txt";
            File inputFile = new File(inputFileName);
            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());
         } finally {
            try {
               out.close();
               in.close();
            } catch(java.io.IOException e) {
               System.out.println("Cannot close");
            }
         }
         //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);
   }
}
```

Database Access and Permissions 7

This lesson converts the application, applet, and servlet examples from *Chapter 6, Access and Permissions* to write to and read from a database using JDBC technology. JDBC is the Java database connectivity API available in the Java platform software.

The code for this lesson is very similar to the code you saw in *Chapter 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 typically not part of your program code. However, establishing a database connection and accessing the database are part of your code.

As in *Chapter 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. This lesson explains how to determine the permissions your program needs to successfully run.

This lesson covers the following topics:

Database Setup

You need access to a database to run the examples in this lesson. You can install a database on your machine or you might have access to a database at work. Either way, you also need a database driver and any relevant environment settings so your program can load the driver into memory and locate the database.

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 is 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 driver documentation for information on installation and other environment settings you need for your platform. If you are using a database at work, consult your database administrator.

To show you three ways to establish a database connection, 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 involve similar steps and code.

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 named `dba` 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 *Chapter 6* to write data to and read data from a database. The top window in *Figure 15* 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 table. Then, the data is retrieved from the database table and redisplayed in the window as shown on the bottom. If you write data to the table more than once, the database table will have multiple rows and you might have to enlarge the window to see all the data.

See *Chapter 7, Code for This Lesson* for the full source code listings.

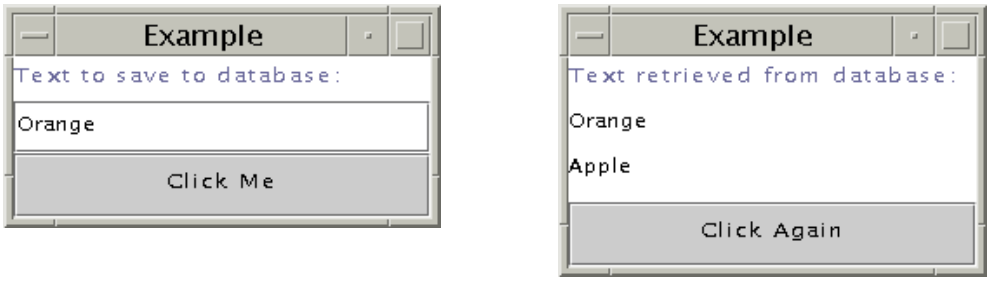


Figure 15. Database Access by Applications

Establish a Database Connection

A database connection is established with the `DriverManager` and `Connection` classes available in the `java.sql package`. The JDBC `DriverManager` class handles multiple database drivers and initiates all database communication. To load the driver and connect to the database, the application needs a `Connection` object and `String` objects that represent the `_driver` and `_url`.

The `_url` string is in the form of a URL. It consists of the URL, Oracle subprotocol, and Oracle data source in the form `jdbc:oracle:thin`, plus username, password, machine, port, and protocol information.

```
private Connection c;
private final String _driver =   “oracle.jdbc.driver.OracleDriver”;
private final String _url =
               “jdbc:oracle:thin:username/password@developer:1521:ansid”;
```

The `actionPerformed` method calls the `Class.forName(_driver)` method to load the driver, and the `DriverManager.getConnection` method to establish the connection. These calls are enclosed by `try` and `catch` blocks.

Exception Handling describes `try` and `catch` blocks. The only thing different in this code is that this block uses two `catch` statements because two different checked exceptions must be caught.

The call to `Class.forName(_driver)` throws the `java.lang.ClassNotFoundException`, and the call to `c = DriverManager.getConnection(_url)` throws the `java.sql.SQLException`. With 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 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 the information.

`private`: A `private` variable can only be 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.

Write and Read 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) {
   if (_clickMeMode) {
      JTextArea displayText = new JTextArea();
      Statement stmt = null;
      ResultSet results = null'
      try {
         //Code to write to database
         String theText = textField.getText();
         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 sign (+). 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. The code iterates through the `ResultSet`, retrieves the data, and appends the data to the `displayText` text area.

```
         //Code to read from database
         results = stmt.executeQuery( “SELECT TEXT FROM dba “);
         while(results.next()){
         String s = results.getString(“TEXT”);
         displayText.append(s + “\n”);
      }
      } catch(java.sql.SQLException e) {
         System.out.println(“Cannot create SQL statement”);
      } finally {
         try {
            stmt.close();
            results.close();
         } catch(java.sql.SQLException e) {
            System.out.println("Cannot close");
         }
      }
      //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;
   }
}
```

Database Access by Applets

The applet version of the example is like the application code except for the standard differences between applications and applets described in *Chapter 3, Building Applets*..

However, if you run the applet without a policy file, you get a stack trace that indicates access denied errors. You learned about policy files and how to use one to launch an applet with the permissions it needs in *Grant Applets Permission* . In that lesson, you had a policy file with the correct permissions and told how to use it to launch the applet. This lesson explains how to read a stack trace to determine the permissions you need in a policy file.

This lesson has two versions of the database access applet: one uses the JDBC driver, and the other uses the JDBC-ODBC bridge with an Open DataBase Connectivity (ODBC) driver. Both applets perform the same operations on the same database table with different drivers. Each applet has its own policy file with different permission lists and has different requirements for locating the database driver.

See *Code for This Lesson* for the full source code listings.

JDBC Driver

The JDBC driver is meant to be used in a program written exclusively in Java. 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.

Start the Applet

To successfully run, the `DbaAppl` applet needs an available database driver and policy file. This section walks through the steps to get everything set up. Here is the 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 DbaApplet.html
```

Locate 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 that directory, 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 dbaApplet.html
```

Read a Stack Trace

Assuming the driver is locally available to the applet and the `DbaAppl` applet is launched without a policy file, you get the following stack trace when you click the `Click Me` button.

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

The first line in the 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 tells you that to correct this condition you need a `SocketPermission` that gives the applet access to the machine where the database is located. For this example, that machine is named `developer`.

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

```
grant {
   permission java.net.SocketPermission “developer”, “resolve”;
};
```

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

```
appletviewer -J-Djava.security.policy=DbaApplPol dbaApplet.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 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 with the above policy file with the permissions.

```
appletviewer -J-Djava.security.policy=DbaApplPol dbaApplet.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 UNIX and Windows versions of the Java platform so you can do two things:

- Use ODBC from a program written in Java.
- 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. This means the driver file does not have to be locally accessible.

Start the Applet

Here is an HTML file for running the `DbaOdbAppl` applet. The entire `DbaOdbAppl` source code appears on 82.

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

And here is how to start the applet:

```
appletviewer dbaOdb.html
```

Read a Stack Trace

If the `DbaOdbAppl` applet is launched without a policy file, the following stack trace is generated when the user clicks the `Click Me` button.

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

The first line in the 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 JVM.

You can use Policy tool to create the policy file you need, or you can create it with an ASCII editor. This 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, but 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. This 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 permissions indicated, it should work just fine.

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

Note: If you install Java Plug-In and run this applet from your browser, put the policy file in your home directory and rename it `java.policy` for Windows and `.java.policy` for UNIX.

Database Access by Servlets

As explained in *Chapter 5, Building Servlets*, servlets are under the security policy in force for the web server under which they run. When the `DbaServlet` 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.

See *Code for This Lesson* for the full source code listings.

Exercises

1. What are `final` variables?
2. What does a `Statement` object do?
3. How can you determine which permissions an applet needs to access local system resources such as a database?

Code for This Lesson

- *Dba Program*
- *DbaAppl Program*
- *DbaOdbAppl Program*
- *DbaServlet Program*

Dba Program

```
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;
   JButton button;
   JPanel panel;
   JTextField textField;
   private Connection c;
   private boolean _clickMeMode = true;

   private final String _driver = “oracle.jdbc.driver.OracleDriver”;
   private final String _url =
                  "jdbc:oracle:thin:username/password@developer:1521: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”);
      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();
         Statement stmt = null;
         ResultSet results = null;
         try{
            //Code to write to database
            String theText = textField.getText();
            stmt = c.createStatement();
            String updateString = "INSERT INTO dba VALUES (‘" + theText + "‘)";
            int count = stmt.executeUpdate(updateString);
            //Code to read from database
            results = stmt.executeQuery("SELECT TEXT FROM dba ");
            while (results.next()) {
               String s = results.getString("TEXT");
               displayText.append(s + "\n");
            }
         } catch(java.sql.SQLException e) {
            System.out.println("Cannot create SQL statement");
         } finally {
            try {
               stmt.close();
               results.close();
            } catch(java.sql.SQLException e) {
               System.out.println("Cannot close");
```

```
               }
            }
            //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);
   }
}
```

DbaAppl Program

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

public class DbaAppl extends JApplet implements ActionListener {
   JLabel text;
   JButton button;
```

```
JTextField textField;
private Connection c;
private boolean _clickMeMode = true;

private final String _driver = “oracle.jdbc.driver.OracleDriver”;
private final String _url =
         "jdbc:oracle:thin:username/password@developer:1521:ansid";

public void init() {
   getContentPane().setBackground(Color.white);
   text = new JLabel(“Text to save to file:”);
   button = new JButton(“Click Me”);
   button.addActionListener(this);
   textField = new JTextField(20);
   getContentPane().setLayout(new BorderLayout());
   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) {
   try{
      Class.forName (_driver);
      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 (_clickMeMode){
       JTextArea displayText = new JTextArea();
       Statement stmt = null;
       ResultSet results = null;
       try{
          //Code to write to database
          String theText = textField.getText();
          stmt = c.createStatement();
          String updateString = "INSERT INTO dba VALUES (‘" + theText + "‘)";
          int count = stmt.executeUpdate(updateString);
          //Code to read from database
          results = stmt.executeQuery("SELECT TEXT FROM dba ");
          while (results.next()) {
             String s = results.getString("TEXT");
             displayText.append(s + "\n");
          }
       } catch(java.sql.SQLException e) {
          System.out.println("Cannot create SQL statement");
       } finally {
          try {
             stmt.close();
             results.close();
          } catch(java.sql.SQLException e) {
             System.out.println("Cannot close");
          }
       }
       //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;
       }
    }
}
```

```
}
```

DbaOdbAppl Program

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

public class DbaOdbAppl extends JApplet implements ActionListener {
   JLabel text, clicked;
   JButton button, clickButton;
   JTextField textField;
   private boolean _clickMeMode = true;
   private Connection c;
   private final String _driver = “sun.jdbc.odbc.JdbcOdbcDriver”;
   private final String _user = “username”;
   private final String _pass = “password”;
   private final 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);
      getContentPane().setLayout(new BorderLayout());
      getContentPane().setBackground(Color.white);
      getContentPane().add(BorderLayout.NORTH, text);
      getContentPane().add(BorderLayout.CENTER, textField);
      getContentPane().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();
         Statement stmt = null;
         ResultSet results = null;
         try{
            //Code to write to database
            String theText = textField.getText();
            stmt = c.createStatement();
            String updateString = "INSERT INTO dba VALUES (‘" + theText + "‘)";
            int count = stmt.executeUpdate(updateString);
            //Code to read from database
            results = stmt.executeQuery("SELECT TEXT FROM dba ");
            while (results.next()) {
               String s = results.getString("TEXT");
               displayText.append(s + "\n");
            }
         } catch(java.sql.SQLException e) {
            System.out.println("Cannot create SQL statement");
         } finally {
            try {
            stmt.close();
            results.close();
            } catch(java.sql.SQLException e) {
               System.out.println("Cannot close");
            }
```

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

DbaServlet Program

```
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;
   private static final String _driver = “sun.jdbc.odbc.JdbcOdbcDriver”;
   private static final String _user = “username”;
   private static final String _pass = “password”;
   private static final 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(“<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.”);
   }

//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");
   }
   Statement stmt = null;
   ResultSet results = null;
   try {
      //Code to write to database
      stmt = c.createStatement();
      String updateString = "INSERT INTO dba VALUES (‘" + data + "‘)";
      int count = stmt.executeUpdate(updateString);
      //Code to read from database
      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);
      }
   } catch(java.sql.SQLException e) {
      System.out.println("Cannot create SQL statement");
   } finally {
      try {
         stmt.close();
         results.close();
      } catch(java.sql.SQLException e) {
         System.out.println("Cannot close");
      }
   }
   out.println("<P>Return to <A HREF=../dbaHTML.html>Form</A>");
   out.close();
   }
}
```

Remote Method Invocation

8

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

This lesson covers the following topics:

RMI Scenario

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

Figure 16 shows a publicly accessible remote server object that enables client and server communications. 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.

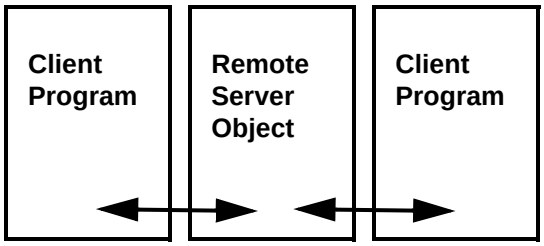


Figure 16. Client and Server Communications

Enterprise JavaBeans technology is another Java API for remote communications. While writing a simple Enterprise Bean is easy, running it requires an application server and deployment tools. Because the Enterprise JavaBeans API is similar to RMI, you should be able to go on to a good text on Enterprise JavaBeans and continue your studies when you finish here.

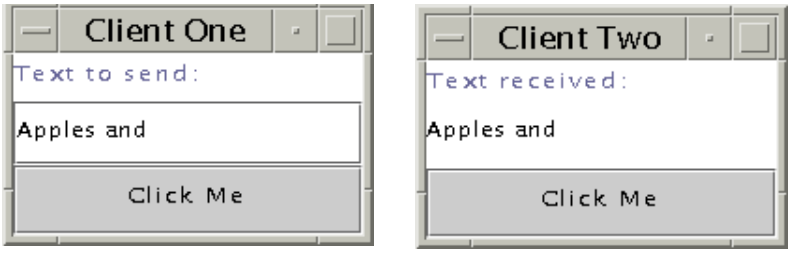
About the Example

This lesson adapts the `FileIO` program from *Chapter 6, Access and Permissions* to use the RMI API.

See *Code for This Lesson* for the full source code listings.

Program Behavior

Figure 17, shows that 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 remote server object. When you click the `Click Me` button on the `RMIClient2` program, the text is retrieved from the remote server object and displayed in the `RMIClient2` user interface.



First instance of Client One

Figure 17. Sending Data Over the Network

As shown in *Figure 18*, if you start a second instance of `RMIClient1`, type in some text and click its `Click Me` button, that text is sent to the remote server object where it can be retrieved by the `RMIClient2` program. To see the text sent by the second client, click the `RMIClient2` `Click Me` button.

Second instance of Client One

Figure 18. Two Instances of Client One

File Summary

Figure 19 shows that the example program consists of the `RMIClient1` program, the remote server object and interface, and the `RMIClient2` program. The corresponding source code files for these executables are described in the bullet list. See *Code for This Lesson* for the full source code listings.

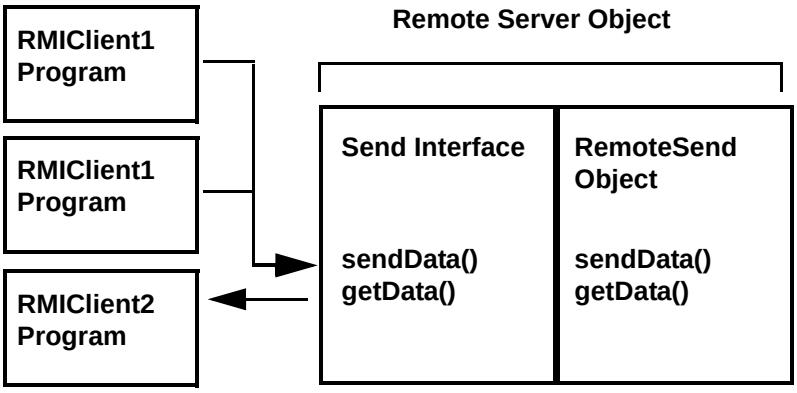


Figure 19. Simple RMI Application

- `RMIClient1.java`: Client program that calls the `sendData` method on the `RemoteServer` server object. The `sendData` method is made available to `RMIClient1` through the `Send` interface.
- `RMIClient2.java`: Client program that calls the `getData` method on the `RemoteServer` server object. The `getData` method is made available to `RMIClient2` through the `Send` interface.
- `Send.java`: Remote interface that declares the `sendData` and `getData` remote server methods. This interface makes the remote server object methods available to clients anywhere on the system.
- `RemoteServer.java`: Remote server object that implements `Send.java` and the `sendData` and `getData` remote 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.

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 next two `javac` commands compile the `RMIClient2` and `RMIClient1` classes.

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 `stub` and `skel` classes let client programs communicate with 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.

- In *Figure 20* you can see that `RMIClient1` is invoked from a client-side directory and uses the server-side web server and client-side JVM to download the publicly accessible files.
- `RMIClient2` is invoked from a client-side directory and uses the server-side web server and client-side JVM to download the publicly accessible files.

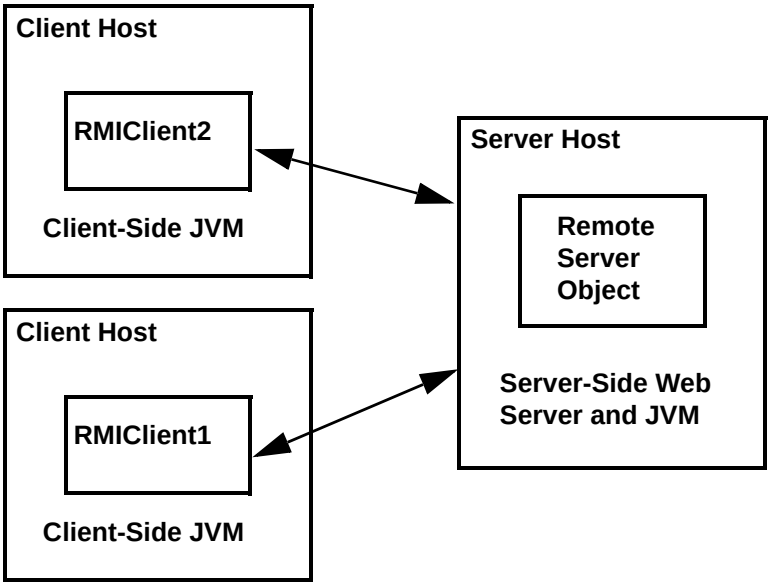


Figure 20. Downloading Publicly Accessible Files

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. The `CLASSPATH` environment variable should not point to `stub` and `skel` classes anywhere on your system except where they are supposed to be. If the RMI Registry finds these classes in another location when it starts, it will not load them from the JVM where the server is running, 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 where 4321 is a port number: `rmiregistry 4321 &`.

If you specify a different port number, you must specify the same port number in your server-side code.

UNIX

```
unsetenv CLASSPATH
```

```
rmiregistry &
```

Win32

```
set CLASSPATH=  CLASSPATH
start rmiregistry
```

> **Note:** You might want to set the `CLASSPATH` back to its original setting now.

Start the Server

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

> **Note:** In this example, the host machine is `kq6py`. To make this example work, substitute this host name with the name of your own machine.

UNIX

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

Win32

```
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

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.

> **Note:** In this example, the host machine is `kq6py`. To make this example work, substitute this host name with the name of your own machine.

UNIX

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

Win32

```
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 the RMIClient2 Program

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.

> **Note:** In this example, the host machine is `kq6py`. To make this example work, substitute this host name with the name of your own machine.

UNIX

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

Win32

```
java -Djava.rmi.server.codebase=file:c:\home\zelda\public_html\classes
\home\zelda\public_html
-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 remotely accessible `sendData` and `getData` methods declared in the `Send` interface. `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 {
   private 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 RMI 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 `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, you can add it with a colon as follows where 4321 is the port number: `kq6py.eng.sun.com:4321`. If you change the port here, you must start the RMI Registry with the same port number. The `try` block creates a `RemoteServer` instance and binds the `name` to the remote object to the RMI Registry with the `Naming.rebind(name, remoteServer)` statement. The string passed to the `rebind` and `lookup` methods is the name of the host on which the name server is running.

```
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 crate 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 in its `main` method and sends data to the remote server object in its `actionPerformed` method.

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 (Exception 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 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 access data in 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 {
      ring 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”);
   }
   ...
```

Exercises

1. What is the RMI Registry?
2. What do you have to start first, the server program or the RMI Registry?
3. What does the RMISecurityManager do?
4. Is the Send or `RemoteServer` object available to clients?

Code for This Lesson

- *RMIClient1 Program*
- *RMIClient2 Program*
- *RemoteServer Program*
- *Send Interface*

RMIClient1 Program

```
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;
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(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) {
   //Send data over net
   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”);
         }
      }
}
```

RMIClient2 Program

```
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;
   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(BorderLayout.NORTH, text);
```

```
      panel.add(BorderLayout.CENTER, textArea);
      panel.add(BorderLayout.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");
      }
   }
}
```

RemoteServer Program

```
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”);
      }
   }
}
```

Send Interface

```
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;
}
```

Socket Communications

9

In the *Chapter 8, Remote Method Invocation* example, multiple client programs communicate with one server program without your writing any explicit code to establish the communication or field the client requests. This is because the RMI API is built on sockets, which enable network communications, and threads that allow the server program to handle simultaneous requests from multiple clients. To help you understand what you get for free with the RMI API, and to introduce the APIs for sockets and multithreaded programming, this lesson presents a simple sockets-based program with a multithreaded server.

Threads let a program perform multiple tasks at one time. Using threads in a server program to field simultaneous requests from multiple client programs is one common use, but threads can be used in programs in many other ways. For example, you can start a thread to play sound during an animation sequence or start a thread to load a large text file while the window to display the text in the file appears. These other uses for threads are not covered in this lesson.

This lesson covers the following topics:

- *What are Sockets and Threads?*
- *About the Examples*
- *Exercises*
- *Code for This Lesson*

What are Sockets and Threads?

A socket is a software endpoint that establishes bidirectional communication between a server program and one or more client programs. *Figure 21* shows how a socket associates the server program with a specific hardware port on the machine where it runs so that any client program anywhere in the network with a socket associated with that same port can communicate with the server program.

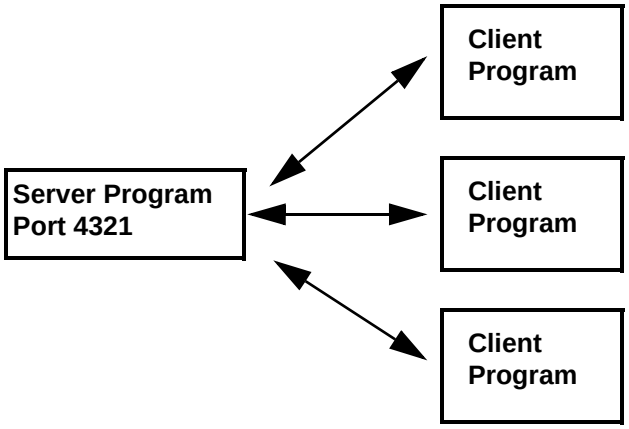


Figure 21. Sockets Connect a Server Program to One or More Client Programs

A server program typically provides resources to a network of client programs. Client programs send requests to the server program, and the server program responds to the request. One way to handle requests from more than one client is to make the server program multithreaded. A thread is a sequence of instructions that run independently of the program and any other threads.

A multithreaded server creates a thread for each communication it accepts from a client. Using threads, a multithreaded server program can accept a connection from a client, start a thread for that communication, and continue listening for requests from other clients.

About the Examples

There are two examples for this lesson. Both are adapted from the `FileIO` program. Example 1 sets up a client-server communication between one server program and one client program. The server program is not multithreaded and cannot handle requests from more than one client.

Example 2 converts the server program to a multithreaded version so it can handle requests from more than one client.

See *Code for This Lesson* for the full source code listings.

Example 1: Client-Side Behavior

The `SocketClient` client program shown in *Figure 22* presents a simple user interface and prompts for text input. When you click the `Click Me` button, the text is sent to the server program. The client program expects an echo from the server and prints the echo on its standard output.

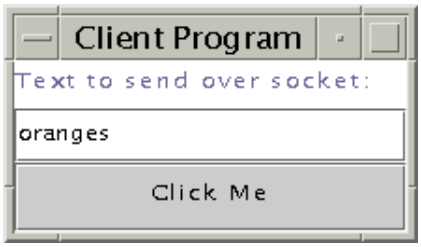


Figure 22. Client Program

Example 1: Server-Side Behavior

The `SocketServer` program shown in *Figure 23* presents a simple user interface, and when you click the `Click Me` button, the text received from the client is displayed. The server echoes the text it receives whether or not you click the `Click Me` button.

Figure 23. Server Program

Example 1: Compile and Run

The following are the compiler and interpreter commands to compile and run the example. To run the example, start the server program first. If you do not, the client program cannot establish the socket connection.

```
javac SocketServer.java
javac SocketClient.java
java SocketServer
java SocketClient
```

Example 1: Server-Side Program

The `SocketServer` program establishes a socket connection on Port 4321 in its `listenSocket` method. It reads data sent to it and sends that same data back to the server in its `actionPerformed` method.

listenSocket Method

The `listenSocket` method creates a `ServerSocket` object with the port number on which the server program is going to listen for client communications. The port number must be an available port, which means the number cannot be reserved or already in use. For example, UNIX systems reserve ports 1 through 1023 for administrative functions leaving port numbers greater than 1024 available for use.

```
public void listenSocket(){
   try {
      server = new ServerSocket(4321);
   } catch (IOException e) {
      System.out.println("Could not listen on port 4321");
      System.exit(-1);
   }
```

Next, the `listenSocket` method creates a `Socket` connection for the requesting client. This code executes when a client starts and requests the connection on the host and port where this server program is running. When the connection is successfully established, the `server.accept` method returns a new `Socket` object.

```
try{
   client = server.accept();
} catch (IOException e) {
   System.out.println("Accept failed: 4321");
   System.exit(-1);
}
```

Then, the `listenSocket` method creates a `BufferedReader` object to read the data sent over the socket connection from the client program. It also creates a `PrintWriter` object to send the data received from the client back to the server.

```
try{
   in = new BufferedReader(
            new  InputStreamReader(client.getInputStream()));
            out = new PrintWriter(client.getOutputStream(), true);
} catch (IOException e) {
   System.out.println("Read failed");
   System.exit(-1);
}
```

Finally, the `listenSocket` method loops on the input stream to read data as it arrives from the client and writes to the output stream to send the data back.

```
while (true) {
   try {
   line = in.readLine();
   //Send data back to client
   out.println(line);
   } catch (IOException e) {
      System.out.println("Read failed");
      System.exit(-1);
   }
}
```

actionPerformed Method

The `actionPerformed` method is called by the Java platform for action events such as button clicks. This `actionPerformed` method uses the text stored in the `line` object to initialize the `textArea` object so the retrieved text can be displayed to the user.

```
public void actionPerformed(ActionEvent event) {
   Object source = event.getSource();
   if (source == button) {
      textArea.setText(line);
   }
}
```

Example 1: Client-Side Program

The `SocketClient` program establishes a connection to the server program on a particular host and port number in its `listenSocket` method and sends the data entered by the user to the server program in its `actionPerformed` method. The `actionPerformed` method also receives the data back from the server and prints it to the command line.

listenSocket Method

The `listenSocket` method first creates a `Socket` object with the computer name (`kq6py`) and port number (4321) where the server program is listening for client connection requests. Next, it creates a `PrintWriter` object to send data over the socket connection to the server program. It also creates a `BufferedReader` object to read the text sent by the server back to the client.

> **Note:** To make this example work, substitute this host name kq6py with the name of your own machine.

```
public void listenSocket() {
   //Create socket connection
   try {
      socket = new Socket(“kq6py”, 4321);
      out = new PrintWriter(socket.getOutputStream(), true);
      in = new BufferedReader(new InputStreamReader( socket.getInputStream()));
   } catch (UnknownHostException e) {
      System.out.println(“Unknown host: kq6py”);
      System.exit(1);
   } catch (IOException e) {
      System.out.println(“No I/O”);
      System.exit(1);
   }
}
```

actionPerformed Method

The `actionPerformed` method is called by the Java platform for action events such as button clicks. This `actionPerformed` method code gets the text in the `textField` object and passes it to the `PrintWriter` object (`out`), which then sends it over the socket connection to the server program. The `actionPerformed` method then makes the `Textfield` object blank so it is ready for more user input. Lastly, it receives the text sent back to it by the server and prints the text.

```
public void actionPerformed(ActionEvent event) {
   Object source = event.getSource();
   if (source == button) {
   //Send data over socket
      String text = textField.getText();
      out.println(text);
      textField.setText(new String(“”));
      out.println(text);
   }
//Receive text from server
   try {
      String line = in.readLine();
      System.out.println(“Text received: “ + line);
   } catch (IOException e){
      System.out.println(“Read failed”);
      System.exit(1);
   }
}
```

Example 2: Multithreaded Server Example

The example in its current form works between the server program and one client program only. To allow multiple client connections as shown in *Figure 24*, the server program has to be converted to a multithreaded server program.

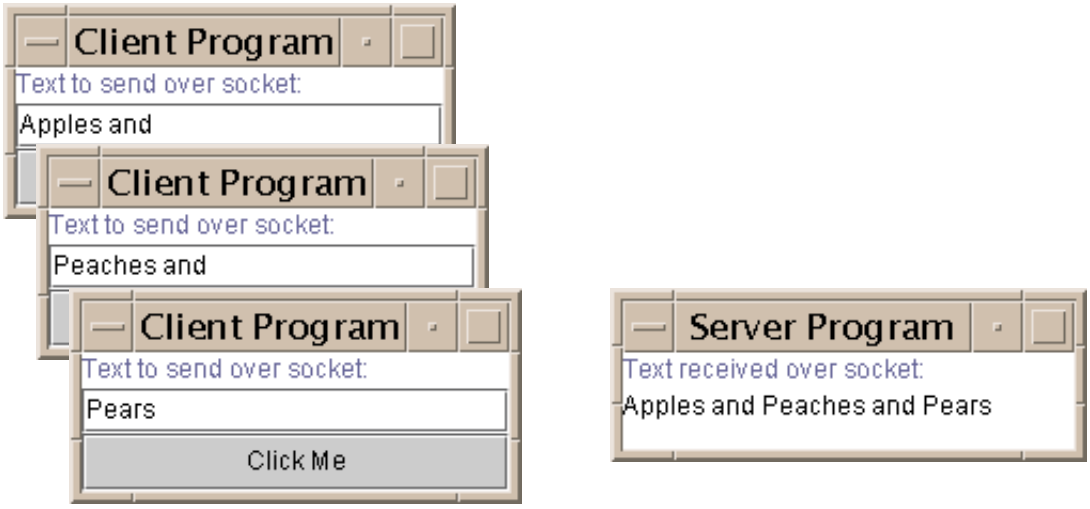


Figure 24. Three Clients Sending Data to One Server Program

The multithreaded server program creates a new thread for every client request. Each client has its own connection to the server for passing data back and forth. When running multiple threads, you have to be sure that one thread cannot interfere with the data in another thread.

In this example the `listenSocket` method loops on the `server.accept` call waiting for client connections and creates an instance of the `ClientWorker` class for each client connection it accepts. The `textArea` component that displays the text received from the client connection is passed to the `ClientWorker` instance with the accepted client connection.

```
public void listenSocket() {
   try {
      server = new ServerSocket(4321);
   } catch (IOException e) {
      System.out.println("Could not listen on port 4321");
      System.exit(-1);
   }
   while (true) {
      ClientWorker w;
      try{
      //server.accept returns a client connection
      w = new ClientWorker(server.accept(), textArea);
      Thread t = new  Thread(w);
      t.start();
```

```
      } catch (IOException e) {
         System.out.println("Accept failed: 4444");
         System.exit(-1);
      }
   }
}
```

The important changes in this version of the server program over the non-threaded server program are that the `line` and `client` variables are no longer instance variables of the server class, but are handled inside the `ClientWorker` class.

The `ClientWorker` class implements the `Runnable` interface, which has one method, `run`. The `run` method executes independently in each thread. If three clients request connections, three `ClientWorker` instances are created, a thread is started for each `ClientWorker` instance, and the `run` method executes for each thread.

In this example, the `run` method creates the input buffer and output writer, loops on the input stream waiting for input from the client, sends the data it receives back to the client, and sets the text in the text area.

```
class ClientWorker implements Runnable {
   private Socket client;
   private JTextArea textArea;
//Constructor
   ClientWorker(Socket client, JTextArea textArea) {
      this.client = client;
      this.textArea = textArea;
   }
public void run() {
   String line;
   BufferedReader in = null;
   PrintWriter out = null;
   try{
      in = new BufferedReader(new InputStreamReader(client.getInputStream()));
      out = new PrintWriter(client.getOutputStream(), true);
   } catch (IOException e) {
      System.out.println("in or out failed");
      System.exit(-1);
   }
   while (true) {
      try {
         line = in.readLine();
         //Send data back to client
         out.println(line);
         textArea.append(line);
      } catch (IOException e) {
```

```
            System.out.println("Read failed");
            System.exit(-1);
         }
      }
   }
}
```

The API documentation for the `JTextArea.append` method states that this method is thread safe, which means its implementation includes code that allows one thread to finish its append operation before another thread can start an append operation. This prevents one thread from overwriting all or part of a string of appended text and corrupting the output.

If the `JTextArea.append` method were not thread safe, you would need to wrap the call to `textArea.append(line)` in a `synchronized` method and replace the call in the `run` method to `textArea.append(line)` with a call to the `appendText(line)` method instead.

The `synchronized` keyword gives this thread a lock on the `textArea` when the `appendText` method is called so no other thread can change the `textArea` until this method completes. *Chapter 12, Develop the Example* uses `synchronized` methods for writing data out to a series of files.

The `finalize` method is called by the JVM before the program exits to give the program a chance to clean up and release resources. Multithreaded programs should close all `Files` and `Sockets` they use before exiting so they do not face resource starvation. The call to `server.close` in the `finalize` method closes the `Socket` connection used by each thread in this program.

```
protected void finalize() {
   //Objects created in run method are finalized when
   //program terminates and thread exits
   try {
      server.close();
   } catch (IOException e) {
      System.out.println(“Could not close socket”);
      System.exit(-1);
   }
}
```

Exercises

1. How do you handle server requests from more than one client?
2. What does it mean when the API documentation states that a method is thread safe?
3. What is the `synchronize` keyword for?
4. What does the `finalize` method do?

Code for This Lesson

- *SocketClient Program*
- *SocketServer Program*
- *SocketThrdServer Program*

SocketClient Program

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

class SocketClient extends JFrame implements ActionListener {
   JLabel text, clicked;
   JButton button;
   JPanel panel;
   JTextField textField;
   Socket socket = null;
   PrintWriter out = null;
   BufferedReader in = null;
   SocketClient(){ //Begin Constructor
      text = new JLabel("Text to send over socket:");
      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();
        out.println(text);
        textField.setText(new String(“”));
        //Receive text from server
        try{
            String line = in.readLine();
            System.out.println(“Text received :” + line);
        } catch (IOException e) {
            System.out.println(“Read failed”);
            System.exit(1);
        }
    }
}

public void listenSocket() {
    //Create socket connection
    try{
        socket = new Socket(“kq6py”, 4444);
        out = new PrintWriter(socket.getOutputStream(), true);
        in = new BufferedReader(
        new InputStreamReader(socket.getInputStream()));
    } catch (UnknownHostException e) {
        System.out.println(“Unknown host: kq6py”);
        System.exit(1);
    } catch  (IOException e) {
        System.out.println(“No I/O”);
        System.exit(1);
    }
}
public static void main(String[] args) {
    SocketClient frame = new SocketClient();
    frame.setTitle(“Client Program”);
    WindowListener l = new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
```

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

SocketServer Program

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

class SocketServer extends JFrame implements ActionListener {
   JButton button;
   JLabel label = new JLabel("Text received over socket:");
   JPanel panel;
   JTextArea textArea = new JTextArea();
   ServerSocket server = null;
   Socket client = null;
   BufferedReader in = null;
   PrintWriter out = null;
   String line;
   SocketServer(){ //Begin Constructor
      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", label);
      panel.add("Center", textArea);
      panel.add("South", button);
   } //End Constructor

   public void actionPerformed(ActionEvent event) {
```

```
    Object source = event.getSource();
    if(source == button) {
       textArea.setText(line);
    }
}
public void listenSocket() {
    try{
       server = new ServerSocket(4444);
    } catch (IOException e) {
       System.out.println("Could not listen on port 4444");
       System.exit(-1);
    }

    try {
       client = server.accept();
    } catch (IOException e) {
       System.out.println("Accept failed: 4444");
       System.exit(-1);
    }

    try {
       in = new BufferedReader(
       new InputStreamReader(client.getInputStream()));
       out = new PrintWriter(client.getOutputStream(), true);
}   catch (IOException e) {
       System.out.println("Accept failed: 4444");
       System.exit(-1);
    }

    while (true) {
       try{
          line = in.readLine();
          //Send data back to client
          out.println(line);
       } catch (IOException e) {
          System.out.println("Read failed");
          System.exit(-1);
       }
    }
}

protected void finalize() {
```

```
   //Clean up
      try {
         in.close();
         out.close();
         server.close();
      } catch (IOException e) {
         System.out.println("Could not close.");
         System.exit(-1);
      }
   }
   public static void main(String[] args) {
      SocketServer frame = new SocketServer();
      frame.setTitle("Server Program");
      WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
         System.exit(0);
         }
      };
      frame.addWindowListener(l);
      frame.pack();
      frame.setVisible(true);
      frame.listenSocket();
   }
}
```

SocketThrdServer Program

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

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

class ClientWorker implements Runnable {
   private Socket client;
   private JTextArea textArea;

   ClientWorker(Socket client, JTextArea textArea) {
      this.client = client;
      this.textArea = textArea;
```

```
   }

   public void run() {
      String line;
      BufferedReader in = null;
      PrintWriter out = null;
      try {
         in = new BufferedReader(new InputStreamReader(client.getInputStream()));
         out = new PrintWriter(client.getOutputStream(), true);
      } catch (IOException e) {
         System.out.println("in or out failed");
         System.exit(-1);
   }

      while (true) {
         try {
            line = in.readLine();
            //Send data back to client
            out.println(line);
            textArea.append(line);
         } catch (IOException e) {
            System.out.println("Read failed");
            System.exit(-1);
         }
      }
   }
}

class SocketThrdServer extends JFrame{

   JLabel label = new JLabel("Text received over socket:");
   JPanel panel;
   JTextArea textArea = new JTextArea();
   ServerSocket server = null;

   SocketThrdServer(){ //Begin Constructor
      panel = new JPanel();
      panel.setLayout(new BorderLayout());
      panel.setBackground(Color.white);
      getContentPane().add(panel);
      panel.add("North", label);
      panel.add("Center", textArea);
```

```
} //End Constructor

public void listenSocket() {
   try {
      server = new ServerSocket(4444);
   } catch (IOException e) {
      System.out.println("Could not listen on port 4444");
      System.exit(-1);
   }
   while (true) {
      ClientWorker w;
      try{
         w = new ClientWorker(server.accept(), textArea);
         Thread t = new Thread(w);
         t.start();
      } catch (IOException e) {
         System.out.println("Accept failed: 4444");
         System.exit(-1);
      }
   }
}

protected void finalize(){
   //Objects created in run method are finalized when
   //program terminates and thread exits
   try {
      server.close();
   } catch (IOException e) {
      System.out.println("Could not close socket");
      System.exit(-1);
   }
}

public static void main(String[] args) {
   SocketThrdServer frame = new SocketThrdServer();
   frame.setTitle("Server Program");
   WindowListener l = new WindowAdapter() {
   public void windowClosing(WindowEvent e) {
      System.exit(0);
      }
   };
   frame.addWindowListener(l);
```

```
        frame.pack();
        frame.setVisible(true);
        frame.listenSocket();
    }
}
```

Object-Oriented Programming 10

You already know about object-oriented programming because after working the examples in this tutorial, you are familiar with the object-oriented concepts of class, object, instance, and inheritance plus the access levels `public` and `private`. Essentially, you have been doing object-oriented programming without thinking about it.

One of the great things about Java is that its design supports the object oriented model. To help you gain a better understanding of object-oriented programming and its benefits, this lesson presents a very brief overview of object-oriented concepts and terminology as they relate to the example code presented in this tutorial. You will find pointers to good books on object-oriented programming at the end of this chapter.

This lesson covers the following topics:

Object-Oriented Programming

Java supports object-oriented programming techniques that are based on a hierarchy of classes and well-defined and cooperating objects.

Classes

A class is a structure that defines the data and the methods to work on that data. When you write programs in Java, all program data is wrapped in a class, whether it is a class you write or a class you use from the Java API libraries.

For example, the `ExampleProgram` class from *Chapter 1, Compile and Run a Simple Program* is a programmer-written class that uses the `java.lang.System` class from the Java API libraries to print a string literal (character string) to the command line.

```
class ExampleProgram {
   public static void main(String[] args) {
      System.out.println(“I’m a simple Program”);
   }
}
```

Classes in the Java API libraries define a set of objects that share a common structure and behavior. The `java.lang.System` class used in the example provides access to standard input, output, error streams, access to system properties, and more. In contrast, the `java.lang.String` class defines string literals.

In the `ExampleProgram` class example, you do not see an explicit use of the `String` class, but in Java, a string literal can be used anywhere a method expects to receive a `String` object.

During execution, the Java platform creates a `String` object from the character string passed to the `System.out.println` call, but your program cannot call any of the `String` class methods because it did not instantiate the `String` object. If you want access to the `String` methods, rewrite the example program to create a `String` object. The `String.concat` method shown below adds text to the original string.

```
class ExampleProgram {
   public static void main(String[] args){
      String text = new String(“I’m a simple Program “);
      System.out.println(text);
      String text2 = text.concat(“that uses classes and objects”);
      System.out.println(text2);
   }
}
```

The output looks like this:

```
I’m a simple Program
I’m a simple Program that uses classes and objects
```

Objects

An instance is a synonym for object. A newly created instance has data members and methods as defined by the class for that instance. In the last example, various `String` objects are created for the concatenation operation.

Because `String` objects cannot be edited, the `java.lang.String.concat` method converts the `String` objects to editable `StringBuffer` string objects to do the concatenation. In addition to the `String` object, there is an instance of the `ExampleProgram` class in memory.

Well-Defined Boundaries and Cooperation

Class definitions must allow objects to cooperate during execution. In the previous section, you saw how the `System`, `String`, and `StringBuffer` objects cooperated to print a concatenated character string to the command line.

This section changes the example program to display the concatenated character string in a `JLabel` component in a user interface to further illustrate the concepts of well-defined class boundaries and object cooperation. The program code uses a number of cooperating classes. Each class has its own purpose as summarized below, and where appropriate, the classes are defined to work with objects of another class.

- `ExampleProgram` defines the program data and methods to work on that data.
- `JFrame` defines the top-level window including the window title and frame menu.
- `WindowListener` defines behavior for (works with) the Close option on the frame menu.
- `String` defines a character string passed to the label.
- `JLabel` defines a user interface component to display non-editable text.
- `JPanel` defines a container and uses the default layout manager (`java.awt.FlowLayout`) to lay out the objects it contains.

While each class has its own specific purpose, they all work together to create the simple user interface you see in *Figure 25*.

Figure 25. Simple User Interface

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

class ExampleProgram extends JFrame {
   public ExampleProgram() {
      String text = new String(“I’m a simple Program “);
```

```
   String text2 = text.concat("that uses classes and objects");
   JLabel label = new JLabel(text2);
   JPanel panel = new JPanel();
   panel.setBackground(Color.white);
   getContentPane().add(panel);
   panel.add(label);   }
public static void main(String[] args){
   ExampleProgram frame = new ExampleProgram();
   frame.setTitle("Fruit $1.25 Each");
   WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
         System.exit(0);
         }
      };
      frame.addWindowListener(l);
      frame.pack();
      frame.setVisible(true);
   }
}
```

Inheritance and Polymorphism

One object-oriented concept that helps objects work together is inheritance. Inheritance defines relationships among classes in an object-oriented language. The relationship is one of parent to child where the child or extending class inherits all the attributes (methods and data) of the parent class. In Java, all classes descend from `java.lang.Object` and inherit its methods.

Figure 26 shows the class hierarchy as it descends from `java.lang.Object` for the classes in the user interface example above. The `java.lang.Object` methods are also shown because they are inherited and implemented by all of its subclasses, which is every class in the Java API libraries. `java.lang.Object` defines the core set of behaviors that all classes have in common.

As you move down the hierarchy, each class adds its own set of class-specific fields and methods to what it inherits from its superclass. The `java.awt.swing.JFrame` class inherits fields and methods from `java.awt.Frame`, which inherits fields and methods from `java.awt.Container`, which inherits fields and methods from `java.awt.Component`, which finally inherits from `java.lang.Object`, and each subclass adds its own fields and methods as needed.

Each class in the hierarchy adds its own class-specific behavior to inherited methods. This way different classes with a common parent can have like-named methods that exhibit behavior appropriate to each class. For example, the `Object` class has a `toString` method inherited by all its subclasses. You can call the `toString` method on any class to get its string representation, but the actual behavior of the `toString` method in each class

depends on the class of the object on which it is invoked.

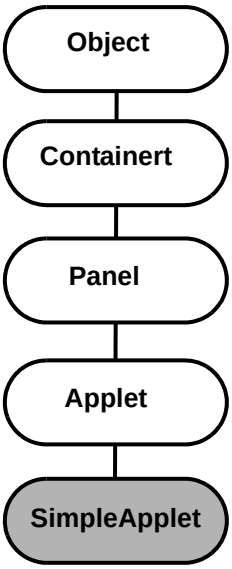


Figure 26. Object Hierarchy

Another way objects work together is to define methods that take other objects as parameters. For example, if you define a method that takes a `java.lang.Object` as a parameter, it can accept any object in the entire Java platform. If you define a method that takes a `java.awt.Component` as a parameter, it can accept any class that derives from `Component`. This form of cooperation is called *polymorphism*.

You saw an example of polymorphism in *Maintain and Display a Customer List* . A `Collection` object can contain any type of object as long as it descends from `java.lang.Object`. The code is repeated here to show you that a `HashSet` collection can add a `String` object and an `Integer` object to the set because the `HashSet.add` method is defined to accept any class instance that traces back to the `java.lang.Object` class.

Polymorphism lets you call `Set s = new HashSet()` and return an object of type `Set` even though a `Set` is an interface with no implementation. This is possible because the `HashSet` class implements the `Set` interface, and so this statement actually returns an object of type `HashSet`.

```
String custID = “munchkin”;
Integer creditCard =  new Integer(25);
Set s = new HashSet();
s.add(custID);
s.add(creditCard);
```

Data Access Levels

Another way classes work together is through access level controls. Classes, and their fields and methods have access levels to specify how they can be used by other objects during execution, While cooperation among objects is desirable, there are times when you will want to explicitly control access, and specifying access levels is the way to gain that control. When you do not specify an access level, the default access level is in effect.

Classes

Classes can be declared `package`, `public`, `private`, or `protected`. If no access level is specified, the class is `package` by default.

`package`: The class can be used only by instances of other classes in the same package.

`public`: A class can be declared `public` to make it accessible to all class instances regardless of what package its class is in. You might recall that in *Chapter 3, Building Applets*, the pallet class had to be declared `public` so it could be accessed by the `appletviewer` tool because the appletviewer program is created from classes in another package. Also, in *Chapter 13, Internationalization*, the server classes are made `public` so client classes can access them.

`private`: A class declared `private` cannot be instantiated by any other class. Usually `private` classes have `public` methods called factory methods than can be called by other classes. These `public` factory methods create (manufacture) an instance of the class and return it to the calling method.

`protected`: Only subclasses of a `protected` class can create instances of it.

Fields and Methods

Fields and methods can be declared `private`, `protected`, or `public`. If no access level is specified, the field or method access level is `package` by default.

`private`: A private field or method is accessible only to the class in which it is defined. In *Chapter 7, Database Access and Permissions*, the connection, user name, and password for establishing the database access are all private. This is to prevent an outside class from accessing them and jeopardizing the database connection, or compromising the secret user name and password information.

`protected`: A protected field or method is accessible to the class itself, its subclasses, and classes in the same package.

`public`: A public field or method is accessible to any class of any parentage in any package. In *Chapter 13* server data accessed by client programs is made public.

`package`: A package field or method is accessible to other classes in the same package.

Global Variables and Methods

Java does not have global variables and methods because all fields are wrapped in a class and all classes are part of a package. To reference a field or method, you use the package, class, and field or method name.

However, fields declared `public static` can be accessed and changed by any instance regardless of parentage or package (similar to global variable data). If the field is declared `final public static`, its value can never be changed, which makes it similar to a global constant.

Your Own Classes

When you use the Java API classes, they have already been designed with the above concepts in mind. They all descend from `java.lang.Object` giving them an inheritance relationship; they have well-defined boundaries; and they are designed to cooperate with each other where appropriate.

For example, you will not find a `String` class that takes an `Integer` object as input because that goes beyond the well-defined boundary for a `String`. You will, however, find the `Integer` class has a method for converting its integer value to a `String` so its value can be displayed in a user interface component, which only accepts `String` objects.

But when you write your own classes, how can you be sure your classes have well-defined boundaries, cooperate, and make use of inheritance? One way is to look at what a program needs to do and separate those operations into distinct modules where each operational module is defined by its own class or group of classes.

Well-Defined Boundaries and Cooperation

Looking at the `RMIClient2` class from *Chapter 12, Develop the Example*, you can see it performs the following operations: Get data, display data, store customer IDs, print customer IDs, and reset the display.

Getting data, displaying the data, and resetting the display are closely related and easily form an operational module. But in a larger program with more data processing, the storing and printing of customer IDs could be expanded to store and print a wider range of data. In such a case, it would make sense to have a separate class for storing data, and another class for printing it in various forms.

You could, for example, have a class that defines how to store customer IDs, and tracks the number of apples, peaches, and pears sold during the year. You could also have another class that defines report printing. It could access the stored data to print reports on apples, peaches, and pears sold by the month, per customer, or throughout a given season.

Making application code modular by separating out operational units makes it easier to update and maintain the source code. When you change a class, as long as you did not change any part of its public interface, you only have to recompile that one class.

Inheritance

Deciding what classes your program needs means separating operations into modules, but making your code more efficient and easier to maintain means looking for common operations where you can use inheritance. If you need to write a class that does similar things to a class in the Java API libraries, it makes sense to extend that API library class and use its methods rather than write everything from scratch.

The `RMIClient2` class from *Chapter 12* extends `JFrame` to leverage the ready-made behavior it provides for a program's top-level window including, frame menu closing behavior, background color setting, and a customized title.

Likewise, to add customized behavior to an existing class, extend that class and add the behavior you want. For example, you might want to create a custom `Exception` to use in your program. To do this, write an exception class that extends `java.lang.Exception` and implement it to do what you want.

Access Levels

You should always keep access levels in mind when you declare classes, fields, and methods. Consider which objects really need access to the data, and use packages and access levels to protect your application data from all other objects executing in the system.

Most object-oriented applications declare their data fields `private` so other objects cannot access their fields directly and make the methods that access the `private` data protected, `public`, or `package` as needed so other objects can manipulate their private data by calling the methods only.

Keeping data private gives an object the control to maintain its data in a valid state. For example, a class can include behavior for verifying that certain conditions are true before and after the data changes. If other objects can access the data directly, it is not possible for an object to maintain its data in a valid state like this.

Another reason to keep data fields private and accessible only through the class methods is to make it easier to maintain source code. You can update your class by changing a field definition and the corresponding method implementation, but other objects that access that data do not need to be changed because their interface to the data (the method signature) has not changed.

Exercises

Exercises for this lesson include setting the correct access levels and organizing a program into functional units.

Setting Access Levels

So that an object has the control it needs to maintain its data in a valid state, it is always best to restrict access as much as possible. Going back to *Chapter 14, Packages and JAR File Format*, the server classes had to be made `public` and the `DataOrder` class fields also had to be made `public` so the client programs could access them. At that time, no access level was specified for the other classes and fields so they are all `package` by default, and all methods have an access level of `public`.

A good exercise is to go back to the client classes from *Chapter 13* and give the classes, fields, and methods an access level so they are not accessed inappropriately by other objects. You will find solutions for the `RMIClient1` and `RMIClient2` client programs in *Appendix A, RMIClient1*.

Organizing Code into Functional Units

One way to divide code into functional units is to put the user interface code in one class and the code that responds to user interface interactions in another class.

Go back to *Chapter 13*, and move the `actionPerformed` method in the `RMIClient1` class into its own class and file.

Hints:

- When you make the *RMIClient1* class into two classes, keep the following in mind:
- The class that builds the UI has a *main* method; the second class does not.
- The class with the *main* method creates an instance of the second class, and passes an instance of itself to the second class.
- The second class accesses user interface components in the first class by referencing the class instance.
- You will need to work out how the second class will get access to the message bundle with the translated text in it.
- You will have to change some access levels so the classes can access each other's members.

You can find a possible solution in *Appendix A, Code Listings*.

User Interfaces Revisited

11

In *Chapter 4, Building a User Interface* , you learned how to use Project Swing components to build a simple user interface with very basic backend functionality. *Chapter 8, Remote Method Invocation* also showed you how to use the RMI API to send data from a client program to a server program on the net where the data can be accessed by other client programs.

This lesson takes the RMI application from he *Chapter 8*, creates a more complex user interface and data model, and uses a different layout manager. These changes provide the beginnings of a very simple electronic-commerce application that consists of two types of client programs: one lets users place purchase orders, and the other lets order processors view the orders.

This lesson covers the following topics:

- *About the Example*
- *Exercises*
- *Code for This Lesson*

About the Example

This is a very simple electronic commerce example for instructional purposes only. It has three programs: two client programs, one for ordering fruit and another for viewing the fruit order, and one server program that is a repository for the order information.

The fruit order data is wrapped in a single data object defined by the `DataOrder` class. The fruit order client retrieves the data from the user interface components where the user enters it, stores the data in a `DataOrder` instance, and sends the `DataOrder` instance to the server program. The view order client retrieves the `DataOrder` instance from the server, gets the data out of the `DataOrder` instance, and displays the retrieved data in its user interface.

Fruit Order Client (RMIClient1)

The `RMIClient1` program shown in *Figure 27* presents a user interface and prompts the user to order apples, peaches, and pears at $1.25 each.

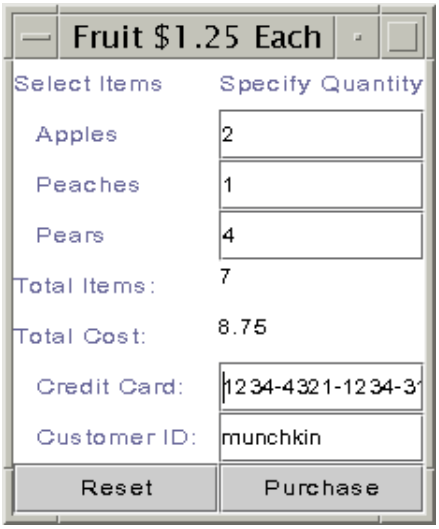


Figure 27. Fruit Order Client Program

After the user enters the number of each item to order, he or she presses the Return key to commit the order and update the running total. The Tab key or mouse moves the cursor to the next field. At the bottom, the user provides a credit card number and customer ID. When the user selects the `Purchase` button, all values entered into the form are stored in a `DataOrder` instance, which is then sent to the server program.

The user must press the Return key for the total to update. If the Return key is not pressed, an incorrect total is sent across the net with the order. The *Exercises* for this lesson ask you to change the code so incorrect totals are not sent across the net because the user did not press the Return key.

Server Program

The `Send` interface and `RemoteServer` class have one `getOrder` method to return an instance of `DataOrder`, and one `setOrder` method to accept an instance of `DataOrder`. The fruit order clients call the `send` method to send data to the server, and view order clients call the `get` method to retrieve the data. In this example, the server program has no user interface.

View Order Client (RMIClient2)

The `RMIClient2` program shown in *Figure 28* presents a user interface, and when the user clicks `View Order`, the program gets a `DataOrder` instance from the server program, retrieves its data, and displays the data on the screen.

Figure 28. View Order Client Program

Compile and Run the Example

The code for this lesson is compiled and run the same way as the code in *Program Behavior*. This summarized version includes steps to handle the `DataOrder` class introduced in this lesson.

Compile

UNIX:

```
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
cp DataOrder.class /home/zelda/public_html/classes
```

Win32:

```
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
copy DataOrder.class \home\zelda\public_html\classes
```

Start the RMI Registry

UNIX:

```
unsetenv CLASSPATH
rmiregistry  &
```

Win32:

```
set CLASSPATH=
start rmiregistry
```

Start the Server

UNIX:

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

Win32:

```
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
```

Start the RMIClient1 Program

UNIX:

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

Win32:

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

Start the RMIClient2 Program

UNIX:

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

Win32:

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

Fruit Order (RMIClient1) Code

The `RMIClient1` code uses the label, text field, text area, and button components shown in *Figure 29* to create the user interface for ordering fruit.

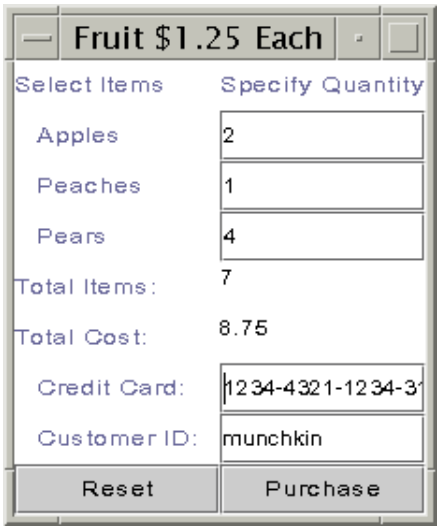


Figure 29. User Interface Components

On the display, the user interface components are arranged in a 2-column grid with labels in the left column, and the input and output data fields (text fields and text areas) aligned in the right column.

The user enters his or her apples, peaches, and pears order into the text fields and presses the Return key after each fruit entry. When the Return key is pressed, the text field behavior updates the item and cost totals displayed in the text areas.

The `Reset` button clears the display and the underlying variables for the total cost and total items. The `Purchase` button sends the order data to the server program. If the `Reset` button is clicked before the `Purchase` button, `null` values are sent to the server.

Instance Variables

These next lines declare the Project Swing component classes the `RMIClient1` class uses. These instance variables can be accessed by any method in the instantiated class. In this example, they are built in the constructor and accessed in the `actionPerformed` method implementation.

```
JLabel col1, col2;
JLabel totalItems, totalCost;
JLabel cardNum, custID;
JLabel applechk, pearchk, peachchk;
JButton purchase, reset;
JPanel panel;
JTextField appleqnt, pearqnt, peachqnt;
JTextField creditCard, customer;
JTextArea items, cost;
static Send send;
int itotal=0;
double icost=0;
```

Constructor

The constructor is fairly long because it creates all the components, sets the layout to a 2-column grid, and places the components in the grid on a panel.

The `Reset` and `Purchase` buttons and the `appleqnt`, `pearqnt`, and `peachqnt` text fields are added to the `RMIClient1` object so the `RMIClient1` object will listen for action events from these components. When the user clicks one of the buttons or presses Return in a text field, an action event causes the platform to call the `RMIClient1.actionPerformed` method where the behaviors for these components is defined.

Chapter 4 explains how a class declares the `ActionListener` interface and implements the `actionPerformed` method if it needs to handle action events such as button clicks and text field Returns. Other user interface components generate different action events, and as a result, require you to declare different interfaces and implement different methods.

```
//Create left and right column labels
col1 = new JLabel("Select Items");
col2 = new JLabel("Specify Quantity");

//Create labels and text field components
applechk = new JLabel("   Apples");
appleqnt = new JTextField();
appleqnt.addActionListener(this);

pearchk = new JLabel("   Pears");
```

```
pearqnt = new JTextField();
pearqnt.addActionListener(this);

peachchk = new JLabel("   Peaches");
peachqnt = new JTextField();
peachqnt.addActionListener(this);

cardNum = new JLabel("   Credit Card:");
creditCard = new JTextField();
customer = new JTextField();
custID = new JLabel("   Customer ID:");

//Create labels and text area components
totalItems = new JLabel("Total Items:");
totalCost = new JLabel("Total Cost:");
items = new JTextArea();
cost = new JTextArea();

//Create buttons and make action listeners
purchase = new JButton("Purchase");
purchase.addActionListener(this);
reset = new JButton("Reset");
reset.addActionListener(this);
```

In the next lines, a `JPanel` component is created and added to the top-level frame, and the layout manager and background color for the panel are specified. The layout manager determines how user interface components are arranged on the panel.

The example in *Chapter 4*, used the `BorderLayout` layout manager. This example uses the `GridLayout` layout manager, which arranges components in a grid using the number of rows and columns you specify. The example uses a 2-column grid with an unlimited number of rows as indicated by the zero (unlimited rows) and two (two columns) in the statement `panel.setLayout(new GridLayout(0,2))`. Components are added to a panel using `GridLayout` going across and down.

The layout manager and color are set on the panel, and the panel is added to the content pane with a call to the `getContentPane` method of the `JFrame` class. A content pane enables different types of components to work together in Project Swing.

```
//Create a panel for the components
panel = new JPanel();

//Set panel layout to 2-column grid
//on a white background
panel.setLayout(new GridLayout(0,2));
panel.setBackground(Color.white);
```

```
//Add components to panel columns
//going left to right and top to bottom
getContentPane().add(panel);
panel.add(col1);
panel.add(col2);
panel.add(applechk);
panel.add(appleqnt);
panel.add(peachchk);
panel.add(peachqnt);
panel.add(pearchk);
panel.add(pearqnt);
panel.add(totalItems);
panel.add(items);
panel.add(totalCost);
panel.add(cost);
panel.add(cardNum);
panel.add(creditCard);
panel.add(custID);
panel.add(customer);
panel.add(reset);
panel.add(purchase);
```

Event Handling

The `actionPerformed` method provides behavior when the `Purchase` or `Reset` button is clicked, or the Return key is pressed in the `appleqnt`, `peachqnt`, or `pearqnt` text fields. The `Reset` button is similar to the `purchase` button, and the other text fields are similar to `appleqnt`, so this section will focus on the `Purchase` button, `appleqnt` text field, and the `DataOrder` class.

The `actionPerformed` method in the `RMIClient1` class retrieves the event, declares its variables, and creates an instance of the `DataOrder` class.

```
public void actionPerformed(ActionEvent event) {
   Object source = event.getSource();
   Integer applesNo, peachesNo, pearsNo, num;
   Double cost;
   String number, text, text2;
   DataOrder order = new DataOrder();
```

The `DataOrder` class defines fields that wrap and store the fruit order data. As you can see by its class definition below, the `DataOrder` class has no methods. It does, however, implement the `Serializable` interface.

An object created from a class that implements the `Serializable` interface can be serialized. Object serialization transforms an object's data to a byte stream that represents the state of the data. The serialized form of the data contains enough information so the receiving program can create an object with its data in the same state to what it was when first serialized.

The RMI API uses object serialization to send data over the network, and many Java API classes are serializable. So if you use RMI to send, for example, a `JTextArea` over the network everything should work. But in this example, a special `DataOrder` object class was created to wrap the data, and so this `DataObject` class has to be serializable so the RMI services can serialize the data and send it between the client and server programs.

```
import java.io.*;

class DataOrder implements Serializable{
   String apples, peaches, pears, cardnum, custID;
   double icost;
   int itotal;
}
```

Purchase Button

The `Purchase` button behavior involves retrieving data from the user interface components, initializing the `DataOrder` instance, and sending the `DataOrder` instance to the server program.

```
if (source == purchase) {
   order.cardnum = creditCard.getText();
   order.custID = customer.getText();
   order.apples = appleqnt.getText();
   order.peaches = peachqnt.getText();
   order.pears = pearqnt.getText();
   order.itotal = itotal;
   order.icost = icost;
   //Send data over net
   try {
      send.sendOrder(order);
   } catch (java.rmi.RemoteException e) {
      System.out.println("Cannot send data to server");
   }
}
```

appleqnt Text Field

The appleqnt text field behavior involves retrieving the number of pears the user wants to order, adding the number to the items total, using the number to calculate the cost, and adding the cost for pears to the total cost.

```
//If Return in apple quantity text field
//Calculate totals
if (source == appleqnt) {
   number = appleqnt.getText();
   if (number.length() > 0) {
      applesNo = Integer.valueOf(number);
      itotal += applesNo.intValue();
   } else {
      /* else no need to change the total */
   }
}
```

The total number of items is retrieved from the `itotal` variable and displayed in the UI.

```
num = new Integer(itotal);
text = num.toString();
this.items.setText(text);
```

Similarly, the total cost is calculated and displayed in the user interface using the `icost` variable.

```
icost = (itotal * 1.25);
cost = new Double(icost);
text2 = cost.toString();
this.cost.setText(text2);
```

Note: The `cost` text area is referenced as `this.cost` because the `actionPerformed` method has a `cost` variable of type `Double`. To reference the instance text area and not the local `Double` by the same name, you have to reference it as `this.cost`.

Cursor Focus

Users can use the Tab key to move the cursor from one component to another within the user interface. The default Tab key movement steps through all user interface components including the text areas in the order they were added to the panel.

The example program has a constructor call to `pearqnt.setNextFocusableComponent` to make the cursor move from the `pearqnt` text field to the `creditcard` text field bypassing the total cost and total items text areas when the Tab key is pressed.

```
cardNum = new JLabel(“ Credit Card:”);
creditCard = new JTextField();
//Make cursor go to creditCard component
pearqnt.setNextFocusableComponent(creditCard);
```

Converting Strings to Numbers and Back

To calculate the items ordered and their cost, the string values retrieved from the `appleqnt`, `peachqnt`, and `pearqnt` text fields have to be converted to their number equivalents.

The string value is returned in the `number` variable. To be sure the user actually entered a value, the string length is checked. If the length is not greater than zero, the user pressed Return without entering a value. In this case, the `else` statement does nothing.

If the length is greater than zero, an instance of the `java.lang.Integer` class is created from the string. Next, the `Integer.intValue` method is called to produce the integer (`int`) equivalent of the string value so it can be added to the items total kept in the `itotal` integer variable.

```
if (number.length() > 0) {
   pearsNo = Integer.valueOf(number);
   itotal += pearsNo.intValue();
} else {
   /* else no need to change the total */
}
```

To display the running item and cost totals in their respective text areas, the totals have to be converted back to strings. The code at the end of the `actionPerformed` method and shown next does this.

To display the total items, a `java.lang.Integer` object is created from the `itotal` integer variable. The `Integer.toString` method is called to produce the `String` equivalent of the integer (`int`). This string is passed to the call to `this.cost.setText(text2)` to update the Total Cost field in the display.

```
num = new Integer(itotal);
text = num.toString();
this.items.setText(text);
icost = (itotal * 1.25);
cost = new Double(icost);
text2 = cost.toString();
this.cost.setText(text2);
```

Until now, all data types used in the examples have been classes. But, the `int` and `double` data types are not classes. They are primitive data types.

The `int` primitive type contains a single whole 32-bit integer value that can be positive or negative. Use the standard arithmetic operators (+, -, *, and /) to perform arithmetic operations on the integer. The `Integer` class also provides methods for working on the value. For example, the `Integer.intValue` method lets you convert an `Integer` to an `int` to perform arithmetic operations.

The `double` primitive type contains a 64-bit double-precision floating point value. The `Double` class also provides methods for working on the value. For example, the `Double.doubleValue` method lets you convert a `Double` to a `double` to perform arithmetic operations.

Server Program Code

The server program consists of the `RemoteServer` class that implements the `get` and `set` methods declared in the `Send` interface. Data of any type and size can be passed from one client through the server to another client using the RMI API. No special handling is needed for large amounts of data or special considerations for different data types, which can sometimes be issues when using socket communications.

Send Interface

The server program is available to the `RMIClient1` program through its `Send` interface, which declares the remote server send and get methods.

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

public interface Send extends Remote {
   public void sendOrder(DataOrder order) throws RemoteException;
   public DataOrder getOrder() throws RemoteException;
}
```

RemoteServer Class

The `RemoteServer` class implements the methods declared by the `Send` interface.

```
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;

class RemoteServer extends UnicastRemoteObject implements Send {
   private DataOrder order;
   public RemoteServer() throws RemoteException {
      super();
   }
   public void sendOrder(DataOrder order) {
      this.order = order;
   }
   public DataOrder getOrder() {
      return this.order;
   }

   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");
        }
    }
}
```

View Order Client (RMIClient2) Code

The *RMIClient2 Program* uses text areas and buttons to display order information.

The code is very similar to the `RMIClient1` class so this section explains how the order data is retrieved.

The first lines retrieve the credit card number, the number of apples, peaches, and pears ordered from the server program, and sets those values in the corresponding text areas.

The last lines retrieve the cost and item totals, which are `double` and `integer`, respectively. It then converts the total cost to a `java.lang.Double` object, the total items to a `java.lang.Integer` object, and calls the `toString` method on each to get the string equivalents. Finally, the strings can be used to set the values for the corresponding text areas.

```
DataOrder order = new DataOrder();
if (source == view) {
   try {
      order = send.getOrder();
      creditNo.setText(order.cardnum);
      customerNo.setText(order.custID);
      applesNo.setText(order.apples);
      peachesNo.setText(order.peaches);
      pearsNo.setText(order.pears);
      cost = order.icost;
      price = new Double(cost);
      unit = price.toString();
      icost.setText(unit);
      items = order.itotal;
      itms = new Integer(items);
      i = itms.toString();
itotal.setText(i);
```

Exercises

The example program has been kept simple for instructional purposes, and would need a number of improvements to be an enterprise-worthy application. These exercises ask you to improve the program in several ways.

Calculations and Pressing Return

If the user enters a value for apples, peaches, or pears and moves to the next field without pressing the Return key, no calculation is made. When the user clicks the Purchase key, the order is sent, but the item and cost totals are incorrect. So, in this application, relying on the Return key action event is not good design. Modify the `actionPerformed` method so this does not happen. You will find one way to modify it in *RMIClient1 Program*.

Non-Number Errors:

If the user enters a non-number value for apples, peaches, or pears, the program presents a stack trace indicating an illegal number format. A good program will catch and handle the error, rather than produce a stack trace. See *RMIClient1 Improved Program* for one way to modify the RMIClient1 code..

> **Note:** Find where the code throws an error and use a `try` and `catch` block. The error is `java.lang.NumberFormatException`.

Extra Credit

RMIClient2 Program produces a stack trace if it gets null data from the `DataOrder` object and tries to use it to set the text on the text area component. Add code to test for null fields in the `DataOrder` object and supply an alternative value in the event a null field is found. No solution for this exercise is provided.

If someone enters 2 apples and 2 pears, then decides they want 3 apples, the calculation produces a total of 7 items at $8.75 when it should be 5 items at $6.25. See if you can fix this problem. No solution for this exercise is provided.

The `DataOrder` class should really handle apples, peaches, pears, cardnum, and custID as integers instead of strings. This not only makes more sense, but dramatically reduces the packet size when the data is sent over the net. Change the `DataOrder` class to handle this information as integers. You will also need to change some code in the `RMIClient1` and `RMIClient2` classes because the user interface components handle this data as text. No solution for this exercise is provided.

Code for This Lesson

- *RMIClient1 Program*
- *RMIClient2 Program*
- *RMIClient1 Improved Program*

RMIClient1 Program

```
import java.awt.Color;
import java.awt.GridLayout;
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 col1, col2;
   JLabel totalItems, totalCost;
   JLabel cardNum, custID;
   JLabel applechk, pearchk, peachchk;
   JButton purchase, reset;
   JPanel panel;
   JTextField appleqnt, pearqnt, peachqnt;
   JTextField creditCard, customer;
```

```
JTextArea items, cost;
static Send send;
int itotal=0;
double icost=0;
RMIClient1(){ //Begin Constructor

//Create left and right column labels
   col1 = new JLabel(“Select Items”);
   col2 = new JLabel(“Specify Quantity”);

//Create labels and text field components
   applechk = new JLabel(“   Apples”);
   appleqnt = new JTextField();
   appleqnt.addActionListener(this);
   pearchk = new JLabel(“   Pears”);
   pearqnt = new JTextField();
   pearqnt.addActionListener(this);
   peachchk = new JLabel(“   Peaches”);
   eachqnt = new JTextField();
   peachqnt.addActionListener(this);
   cardNum = new JLabel(“   Credit Card:”);
   creditCard = new JTextField();
   pearqnt.setNextFocusableComponent(creditCard);
   customer = new JTextField();
   custID = new JLabel(“   Customer ID:”);

//Create labels and text area components
   totalItems = new JLabel(“Total Items:”);
   totalCost = new JLabel(“Total Cost:”);
   items = new JTextArea();
   cost = new JTextArea();

//Create buttons and make action listeners
   purchase = new JButton(“Purchase”);
   purchase.addActionListener(this);
   reset = new JButton(“Reset”);
   reset.addActionListener(this);

//Create a panel for the components
   panel = new JPanel();

//Set panel layout to 2-column grid
```

```
//on a white background
   panel.setLayout(new GridLayout(0,2));
   panel.setBackground(Color.white);

//Add components to panel columns
//going left to right and top to bottom
   getContentPane().add(panel);
   panel.add(col1);
   panel.add(col2);
   panel.add(applechk);
   panel.add(appleqnt);
   panel.add(peachchk);
   panel.add(peachqnt);
   panel.add(pearchk);
   panel.add(pearqnt);
   panel.add(totalItems);
   panel.add(items);
   panel.add(totalCost);
   panel.add(cost);
   panel.add(cardNum);
   panel.add(creditCard);
   panel.add(custID);
   panel.add(customer);
   panel.add(reset);
   panel.add(purchase);
} //End Constructor

public void actionPerformed(ActionEvent event) {
   Object source = event.getSource();
   Integer applesNo, peachesNo, pearsNo, num;
   Double cost;
   String number, text, text2;
   DataOrder order = new DataOrder();

   //If Purchase button pressed . . .
   if (source == purchase) {
   //Get data from text fields
   order.cardnum = creditCard.getText();
   order.custID = customer.getText();
   order.apples = appleqnt.getText();
   order.peaches = peachqnt.getText();
   order.pears = pearqnt.getText();
```

```
    order.itotal = itotal;
    order.icost = icost;
    try{
       //Send data over net
       send.sendOrder(order);
    } catch (java.rmi.RemoteException e) {
       System.out.println("Cannot send data to server");
    }
}

//If Reset button pressed
//Clear all fields
    if (source == reset) {
       creditCard.setText(“”);
       appleqnt.setText(“”);
       peachqnt.setText(“”);
       pearqnt.setText(“”);
       creditCard.setText(“”);
       customer.setText(“”);
       icost = 0;
       itotal = 0;
    }

    //If Return in apple quantity text field
    //Calculate totals
    if (source == appleqnt) {
       number = appleqnt.getText();
       if (number.length() > 0) {
          applesNo = Integer.valueOf(number);
          itotal += applesNo.intValue();
       } else {
          /* else no need to change the total */
       }
    }

    //If Return in peach quantity text field
    //Calculate totals
    if (source == peachqnt) {
       number = peachqnt.getText();
       if (number.length() > 0) {
          peachesNo = Integer.valueOf(number);
          itotal += peachesNo.intValue();
```

```
      } else {
         /* else no need to change the total */
      }
   }

   //If Return in pear quantity text field
   //Calculate totals
   if (source == pearqnt){
      number = pearqnt.getText();
      if (number.length() > 0){
         pearsNo = Integer.valueOf(number);
         itotal += pearsNo.intValue();
      } else {
         /* else no need to change the total */
      }
   }
   num = new Integer(itotal);
   text = num.toString();
   this.items.setText(text);
   icost = (itotal * 1.25);
   cost = new Double(icost);
   text2 = cost.toString();
   this.cost.setText(text2);
}

public static void main(String[] args){
   RMIClient1 frame = new RMIClient1();
   frame.setTitle("Fruit $1.25 Each");
   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”);
         }
      }
   }
```

RMIClient2 Program

```
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

class RMIClient2 extends JFrame implements ActionListener {
   JLabel creditCard, custID;
   JLabel apples, peaches, pears, total, cost, clicked;
   JButton view, reset;
   JPanel panel;
   JTextArea creditNo, customerNo;
   JTextArea applesNo, peachesNo, pearsNo, itotal, icost;
   static Send send;
   String customer;

   RMIClient2(){ //Begin Constructor
      //Create labels
      creditCard = new JLabel(“Credit Card:”);
      custID = new JLabel(“Customer ID:”);
      apples = new JLabel(“Apples:”);
      peaches = new JLabel(“Peaches:”);
      pears = new JLabel(“Pears:”);
      total = new JLabel(“Total Items:”);
      cost = new JLabel(“Total Cost:”);
```

```
    //Create text area components
    creditNo = new JTextArea();
    customerNo = new JTextArea();
    applesNo = new JTextArea();
    peachesNo = new JTextArea();
    pearsNo = new JTextArea();
    itotal = new JTextArea();
    icost = new JTextArea();

    //Create buttons
    view = new JButton("View Order");
    view.addActionListener(this);
    reset = new JButton("Reset");
    reset.addActionListener(this);

    //Create panel for 2-column layout
    //Set white background color
    panel = new JPanel();
    panel.setLayout(new GridLayout(0,2));
    panel.setBackground(Color.white);

    //Add components to panel columns
    //going left to right and top to bottom
    getContentPane().add(panel);
    panel.add(creditCard);
    panel.add(creditNo);
    panel.add(custID);
    panel.add(customerNo);
    panel.add(apples);
    panel.add(applesNo);
    panel.add(peaches);
    panel.add(peachesNo);
    panel.add(pears);
    panel.add(pearsNo);
    panel.add(total);
    panel.add(itotal);
    panel.add(cost);
    panel.add(icost);
    panel.add(view);
    panel.add(reset);
} //End Constructor
```

```
public void actionPerformed(ActionEvent event) {
   Object source = event.getSource();
   String text=null, unit, i;
   double cost;
   Double price;
   int items;
   Integer itms;
   DataOrder order = new DataOrder();

   //If View button pressed
   //Get data from server and display it
   //Extra Credit: If a DataOrder field is empty, the program
   //produces a stack trace when it tries to setText on a user interface
   //component with empty data. Add code to find empty fields
   //and assign alternate data in the event null fields are found
   if (source == view) {
      try {
         order = send.getOrder();
         creditNo.setText(order.cardnum);
         customerNo.setText(order.custID);
         applesNo.setText(order.apples);
         peachesNo.setText(order.peaches);
         pearsNo.setText(order.pears);
         cost = order.icost;
         price = new Double(cost);
         unit = price.toString();
         icost.setText(unit);
         items = order.itotal;
         itms = new Integer(items);
         i = itms.toString();
         itotal.setText(i);
      } catch (java.rmi.RemoteException e) {
         System.out.println("Cannot get data from server");
      }
   }

//If Reset button pressed
//Clear all fields
   if( source == reset) {
      creditNo.setText(“”);
      customerNo.setText(“”);
      applesNo.setText(“”);
```

```
            peachesNo.setText(“”);
            pearsNo.setText(“”);
            itotal.setText(“”);
            icost.setText(“”);
         }
      }

      public static void main(String[] args) {
         RMIClient2 frame = new RMIClient2();
         frame.setTitle(“Fruit Order”);
         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”);
         }
      }
}
```

RMIClient1 Improved Program

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

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

class RMIClient1Improved extends JFrame  implements ActionListener {
   JLabel col1, col2;
   JLabel totalItems, totalCost;
   JLabel cardNum, custID;
   JLabel applechk, pearchk, peachchk;
   JButton purchase, reset;
   JPanel panel;
   JTextField appleqnt, pearqnt, peachqnt;
   JTextField creditCard, customer;
   JTextArea items, cost;
   static Send send;

   RMIClient1Improved(){ //Begin Constructor
      //Create left and right column labels
      col1 = new JLabel("Select Items");
      col2 = new JLabel("Specify Quantity");

      //Create labels and text field components
      applechk = new JLabel("   Apples");
      appleqnt = new JTextField();
      appleqnt.addActionListener(this);
      pearchk = new JLabel("   Pears");
      pearqnt = new JTextField();
      pearqnt.addActionListener(this);
      peachchk = new JLabel("   Peaches");
      peachqnt = new JTextField();
      peachqnt.addActionListener(this);
      cardNum = new JLabel("   Credit Card:");
      creditCard = new JTextField();
      pearqnt.setNextFocusableComponent(creditCard);
      customer = new JTextField();
      custID = new JLabel("   Customer ID:");

      //Create labels and text area components
      totalItems = new JLabel("Total Items:");
      totalCost = new JLabel("Total Cost:");
      items = new JTextArea();
      cost = new JTextArea();
```

```
    //Create buttons and make action listeners
    purchase = new JButton("Purchase");
    purchase.addActionListener(this);
    reset = new JButton("Reset");
    reset.addActionListener(this);

    //Create a panel for the components
    panel = new JPanel();

    //Set panel layout to 2-column grid
    //on a white background
    panel.setLayout(new GridLayout(0,2));
    panel.setBackground(Color.white);

    //Add components to panel columns
    //going left to right and top to bottom
    getContentPane().add(panel);
    panel.add(col1);
    panel.add(col2);
    panel.add(applechk);
    panel.add(appleqnt);
    panel.add(peachchk);
    panel.add(peachqnt);
    panel.add(pearchk);
    panel.add(pearqnt);
    panel.add(totalItems);
    panel.add(items);
    panel.add(totalCost);
    panel.add(cost);
    panel.add(cardNum);
    panel.add(creditCard);
    panel.add(custID);
    panel.add(customer);
    panel.add(reset);
    panel.add(purchase);
} //End Constructor

public void actionPerformed(ActionEvent event){
    Object source = event.getSource();
    Integer applesNo, peachesNo, pearsNo, num;
    Double cost;
    String text, text2;
```

```
DataOrder order = new DataOrder();

//If Purchase button pressed  . . .
if (source == purchase) {
   //Get data from text fields
   order.cardnum = creditCard.getText();
   order.custID = customer.getText();
   order.apples = appleqnt.getText();
   order.peaches = peachqnt.getText();
   order.pears = pearqnt.getText();
   //Calculate total items
   if (order.apples.length() > 0) {

      //Catch invalid number error
      try {
         applesNo = Integer.valueOf(order.apples);
         order.itotal += applesNo.intValue();
      } catch(java.lang.NumberFormatException e) {
         appleqnt.setText("Invalid Value");
      }
   } else {
      /* else no need to change the total */
   }

   if (order.peaches.length() > 0){
      //Catch invalid number error
      try{
         peachesNo = Integer.valueOf(order.peaches);
         order.itotal += peachesNo.intValue();
      } catch(java.lang.NumberFormatException e) {
         peachqnt.setText("Invalid Value");
      }
   } else {
      /* else no need to change the total */
   }

   if (order.pears.length() > 0) {
      //Catch invalid number error
      try {
         pearsNo = Integer.valueOf(order.pears);
         order.itotal += pearsNo.intValue();
      } catch(java.lang.NumberFormatException e) {
```

```
            pearqnt.setText("Invalid Value");
         }
      } else {
         /* else no need to change the total */
      }

      //Display running total
      num = new Integer(order.itotal);
      text = num.toString();
      this.items.setText(text);

      //Calculate and display running cost
      order.icost = (order.itotal * 1.25);
      cost = new Double(order.icost);
      text2 = cost.toString();
      this.cost.setText(text2);
      try {
         send.sendOrder(order);
      } catch (java.rmi.RemoteException e) {
         System.out.println("Cannot send data to server");
      }
   }

   //If Reset button pressed
   //Clear all fields
   if (source == reset) {
      creditCard.setText("");
      appleqnt.setText("");
      peachqnt.setText("");
      pearqnt.setText("");
      creditCard.setText("");
      customer.setText("");
      order.icost = 0;
      cost = new Double(order.icost);
      text2 = cost.toString();
      this.cost.setText(text2);
      order.itotal = 0;
      num = new Integer(order.itotal);
      text = num.toString();
      this.items.setText(text);
   }
}
```

```
   public static void main(String[] args) {
      RMIClient1Improved frame = new RMIClient1Improved();
      frame.setTitle("Fruit $1.25 Each");
      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");
      }
   }
}
```

12 Develop the Example

The program in its current form lets sending clients overwrite each other's data before receiving clients have a chance to get and process it. This lesson adapts the server code so all orders are processed (nothing is overwritten), and are processed in the order received by the server.

This lesson shows you how to use object serialization in the server program to save the orders to files where they can be retrieved in the order they were saved. This lesson also shows you how to use the Collections API to maintain and display a list of unique customer IDs.

This lesson covers the following topics:

Track Orders

The changes to the example program so it uses serialization to keep track of the orders received and processed are primarily in the `RemoteServer` class. See *Code for This Lesson* for the full code listing.

sendOrder Method

The `sendOrder` method in the `RemoteServer` class accepts a `DataOrder` instance as input, and stores each order in a separate file where the file name is a number. The first order received is stored in a file named *1*, the second order is stored in a file named *2*, and so on. To keep track of the file names, the `value` variable is incremented by 1 each time the `sendOrder` method is called, converted to a `String`, and used for the file name in the serialization process.

Objects are serialized by creating a serialized output stream and writing the object to the output stream. In the code, the first line in the `try` block creates a `FileOutputStream` with the file name to which the serialized object is to be written. The next line creates an `ObjectOutputFileStream` from the file output stream. This is the serialized output stream to which the `order` object is written in the last line of the `try` block.

```
public synchronized void sendOrder(DataOrder order) throws java.io.IOException {
   value += 1;
   String orders = String.valueOf(value);

   try {
      FileOutputStream fos = new FileOutputStream(orders);
      oos = new ObjectOutputStream(fos);
      oos.writeObject(order);
   } catch (java.io.FileNotFoundException e) {
      System.out.println(e.toString());
   } finally {
      if (oos != null) {
      oos.close();
   }
   }
}
```

getOrder Method

The `getOrder` method in the `RemoteServer` class does what the `sendOrder` method does in reverse using the `get` variable to keep track of which orders have been viewed. But first, this method checks the `value` variable. If it is equal to zero, there are no orders to get from a file and view, and if it is greater than the value in the `get` variable, there is at least one order to get from a file and view. As each order is viewed, the `get` variable is incremented by 1.

```
public synchronized DataOrder getOrder() throws java.io.IOException {
   DataOrder order = null;
   ObjectInputStream ois = null;

   if (value == 0) {
      System.out.println(“No Orders To Process”);
   }

   if (value > get) {
      get += 1;
      String orders = String.valueOf(get);
      try {
         FileInputStream fis = new FileInputStream(orders);
         ois = new ObjectInputStream(fis);
         order = (DataOrder)ois.readObject();
      } catch (java.io.FileNotFoundException e) {
         System.out.println(e.toString());
         } catch (java.io.IOException e) {
            System.out.println(e.toString());
         } catch (java.lang.ClassNotFoundException e) {
            System.out.println("No data available");
         } finally {
            if (oos != null) {
            oos.close();
         }
      }
   } else {
      System.out.println(“No Orders To Process”);
   }
   return order;
}
```

Other Changes to Server Code

The `sendOrder` and `getOrder` methods are synchronized, declared to throw `java.io.IOException`, and have a `finally` clause in their `try` and `catch` blocks.

The `synchronized` keyword lets the `RemoteServer` instance handle the get and send requests one at a time. For example, there could be many instances of `RMIClient1` sending data to the server. The `sendOrder` method needs to write the data to file one request at a time so the data sent by one `RMIClient1` client does not overwrite or collide with the data sent by another `RMIClient1`. The `getOrder` method is also synchronized so the server can retrieve data for one `RMIClient2` instance before retrieving data for another `RMIClient2` instance.

The `finally` clause is added to the `try` and `catch` blocks of the `sendOrder` and `getOrder` methods to close the object output and input streams and free up any system resources associated with them. The `close` method for the `ObjectInputStream` and `ObjectOutputStream` classes throws `java.io.IOException` checked exception, which as described in *Chapter 6, Exception Handling*, has to be either caught or declared in the `throws` clause of the method signatures. Because a `finally` clause cannot specify an exception, the best thing to do in this example is declare `java.io.IOException` in the method signatures for the `sendOrder` and `getOrder` methods.

Adding the `java.io.IOException` checked exception to the method signatures means the Send interface has to be changed so the `sendOrder` and `getOrder` methods throw `java.io.IOException` in addition to `RemoteException`:

```
public interface Send extends Remote {
   public void sendOrder(DataOrder order)
                         throws RemoteException, java.io.IOException;
   public DataOrder getOrder()
   throws RemoteException, java.io.IOException;
}
```

This change to the Send interface means you have to add the catch clause shown below to the `RMIClient1` and `RMIClient2` programs to handle the `java.io.IOException` thrown by the `sendOrder` and `getOrder` methods.

RMIClient1

```
try{
   send.sendOrder(order);
} catch (java.rmi.RemoteException e) {
   System.out.println("Cannot send data to server");
//Need to catch this exception
} catch (java.io.IOException e) {
   System.out.println("Unable to write to file");
}
```

RMIClient2

```
if (source == view) {
   try {
      order  = send.getOrder();
      creditNo.setText(order.cardnum);
      customerNo.setText(order.custID);
      applesNo.setText(order.apples);
      peachesNo.setText(order.peaches);
      pearsNo.setText(order.pears);
      cost = order.icost;
      price = new Double(cost);
      unit = price.toString();
      icost.setText(unit);
      items = order.itotal;
      itms = new Integer(items);
      i = itms.toString();
      itotal.setText(i);
   } catch (java.rmi.RemoteException e) {
      System.out.println("Cannot access data in server");
   //Need to catch this exception
   } catch (java.io.IOException e) {
      System.out.println("Unable to write to file");
   }
}
```

Maintain and Display a Customer List

This section adds methods to the RMIClient2 program to manage a list of unique customer IDs. The `addCustomer` method uses the Collections API to create the list and the `getData` method uses the Collections API to retrieve the list. The `getData` method also uses Project Swing APIs to display the customer list in a dialog box.

About Collections

A collection is an object that contains other objects, and provides methods for working on the objects it contains. A collection can consist of the same types of objects, but can contain objects of different types too. The customer IDs are all objects of type `String` and represent the same type of information, a customer ID. You could also have a collection object that contains objects of type `String`, `Integer`, and `Double` if it makes sense in your program.

The Collection classes available to use in programs implement Collection interfaces. The Collection interface implementations for each Collection class let collection objects be manipulated independently of their representation details.

There are three primary types of collection interfaces: `List, Set`, and `Map`. This section focuses on the `List` and `Set` collections.

`Set` implementations do not permit duplicate elements, but `List` implementations do. Duplicate elements have the same data type and value. For example, two customer IDs of type `String` containing the value Zelda are duplicate; whereas, an element of type `String` containing the value *1* and an element of type `Integer` containing the value *1* are not duplicate.

The API provides two general-purpose `Set` implementations. `HashSet`, which stores its elements in a hash table, and `TreeSet`, which stores its elements in a balanced binary tree called a red-black tree. The example for this lesson uses the `HashSet` implementation because it currently has the best performance. *Figure 30* shows the Collection interfaces on the right and the class hierarchy for the `java.util.HashSet` on the left. You can see that the `HashSet` class implements the `Set` interface.

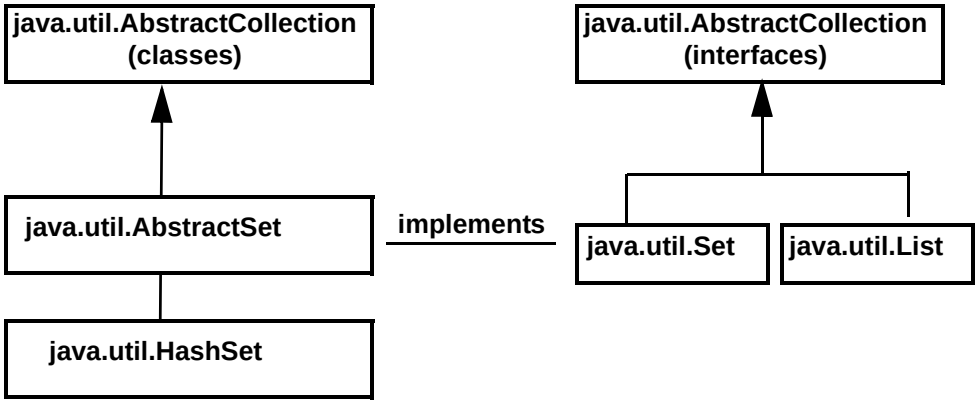


Figure 30. Collections API Interfaces and Class Hierarchy

Create a Set

This example adapts the `RMIClient2` class from *Chapter 11, User Interfaces Revisited* to collect customer IDs in a `Set` and send the list of customer IDs to the command line whenever the `View` button is clicked.

The collection object is a `Set` so if the same customer enters multiple orders, there is only one element for that customer in the list of customer IDs. If the program tries to add an element that is the same as an element already in the set, the second element is simply not added. No error is thrown and there is nothing you have to do in your code.

The `actionPerformed` method in the `RMIClient2` class calls the `addCustomer` method to add a customer ID to the set when the order processor clicks the `View` button. The `addCustomer` method implementation below adds the customer ID to the set and prints a notice that the customer ID has been added.

```
...
Set s = new HashSet();
...
```

```
public void addCustomer(String custID) {
   s.add(custID);
   System.out.println("Customer ID added");
}
```

Access Data in a Set

The `getData` method is called from the `actionPerformed` method in the `RMIClient2` class when the order processor clicks the `View` button. The `getData` method sends the elements currently in the set to the command line. The next section shows you how to display the output in a dialog box.

To traverse the set, an object of type `Iterator` is returned from the set. The `Iterator` object has a `hasNext` method that lets you test if there is another element in the set, a `next` method that lets you move over the elements in the set, and a `remove` method that lets you remove an element.

The example `getData` method shows two ways to access data in the set. The first way uses an iterator and the second way simply calls `System.out.println` on the set. In the iterator approach, the element returned by the `next` method is sent to the command line until there are no more elements in the set.

> **Note:** A HashSet does not guarantee the order of the elements in the set. Elements are sent to the command line in the order they occur in the set, but that order is not necessarily the same as the order in which the elements were placed in the set.

```
public void getData() {
   //Iterator approach
   if (s.size()!=0) {
      Iterator it = s.iterator();
      while (it.hasNext()) {
         System.out.println(it.next());
      }
      //Call System.out.println on the set
      System.out.println(s);
   } else {
      System.out.println("No customer IDs available");
   }
}
```

Here is the command-line output assuming three customer IDs (noel, munchkin, and samantha) were added to the set. Note how the `System.out.println(s)` invocation displays the three customer IDs between square brackets ([]) separated by commas.

```
Customer ID added
noel
munchkin
samantha
[noel, munchkin, samantha]
```

Display Data in a Dialog Box

The `getData` method is modified to display the set data in the dialog box shown in *Figure 31*.

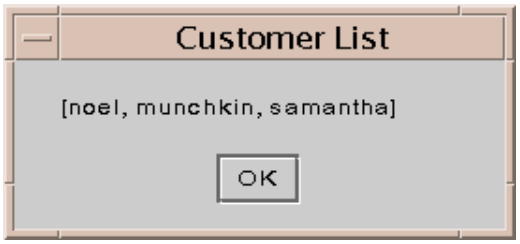


Figure 31. Display Customer Data in Dialog Box

```
public void getData() {
   if (s.size()!=0) {
      Iterator it = s.iterator();
      while (it.hasNext()) {
         System.out.println(it.next());
      }
      System.out.println(s);
      JOptionPane.showMessageDialog(frame, s.toString(), "Customer List",
                                    JOptionPane.PLAIN_MESSAGE);
   } else {
      System.out.println("No customer IDs available");
   }
}
```

Here is a description of the `JOptionPane` line that displays the dialog box:

Parameter	Description
JOptionPane.showMessageDialog(frame)	Show a simple message dialog box attached to this application's `frame`. This is the `frame` that is instantiated in the main method as `frame = new RMIClient2()`.
`s.toString()`	Display the contents of the set in the dialog box
"Customer List"	Use this text for the title.
JOptionPane.PLAIN_MESSAGE	This is a plain message dialog box so no icon such as a question or warning symbol is included with the message

Exercises

Modify the `try` and `catch` blocks in the `RMIClient1` and `RMIClient2` classes so they display error text in a dialog box instead of sending it to the command line. See *RMIClient2* on page 169 for the solution.

To test the program, run the `RMIClient2` program without starting the server. You should see the error dialog box shown in *Figure 32*.

Figure 32. Error Dialog

Code for This Lesson

- RemoteServer
- RMIClient2

RemoteServer Program

```
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;

class RemoteServer extends UnicastRemoteObject implements Send {
   Integer num = null;
   int value = 0, get = 0;
   ObjectOutputStream oos = null;

   public RemoteServer() throws RemoteException {
      super();
   }
   public synchronized void sendOrder(DataOrder order)
                                    throws java.io.IOException {
      value += 1;
      String orders = String.valueOf(value);
      try {
         FileOutputStream fos = new FileOutputStream(orders);
         oos = new ObjectOutputStream(fos);
         oos.writeObject(order);
      } catch (java.io.FileNotFoundException e) {
         System.out.println("File not found");
      } finally {
         if (oos != null) {
            oos.close();
         }
      }
   }
   public synchronized DataOrder getOrder() throws java.io.IOException {
      DataOrder order = null;
      ObjectInputStream ois = null;
      if (value == 0) {
```

```
            System.out.println("No Orders To Process");
         }
         if (value > get) {
            get += 1;
            String orders = String.valueOf(get);
            try {
               FileInputStream fis = new FileInputStream(orders);
               ois = new ObjectInputStream(fis);
               order = (DataOrder)ois.readObject();
            } catch (java.io.FileNotFoundException e) {
               System.out.println("File not found");
            } catch (java.io.IOException e) {
               System.out.println("Unable to read file");
            } catch (java.lang.ClassNotFoundException e){
               System.out.println("No data available");
            } finally {
               if (oos != null) {
                  oos.close();
               }
            }
         } else {
            System.out.println("No Orders To Process");
         }
         return order;
      }

      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");
         }
      }
}
```

RMIClient2

```
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.io.FileInputStream.*;
import java.io.RandomAccessFile.*;
import java.io.File;
import java.util.*;

class RMIClient2 extends JFrame implements ActionListener {
   JLabel creditCard, custID, apples, peaches, pears, total, cost, clicked;
   JButton view, reset;
   JPanel panel;
   JTextArea creditNo, customerNo, applesNo, peachesNo, pearsNo, itotal, icost;
   static Send send;
   String customer;
   Set s = new HashSet();
   static RMIClient2 frame;

   RMIClient2(){ //Begin Constructor
      //Create labels
      creditCard = new JLabel("Credit Card:");
      custID = new JLabel("Customer ID:");
      apples = new JLabel("Apples:");
      peaches = new JLabel("Peaches:");
      pears = new JLabel("Pears:");
      total = new JLabel("Total Items:");
      cost = new JLabel("Total Cost:");
      //Create text areas
      creditNo = new JTextArea();
      customerNo = new JTextArea();
      applesNo = new JTextArea();
      peachesNo = new JTextArea();
      pearsNo = new JTextArea();
      itotal = new JTextArea();
      icost = new JTextArea();
```

```
    //Create buttons
    view = new JButton("View Order");
    view.addActionListener(this);
    reset = new JButton("Reset");
    reset.addActionListener(this);
    //Create panel for 2-column layout
    //Set white background color
    panel = new JPanel();
    panel.setLayout(new GridLayout(0,2));
    panel.setBackground(Color.white);
    //Add components to panel columns
    //going left to right and top to bottom
    getContentPane().add(panel);
    panel.add(creditCard);
    panel.add(creditNo);
    panel.add(custID);
    panel.add(customerNo);
    panel.add(apples);
    panel.add(applesNo);
    panel.add(peaches);
    panel.add(peachesNo);
    panel.add(pears);
    panel.add(pearsNo);
    panel.add(total);
    panel.add(itotal);
    panel.add(cost);
    panel.add(icost);
    panel.add(view);
    panel.add(reset);
} //End Constructor

public void addCustomer(String custID) {
    System.out.println(custID);
    s.add(custID);
    System.out.println("Customer ID added");
}

public void getData() {
    if (s.size()!=0) {
        Iterator it = s.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
```

```
        }
        System.out.println(s);
        JOptionPane.showMessageDialog(frame, s.toString(), “Customer List”,
                                      JOptionPane.PLAIN_MESSAGE);
    } else {
        System.out.println(“No customer IDs available”);
    }
}

public void actionPerformed(ActionEvent event) {
    Object source = event.getSource();
    String unit, i;
    double cost;
    Double price;
    int items;
    Integer itms;
    DataOrder order = new DataOrder();
    //If View button pressed
    //Get data from server and display it
    if (source == view) {
        try {
            order  = send.getOrder();
            creditNo.setText(order.cardnum);
            customerNo.setText(order.custID);
        //Call addCustomer method
            addCustomer(order.custID);
            applesNo.setText(order.apples);
            peachesNo.setText(order.peaches);
            pearsNo.setText(order.pears);
            cost = order.icost;
            price = new Double(cost);
            unit = price.toString();
            icost.setText(unit);
            items = order.itotal;
            itms = new Integer(items);
            i = itms.toString();
            itotal.setText(i);
        } catch (java.rmi.RemoteException e) {
            System.out.println(“Cannot access data in server”);
        } catch (java.io.IOException e) {
            System.out.println(“Unable to write to file”);
        }
```

```
            getData();
        }

        //If Reset button pressed
        //Clear all fields
        if (source == reset) {
            creditNo.setText("");
            customerNo.setText("");
            applesNo.setText("");
            peachesNo.setText("");
            pearsNo.setText("");
            itotal.setText("");
            icost.setText("");
        }
    }

    public static void main(String[] args) {
        frame = new RMIClient2();
        frame.setTitle("Fruit Order");
        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.RemoteException e) {
            System.out.println("Cannot create remote server object");
        } catch (java.net.MalformedURLException e) {
            System.out.println("Cannot look up server object");
        } catch (java.rmi.NotBoundException e) {
            System.out.println("Cannot access data in server");
        }
    }
}
```

13 Internationalization

More and more companies, large and small, are doing business around the world using many different languages. Effective communication is always good business, so it follows that adapting an application to a local language adds to profitability through better communication and increased satisfaction.

The Java 2 platform provides internationalization features that let you separate culturally dependent data from the application (internationalization) and adapt it to as many cultures as needed (localization).

This lesson takes the two client programs from *Chapter 12, Develop the Example*, internationalizes them and localizes the text for France, Germany, and the United States.

This lesson covers the following topics:

Identify Culturally Dependent Data

To internationalize an application, the first thing you need to do is identify the culturally dependent data in your application. Culturally-dependent data is any data that varies from one culture or country to another. Text is the most obvious and pervasive example of culturally dependent data, but other things like number formats, sounds, times, and dates should be considered too.

The `RMIClient1` and `RMIClient2` classes have culturally-dependent data visible to the user. This data is included in the bullet list below. *Figure 33* shows the Fruit Order client, which displays some of the culturally-dependent data mentioned in the bullet list.

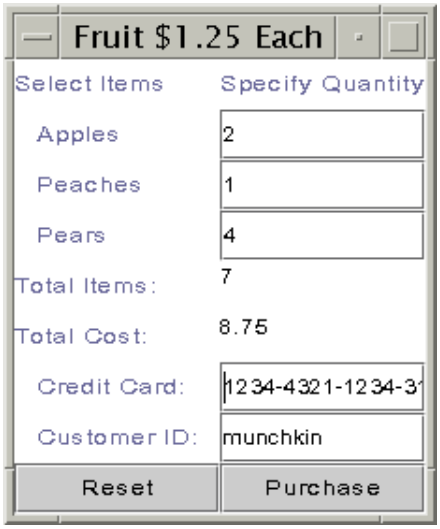


Figure 33. Culturally-Dependent Data

- Titles and labels (window titles, column heads, and left column labels)
- Buttons (Purchase, Reset, View)
- Numbers (values for item and cost totals)
- Error messages

Although the application has a server program, the server program is not being internationalized and localized. The only visible culturally-dependent data in the server program is the error message text. The server program runs in one place and the assumption is that it is not seen by anyone other than the system administrator who understands the language in which the error messages is hard coded. In this example, that language is United States English.

All error messages in `RMIClient1` and `RMIClient2` programs are handled in `try` and `catch` blocks. This way you have access to the error text for translation into another language.

```
//Access to error text
public void someMethod(){
   try {
```

```
      //do something
   } catch (java.util.NoSuchElementException {
      System.out.println(“Print some error text”);
   }
}
```

Methods can be coded to declare the exception in their `throws` clause, but this way you cannot access the error message text thrown when the method tries to access unavailable data in the set. In this case, the system-provided text for this error message is sent to the command line regardless of the locale in use for the application. The point here is it is always better to use `try` and `catch` blocks wherever possible if there is any chance the application will be internationalized so you can access and localize the error message text.

```
//No access to error text
public void someMethod() throws java.util.NoSuchElementException{
   //do something
}
```

Here is a list of the title, label, button, number, and error text visible to the user, and therefore, subject to internationalization and localization. This data was taken from both the `RMIClient1` and `RMIClient2` classes.

- Labels: Apples, Peaches, Pears, Total Items, Total Cost, Credit Card, Customer ID
- Titles: Fruit $1.25 Each, Select Items, Specify Quantity
- Buttons: Reset, View, Purchase
- Number Values: Value for total items, Value for total cost
- Errors: Invalid Value, Cannot send data to server, Cannot look up remote server object, No data available, No customer IDs available, Cannot access data in server

Create Keyword and Value Pair Files

Because all text visible to the user will be moved out of the application and translated, your application needs a way to access the translated text during execution. This is done with properties files that specify a list of keyword and value pairs for each language to be used. The application code loads the properties file for a given language and references the keywords instead of using hard-coded text.

So for example, you could map the keyword purchase to Kaufen in the German file, Achetez in the French file, and Purchase in the United States English file. In your application, you load the properties file for the language you want to use and reference the keyword purchase in your code. During execution when the purchase keyword is encountered, Achetez, Kaufen, or Purchase is loaded depending on the language file in use.

Keyword and value pairs are stored in properties files because they contain information about a program’s properties or characteristics. Property files are plain-text format, and you need one file for each language you intend to use.

In this example, there are three properties files, one each for the English, French, and German translations. Because this application currently uses hard-coded English text, the easiest way to begin the internationalization process is to use the hard-coded text to set up the key and value pairs for the English properties file.

The properties files follow a naming convention so the application can locate and load the correct file at run time. The naming convention uses language and country codes which you should make part of the file name. Both the language and country are included because the same language can vary between countries. For example, United States English and Australian English are a little different, and Swiss German and Austrian German both differ from each other and from the German spoken in Germany.

These are the names of the properties files for the German (`de_DE`), French (`fr_FR`), and American English (`en_US`) translations where `de`, `fr`, and `en` indicate the German (Deutsche), French, and English languages; and `DE`, `FR`, and `US` indicate Germany (Deutschland), France, and the United States:

- MessagesBundle_de_DE.properties
- MessagesBundle_en_US.properties
- MessagesBundle_fr_FR.properties

This is the English language properties file. Keywords are to the left of the equals (=) sign, and text values are on the right.

```
apples = Apples:
peaches = Peaches:
pears = Pears:
items = Total Items:
cost=Total Cost:
card=Credit Card:
customer=Customer ID:
title=Fruit 1.25 Each
1col=Select Items
2col=Specify Quantity
reset=Reset
view=View
purchase = Purchase
invalid = Invalid Value
send = Cannot send data to server
nolookup = Cannot look up remote server object
nodata = No data available
noID = No customer IDs available
noserver = Cannot access data in server
```

You can hand this file off to your French and German translators and ask them to provide the French and German equivalents for the text to the right of the equals (=) sign. Keep a copy because you will need the keywords to internationalize your application text.

The properties file with German translations produces the fruit order client user interface shown in *Figure 34*.

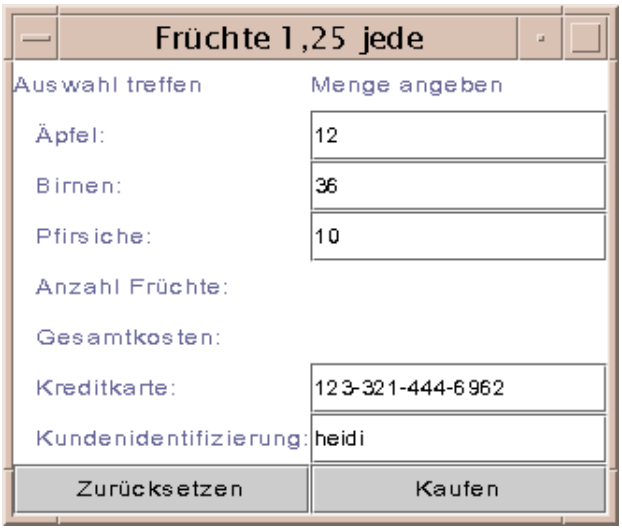


Figure 34. German User Interface

German Translations

```
apples=Äpfel:
peaches=Birnen:
pears=Pfirsiche:
items=Anzahl Früchte:
cost=Gesamtkosten:
card=Kreditkarte:
customer=Kundenidentifizierung:
title=Früchte 1,25 jede
1col=Auswahl treffen
2col=Menge angeben
reset=Zurücksetzen
view=Sehen Sie an
purchase=Kaufen
invalid=Ungültiger Wert
send=Datenübertragung zum Server nicht möglich
nolookup=Das Server läßt sich nicht zu finden
nodata=Keine Daten verfügbar
noID=Keine Kundenidentifizierungen verfügbar
noserver=Kein Zugang zu den Daten beim Server
```

The properties file with French translations produces the fruit order client user interface shown in *Figure 35*.

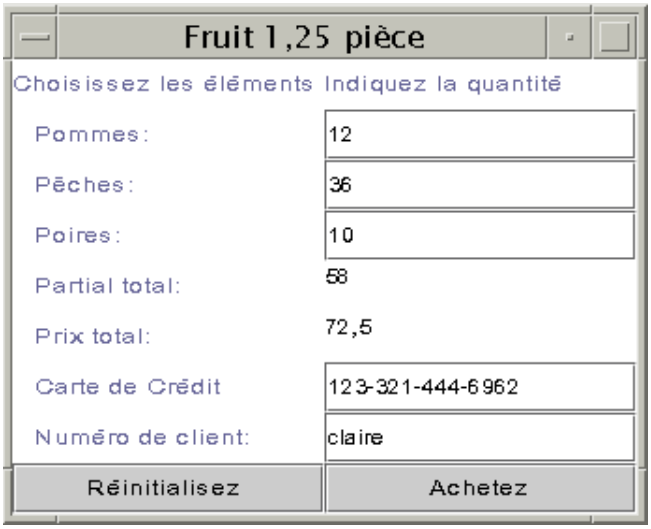


Figure 35. French User Interface

French Translations

```
apples=Pommes:
peaches=Pêches:
pears=Poires:
items=Partial total:
cost=Prix total:
card=Carte de Crédit
customer=Numêro de client:
title=Fruit 1,25 pièce
1col=Choisissez les éléments
2col= Indiquez la quantité
reset=Réinitialisez
view=Visualisez
purchase=Achetez
invalid=Valeur incorrecte
send=Les données n’ont pu être envoyées au serveur
nolookup=Accès impossible à l’objet du serveur distant
nodata=Aucune donnée disponible
noID=dentifiant du client indisponible
noserver=Accès aux données du serveur impossible
```

Internationalize Application Text

This section walks through internationalizing the `RMIClient1` code. The `RMIClient2` code is almost identical so you can apply the same steps to that program on your own.

Instance Variables

In addition to adding an import statement for the `java.util.*` package where the internationalization classes are, this program needs the following instance variable declarations for the internationalization process:

```
 //Initialized in main method
   static String language, country;
   Locale currentLocale;
   static ResourceBundle messages;
//Initialized in actionPerformed method
   NumberFormat numFormat;
```

main Method

The program is designed so the user specifies the language to use at the command line. So, the first change to the `main` method is to add the code to check the command line parameters. Specifying the language at the command line means once the application is internationalized, you can easily change the language without recompiling.

> **Note:** This style of programming makes it possible for the same user to run the program in different languages, but in most cases, the program will use one language and not rely on command-line arguments to set the country and language.

The `String[] args` parameter to the `main` method contains arguments passed to the program from the command line. This code expects 3 command line arguments when the user wants a language other than English. The first argument is the name of the machine on which the program is running. This value is passed to the program when it starts and is needed because this is a networked program using the RMI API.

The other two arguments specify the language and country codes. If the program is invoked with 1 command line argument (the machine name only), the country and language are assumed to be United States English.

As an example, here is how the program is started with command line arguments to specify the machine name and German language (de DE). Everything goes on one line.

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

The `main` method code appears below. The `currentLocale` instance variable is initialized from the `language` and `country` information passed in at the command line, and the `messages` instance variable is initialized from the `currentLocale`.

The `messages` object provides access to the translated text for the language in use. It takes two parameters: the first parameter `MessagesBundle` is the prefix of the family of translation files this application uses, and the second parameter is the `Locale` object that tells the `ResourceBundle` which translation to use. If the application is invoked with `de DE` command line parameters, this code creates a `ResourceBundle` variable to access the `MessagesBundle_de_DE.properties` file.

```
public static void main(String[] args){
//Check for language and country codes
   if (args.length != 3) {
      language = new String(“en”);
      country = new String (“US”);
      System.out.println(“English”);
   } else {
      language = new String(args[1]);
      country = new String(args[2]);
      System.out.println(language + country);
   }

   //Create locale and resource bundle
   currentLocale = new Locale(language, country);
   messages = ResourceBundle.getBundle(“MessagesBundle”, currentLocale);
   WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {System.exit(0);}
   };

   //Create the RMIClient1 object
   RMIClient1 frame = new RMIClient1();
   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(messages.getString(“nolookup”));
```

```
      } catch(java.rmi.RemoteException e) {
         System.out.println(messages.getString("nolookup"));
      } catch(java.net.MalformedURLException e) {
         System.out.println(messages.getString("nolookup"));
      }
   }
```

The applicable error text is accessed by calling the `getString` method on the `ResourceBundle`, and passing it the keyword that maps to the applicable error text.

```
   try {
      String name = "//" + args[0] + "/Send";
      send = ((Send) Naming.lookup(name));
   } catch (java.rmi.NotBoundException e) {
System.out.println(messages.getString("nolookup"));
   } catch(java.rmi.RemoteException e) {
      System.out.println(messages.getString("nolookup"));
   } catch(java.net.MalformedURLException e) {
      System.out.println(messages.getString("nolookup"));
   }
```

Constructor

The window title is set by calling the `getString` method on the `ResourceBundle`, and passing it the keyword that maps to the title text. You must pass the keyword exactly as it appears in the translation file, or you will get a runtime error indicating the resource is unavailable.

```
   RMIClient1(){
      //Set window title
      setTitle(messages.getString("title"));
```

The next thing the constructor does is use the `args` parameter to look up the remote server object. If there are any errors in this process, the `catch` statements get the applicable error text from the `ResourceBundle` and print it to the command line. User interface objects that display text, such as `JLabel` and `JButton`, are created the same way:

```
   //Create left and right column labels
   col1 = new JLabel(messages.getString("1col"));
   col2 = new JLabel(messages.getString("2col"));
   ...
   //Create buttons and make action listeners
   purchase = new JButton(messages.getString("purchase"));
   purchase.addActionListener(this);
   reset = new JButton(messages.getString("reset"));
   reset.addActionListener(this);
```

actionPerformed Method

In the `actionPerformed` method, the `Invalid Value` error is caught and translated. The `actionPerformed` method also calculates item and cost totals, translates them to the correct format for the language currently in use, and displays them in the user interface.

```
if (order.apples.length() > 0) {
   //Catch invalid number error
   try {
      applesNo = Integer.valueOf(order.apples);
      order.itotal += applesNo.intValue();
   } catch(java.lang.NumberFormatException e) {
   appleqnt.setText(messages.getString(“invalid”));
}
} else {
   /* else no need to change the total */
}
```

Internationalize Numbers

Use a `NumberFormat` object to translate numbers to the correct format for the language in use. A `NumberFormat` object is created from the `currentLocale`. The information in the `currentLocale` tells the `NumberFormat` object what number format to use.

Once you have a `NumberFormat` object, all you do is pass in the value you want translated, and you receive a `String` that contains the number in the correct format. The value can be passed in as any data type used for numbers such as `int`, `Integer`, `double`, or `Double`. No code to convert an `Integer` to an `int` and back again is needed.

```
//Create number formatter
numFormat = NumberFormat.getNumberInstance(currentLocale);
//Display running total
text = numFormat.format(order.itotal);
this.items.setText(text);
//Calculate and display running cost
order.icost = (order.itotal * 1.25);
text2 = numFormat.format(order.icost);
this.cost.setText(text2);
try {
   send.sendOrder(order);
} catch (java.rmi.RemoteException e) {
   System.out.println(messages.getString(“send”));
} catch (java.io.IOException e) {
   System.out.println("nodata");
}
```

Compile and Run the Application

Here are the summarized steps for compiling and running the example program. The complete code listings are on page 186. The important thing is when you start the client programs, include language and country codes if you want a language other than United States English.

Compile

UNIX

```
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
cp DataOrder.class /home/zelda/public_html/classes
```

Win32

```
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
copy DataOrder.class \home\zelda\public_html\classes
```

Start the RMI Registry

UNIX

```
unsetenv CLASSPATH
rmiregistry &
```

Win32

```
set CLASSPATH=
start rmiregistry
```

Start the Server

UNIX

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

Win32

```
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
```

Start the RMIClient1 Program in German

Note the addition of `de DE` for the German language and country at the end of the line.

UNIX

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

Win32

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

Start the RMIClient2 Program in French

Note the addition of `fr FR` for the French language and country at the end of the line.

UNIX

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

Win32

```
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 RMIClient2
kq6py.eng.sun.com/home/zelda/public_html fr FR
```

Exercises

A real-world scenario for an ordering application like this might be that `RMIClient` is an applet embedded in a web page. When orders are submitted, order processing staff run `RMIClient2` as applications from their local machines.

So, an interesting exercise is to convert the `RMIClient1` class to its applet equivalent. The translation files would be loaded by the applet from the same directory from which the browser loads the applet class.

One way is to have a separate applet for each language with the language and country codes hard coded. Your web page can let them choose a language by clicking a link that launches a web page with the appropriate applet. The source code files for the English, French, and German applets starts on page 220 in the *Appendix A, Code Listings*.

This is the HTML code to load the French applet on a web page.

```
<HTML>
<BODY>
<APPLET CODE=RMIFrenchApp.class WIDTH=300 HEIGHT=300> &lt;/APPLET>
</BODY> </HTML>
```

To run an applet written with Java 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 the `appletviewer` command 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 virtual machine (JVM) instead of the web browser's default JVM.

To use appletviewer, type the following where `rmiFrench.html` is the HTML file for the French applet.

```
appletviewer rmiFrench.html
```

Another improvement to the program as it currently stands would be enhancing the error message text. You can locate the errors in the Java API docs and use the information there to make the error message text user friendly by providing more specific information.

You might also want to adapt the client programs to catch and handle the error thrown when an incorrect keyword is used. This is the stack trace provided by the system when this type of error occurs:

```
Exception in thread "main" java.util.MissingResourceException: Can't find
resource   at java.util.ResourceBundle.getObject(Compiled Code) at
java.util.ResourceBundle.getString(Compiled Code)   at
RMIClient1.<init>(Compiled Code)   at RMIClient1.main(Compiled Code)
```

Code for This Lesson

- RMIClient1
- RMIClient2
- RMIFrenchApp

RMIClient1

```
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import java.text.*;

class RMIClient1 extends JFrame implements ActionListener {
   JLabel col1, col2;
   JLabel totalItems, totalCost;
   JLabel cardNum, custID;
   JLabel applechk, pearchk, peachchk;
   JButton purchase, reset;
   JPanel panel;
   JTextField appleqnt, pearqnt, peachqnt;
   JTextField creditCard, customer;
   JTextArea items, cost;
   static Send send;

//Internationalization variables
   static Locale currentLocale;
   static ResourceBundle messages;
   static String language, country;
   NumberFormat numFormat;

   RMIClient1() { //Begin Constructor
      setTitle(messages.getString("title"));
      //Create left and right column labels
      col1 = new JLabel(messages.getString("1col"));
```

```
col2 = new JLabel(messages.getString("2col"));

//Create labels and text field components
applechk = new JLabel("   " + messages.getString("apples"));
appleqnt = new JTextField();
appleqnt.addActionListener(this);
pearchk = new JLabel("   " + messages.getString("pears"));
pearqnt = new JTextField();
pearqnt.addActionListener(this);
peachchk = new JLabel("   " + messages.getString("peaches"));
peachqnt = new JTextField();
peachqnt.addActionListener(this);
cardNum = new JLabel("   " + messages.getString("card"));
creditCard = new JTextField();
pearqnt.setNextFocusableComponent(creditCard);
customer = new JTextField();
custID = new JLabel("   " + messages.getString("customer"));

//Create labels and text area components
totalItems = new JLabel("   " + messages.getString("items"));
totalCost = new JLabel("   " + messages.getString("cost"));
items = new JTextArea();
cost = new JTextArea();

//Create buttons and make action listeners
purchase = new JButton(messages.getString("purchase"));
purchase.addActionListener(this);
reset = new JButton(messages.getString("reset"));
reset.addActionListener(this);

//Create a panel for the components
panel = new JPanel();

//Set panel layout to 2-column grid
//on a white background
panel.setLayout(new GridLayout(0,2));
panel.setBackground(Color.white);

//Add components to panel columns
//going left to right and top to bottom
getContentPane().add(panel);
panel.add(col1);
```

```
    panel.add(col2);
    panel.add(applechk);
    panel.add(appleqnt);
    panel.add(peachchk);
    panel.add(peachqnt);
    panel.add(pearchk);
    panel.add(pearqnt);
    panel.add(totalItems);
    panel.add(items);
    panel.add(totalCost);
    panel.add(cost);
    panel.add(cardNum);
    panel.add(creditCard);
    panel.add(custID);
    panel.add(customer);
    panel.add(reset);
    panel.add(purchase);
} //End Constructor

public void actionPerformed(ActionEvent event) {
    Object source = event.getSource();
    Integer applesNo, peachesNo, pearsNo, num;
    Double cost;
    String text, text2;
    DataOrder order = new DataOrder();

    //If Purchase button pressed
    if (source == purchase) {
    //Get data from text fields
       order.cardnum = creditCard.getText();
       order.custID = customer.getText();
       order.apples = appleqnt.getText();
       order.peaches = peachqnt.getText();
       order.pears = pearqnt.getText();
    //Calculate total items
       if (order.apples.length() > 0) {
       //Catch invalid number error
          try {
             applesNo = Integer.valueOf(order.apples);
             order.itotal += applesNo.intValue();
          } catch (java.lang.NumberFormatException e) {
             appleqnt.setText(messages.getString(“invalid”));
```

```
   }
} else {
   /* else no need to change the total */
}

if (order.peaches.length() > 0) {
//Catch invalid number error
   try {
      peachesNo = Integer.valueOf(order.peaches);
      order.itotal += peachesNo.intValue();
   } catch(java.lang.NumberFormatException e) {
      peachqnt.setText(messages.getString(“invalid”));
   }
} else {
   /* else no need to change the total */
}
if (order.pears.length() > 0){
   //Catch invalid number error
   try {
      pearsNo = Integer.valueOf(order.pears);
      order.itotal += pearsNo.intValue();
   } catch (java.lang.NumberFormatException e) {
      pearqnt.setText(messages.getString(“invalid”));
   }
} else {
   /* else no need to change the total */
}

//Create number formatter
numFormat = NumberFormat.getNumberInstance(currentLocale);
//Display running total
text = numFormat.format(order.itotal);
this.items.setText(text);
//Calculate and display running cost
order.icost = (order.itotal * 1.25);
text2 = numFormat.format(order.icost);
this.cost.setText(text2);
try{
   send.sendOrder(order);
} catch (java.rmi.RemoteException e) {
   System.out.println(messages.getString(“send”));
} catch (java.io.IOException e) {
```

```
            System.out.println(“nodata");
        }
    }

    //If Reset button pressed
    //Clear all fields
    if (source == reset) {
        creditCard.setText(“”);
        appleqnt.setText(“”);
        peachqnt.setText(“”);
        pearqnt.setText(“”);
        creditCard.setText(“”);
        customer.setText(“”);
        order.icost = 0;
        cost = new Double(order.icost);
        text2 = cost.toString();
        this.cost.setText(text2);
        order.itotal = 0;
        num = new Integer(order.itotal);
        text = num.toString();
        this.items.setText(text);
    }
}

public static void main(String[] args) {
    if (args.length != 3) {
        language = new String(“en”);
        country = new String (“US”);
        System.out.println(“English”);
    } else {
        language = new String(args[1]);
        country = new String(args[2]);
        System.out.println(language + country);
    }
    currentLocale = new Locale(language, country);
    messages = ResourceBundle.getBundle(“MessagesBundle”, currentLocale);
    WindowListener l = new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    };
    RMIClient1 frame = new RMIClient1();
```

```
      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(messages.getString("nolookup"));
      } catch(java.rmi.RemoteException e){
         System.out.println(messages.getString("nolookup"));
      } catch(java.net.MalformedURLException e) {
         System.out.println(messages.getString("nollokup"));
      }
   }
}
```

RMIClient2

```
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.io.FileInputStream.*;
import java.io.RandomAccessFile.*;
import java.io.File;
import java.util.*;
import java.text.*;

class RMIClient2 extends JFrame implements ActionListener {
   JLabel creditCard, custID, apples, peaches, pears, total, cost, clicked;
   JButton view, reset;
   JPanel panel;
   JTextArea creditNo, customerNo, applesNo, peachesNo, pearsNo, itotal, icost;
   static Send send;
   String customer;
```

```
   Set s = null;
   RMIClient2 frame;
//Internationalization variables
   static Locale currentLocale;
   static ResourceBundle messages;
   static String language, country;
   NumberFormat numFormat;

   RMIClient2(){ //Begin Constructor
      setTitle(messages.getString("title"));
      //Create labels
      creditCard = new JLabel(messages.getString("card"));
      custID = new JLabel(messages.getString("customer"));
      apples = new JLabel(messages.getString("apples"));
      peaches = new JLabel(messages.getString("peaches"));
      pears = new JLabel(messages.getString("pears"));
      total = new JLabel(messages.getString("items"));
      cost = new JLabel(messages.getString("cost"));

      //Create text areas
      creditNo = new JTextArea();
      customerNo = new JTextArea();
      applesNo = new JTextArea();
      peachesNo = new JTextArea();
      pearsNo = new JTextArea();
      itotal = new JTextArea();
      icost = new JTextArea();

      //Create buttons
      view = new JButton(messages.getString("view"));
      view.addActionListener(this);
      reset = new JButton(messages.getString("reset"));
      reset.addActionListener(this);

      //Create panel for 2-column layout
      //Set white background color
      panel = new JPanel();
      panel.setLayout(new GridLayout(0,2));
      panel.setBackground(Color.white);

      //Add components to panel columns
      //going left to right and top to bottom
```

```
   getContentPane().add(panel);
   panel.add(creditCard);
   panel.add(creditNo);
   panel.add(custID);
   panel.add(customerNo);
   panel.add(apples);
   panel.add(applesNo);
   panel.add(peaches);
   panel.add(peachesNo);
   panel.add(pears);
   panel.add(pearsNo);
   panel.add(total);
   panel.add(itotal);
   panel.add(cost);
   panel.add(icost);
   panel.add(view);
   panel.add(reset);
} //End Constructor

//Create list of customer IDs
public void addCustomer(String custID){
   s.add(custID);
   System.out.println("Customer ID added");
}

//Get customer IDs
public void getData(){
   if (s.size()!=0) {
      Iterator it = s.iterator();
      while (it.hasNext()) {
         System.out.println(it.next());
      }
      System.out.println(s);
      JOptionPane.showMessageDialog(frame, s.toString(), "Customer List",
                                    JOptionPane.PLAIN_MESSAGE);
   } else {
      System.out.println("No customer IDs available");
   }
}

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

```
String unit, i;
double cost;
Double price;
int items;
Integer itms;
DataOrder order = new DataOrder();

//If View button pressed
//Get data from server and display it
if (source == view) {
   try {
      order  = send.getOrder();
      creditNo.setText(order.cardnum);
      customerNo.setText(order.custID);
      //Get customer ID and add to list
      addCustomer(order.custID);
      applesNo.setText(order.apples);
      peachesNo.setText(order.peaches);
      pearsNo.setText(order.pears);
      //Create number formatter
      numFormat = NumberFormat.getNumberInstance(currentLocale);
      price = new Double(order.icost);
      unit = numFormat.format(price);
      icost.setText(unit);
      itms = new Integer(order.itotal);
      i = numFormat.format(order.itotal);
      itotal.setText(i);
   } catch (java.rmi.RemoteException e) {
      System.out.println(“Cannot access data in server”);
   }catch (java.io.IOException e) {
      System.out.println("nodata");
   }
   //Get Customer Information
   getData();
}
//If Reset button pressed
//Clear all fields
if(source == reset){
   creditNo.setText(“”);
   customerNo.setText(“”);
   applesNo.setText(“”);
   peachesNo.setText(“”);
```

```
            pearsNo.setText(“”);
            itotal.setText(“”);
            icost.setText(“”);
         }
      }

      public static void main(String[] args) {
         if(args.length != 3) {
            language = new String(“en”);
            country = new String (“US”);
            System.out.println(“English”);
         } else {
            language = new String(args[1]);
            country = new String(args[2]);
            System.out.println(language + country);
         }
         currentLocale = new Locale(language, country);
         messages = ResourceBundle.getBundle(“MessagesBundle”, currentLocale);
         WindowListener l = new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
               System.exit(0);
            }
         };
         RMIClient2 frame = new RMIClient2();
         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(messages.getString(“nolookup”));
         } catch(java.rmi.RemoteException e) {
            System.out.println(messages.getString(“nolookup”));
         } catch(java.net.MalformedURLException e) {
            System.out.println(messages.getString(“nolookup”));
         }
      }
}
```

Packages and JAR File Format

14

A package is a convenient way to organize groups of related classes so they are easier to locate and use, and in development, you should organize application files into packages too. Packages also help you control `access` to class data at run time.

When your application is fully tested, debugged, and ready for deployment, use the Java Archive file format to bundle the application. JAR file format lets you bundle executable files with any other related application files so they can be deployed as one unit.

This lesson shows you how to organize the program files from the *Chapter 13, Internationalization* lesson into packages and deploy the executable and other related files to production using JAR file format.

This lesson covers the following topics:

- *Set up Class Packages*
- *Compile and Run the Example*
- *Exercises*

Set up Class Packages

It is easy to organize class files into packages. All you do is put related class files in the same directory, give the directory a name that relates to the purpose of the classes, and add a line to the top of each class file that declares the package name, which is the same as the directory name where they reside.

For example, the class and other related files for the program files from *Chapter 13* can be divided into three groups of files: fruit order client, view order client, and server files. Although these three sets of classes are related to each other, they have different functions and will be deployed separately.

Create the Directories

To organize the internationalization program into three packages, you could create the following three directories and move the listed source files into them:

- client1 package/directory
  - RMIEnglishApp.java
  - RMIFrenchApp.java
  - RMIGermanApp.java
  - MessagesBundle_de_DE.properties
  - MessagesBundle_en_US.properties
  - MessagesBundle_fr_FR.properties
  - index.html
  - rmiFapp.html
  - rmiGapp.html
  - rmiEapp.html
  - java.policy
- client2 package/directory
  - RMIClient2.java
  - MessagesBundle_de_DE.properties
  - MessagesBundle_en_US.properties
  - MessagesBundle_fr_FR.properties
  - java.policy
- server package/directory
  - DataOrder.java
  - RemoteServer.java
  - Send.java
  - java.policy

Declare the Packages

Each `*.java` file needs a package declaration at the top that reflects the name of the directory. Also, the fruit order (`client1 package`) and view order (`client2 package`) client class files need an import statement for the server package because they have to access the remote server object at runtime. For example, the package declaration and import statements for the `RMIClient2` class look like this:

```
//package declaration
package client2;
//import statements
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import java.text.*;
//import server package
import server.*;
```

Note: If you write an application that will be available for sale, it is important to choose package names that do not conflict with package names from other companies. A good way to avoid this problem is to prefix the package name with `com` and your company name. So, for example, the server package could be named `com.mycompanyname.server`.

Make Classes and Fields Accessible

With class files organized into packages, you have to declare the server classes in the `server` directory `public` so they can be instantiated by client programs, which are created from classes in the `client1` and `client2` directories. If you do not make the server classes `public`, they can only be instantiated by an object created from a class within the same package.

To make it possible for client programs to access the fruit order data, the fields of the `DataOrder` class have to be `public` too. The `RemoteServer` class on page 218 and `Send` 217 interface need to be `public` classes, but their fields do not need to be public because they do not have public data. Fields and methods without an access specifier such as

`public` are `package` by default and can only be accessed by objects created from classes in the same package. Here is the new DataOrder class:

```
package server;
import java.io.*;

//Make class public
public class DataOrder implements Serializable{

   //Make fields public
   public String apples, peaches, pears, cardnum, custID;
   public double icost;
   public int itotal;
}
```

Change Client Code to Find the Properties Files

In the example, the properties files (Messages_*) are stored in the directories with the client source files. This makes it easier to package and deploy the files later. So the programs can find the properties files, you have to make one small change to the client source code.

The code that creates the `messages` variable needs to include the directory (package) name `client2` as follows:

```
messages = ResourceBundle.getBundle("client2" + File.separatorChar +
            "MessagesBundle", currentLocale);
```

Compile and Run the Example

Compiling and running the example organized into packages is a little different from compiling and running the example in previous lessons. First, you have to execute the compiler and interpreter commands from one directory above the package directories, and second, you have to specify the package directories to the compiler and interpreter commands. You will find this code in *RMIClient1* .

Compile

UNIX

```
cd /home/zelda/classes
javac server/Send.java
javac server/RemoteServer.java
javac client2/RMIClient2.java
javac client1/RMIFrenchApp.java
javac client1/RMIGermanApp.java
```

```
javac client1/RMIEnglishApp.java
rmic -d . server.RemoteServer
cp server/RemoteServer*.class /home/zelda/public_html/classes/server
cp server/Send.class /home/zelda/public_html/classes/server
cp server/DataOrder.class /home/zelda/public_html/classes/server
```

Win32

```
cd \home\zelda\classes
javac server\Send.java
javac server\RemoteServer.java
javac client2\RMIClient2.java
javac client1\RMIFrenchApp.java
javac client1\RMIGermanApp.java
javac client1\RMIEnglishApp.java
rmic -d . server.RemoteServer
copy server\RemoteServer*.class \home\zelda\public_html\classes\server
copy server\Send.class \home\zelda\public_html\classes\server
copy server\DataOrder.class \home\zelda\public_html\classes\server
```

Note: The `rmic -d . server.RemoteServer` line uses `server.RemoteServer` instead of `server/RemoteServer` to correctly generate the `_stub` and `_skel` classes with the package.

Start the RMI Registry

UNIX

```
unsetenv CLASSPATH
rmiregistry &
```

Win32

```
set CLASSPATH=
start rmiregistry
```

Start the Server

UNIX

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

```
-Djava.security.policy= server/java.policy server/RemoteServer
```

Win32

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

Start the RMIGermanApp Program

Here is the HTML code to load the German applet, Note the directory/package name prefixed to the applet class name (`client1/RMIFrenchApp.class`).

```
<HTML>
<BODY>
<APPLET CODE=client1/RMIGermanApp.class WIDTH=300 HEIGHT=300>
</APPLET>
</BODY>
</HTML>
```

To run the applet with appletviewer, invoke the HTML file from the directory just above `client1` as follows:

```
appletviewer rmiGapp.html
```

Start the RMIClient2 Program in French

UNIX

```
java -Djava.rmi.server.codebase=http://kq6py/~zelda/classes
-Djava.rmi.server.hostname=kq6py.eng.sun.com
-Djava.security.policy=client2/java.policy  client2/RMIClient2
kq6py.eng.sun.com fr FR
```

Win32

```
java -Djava.rmi.server.codebase=       file:c:\home\zelda\public_html\classes
-Djava.rmi.server.hostname=kq6py.eng.sun.com   -Djava.security.policy=
client2\java.policy client2\RMIClient2 kq6py.eng.sun.com fr FR
```

Using JAR Files to Deploy

After testing and debugging, the best way to deploy the two client and server files is to bundle the executables and other related application files into three separate JAR files, where you have one JAR file for each client program, and one JAR file for the server program.

JAR files use the ZIP file format to compress and pack files into and decompress and unpack files from the JAR file. JAR files make it easy to deploy programs that consist of many files.

Browsers can easily download applets bundled into JAR files, and the download goes much more quickly than if the applet and its related files were not bundled into a JAR file.

Server Set of Files

These are the server files:

- RemoteServer.class
- RemoteServer_skel.class
- RemoteServer_stub.class
- Send.class
- DataOrder.class
- java.policy

Compress and Pack Server Files

To compress and pack the server files into one JAR file, type the following command on one line. This command is executed in the same directory with the files. If you execute the command from a directory other than where the files are, you have to specify the full pathname.

```
jar cf server.jar RemoteServer.class RemoteServer_skel.class
RemoteServer_stub.class  Send.class DataOrder.class java.policy
```

`jar` is the `jar` command. If you type `jar` with no options, you get the following help screen. You can see from the help screen that the `cf` options to the `jar` command mean create a new JAR file named `server.jar` and put the list of files that follows into it. The new JAR file is placed in the current directory.

```
kq6py% jar Usage: jar {ctxu}[vfm0M] [jar-file] [manifest-file] [-C dir] files
... Options:
-c create new archive
-t list table of contents for archive
-x extract named (or all) files from archive
-u update existing archive
-v generate verbose output on standard output
-f specify archive file name
-m include manifest information from specified manifest file
-0 store only; use no ZIP compression
-M Do not create a manifest file for the entries
-C change to the specified directory and include the following file
If any file is a directory then it is processed recursively. The manifest file
name and the archive file name needs to be specified in the same order the ‘m’
and ‘f’ flags are specified.
Example 1: to archive two class files into an archive called classes.jar:
jar cvf classes.jar Foo.class Bar.class
```

```
Example 2: use an existing manifest file ‘mymanifest’ and archive all the
files in the foo/ directory into ‘classes.jar’:      jar cvfm classes.jar
mymanifest -C foo/ .
To deploy the server files, all you have to do is move the
<CODE>server.jar</CODE> file to a publicly accessible directory on the server
where they are to execute.
```

Decompress and Unpack Server Files

After moving the JAR file to its final location, the compressed and packed files can be decompressed and unpacked so you can start the server. The following command means extract (`x`) all files from the `server.jar` file (`f`).

```
jar xf server.jar
```

> **Note:** It is also possible to start the server without decompressing and unpacking the JAR file first. You can find out how to do this by referring to the chapter on JAR files in *The Java Tutorial* referenced at the end of this lesson.

Fruit Order Set of Files (RMIClient1)

The fruit order set of files (below) consists of applet classes, web pages, translation files, and the policy file. Because they live on the web, they need to be in a directory accessible to the web server. The easiest way to deploy these files is to bundle them all into a JAR file and copy them to their location.

- RMIEnglishApp.class
- RMIFrenchApp.class
- RMIGermanApp.class
- index.html (top-level web page where user chooses language)
- rmiEapp.html (second-level web page for English)
- rmiFapp.html (second-level web page for French)
- rmiGapp.html (second-level web page for German)
- MessagesBundle_de_DE.properties
- MessagesBundle_en_US.properties
- MessagesBundle_fr_FR.properties
- java.policy

Compress and Pack Files

```
jar cf applet.jar RMIEnglishApp.class RMIFrenchApp.class RMIGermanApp.class
index.html rmiEapp.html rmiFapp.html rmiGapp.html
```

```
MessagesBundle_de_DE.properties  MessagesBundle_en_US.properties
MessagesBundle_fr_FR.properties java.policy
```

To deploy the fruit order client files, copy the `applet.jar` file to its final location.

Decompress and Unpack Files

An applet in a JAR file can be invoked from an HTML file without being unpacked. All you do is specify the `ARCHIVE` option to the `APPLET` tag in your web page, which tells the `appletviewer` tool the name of the JAR file containing the class file. Be sure to include the package directory when you specify the applet class to the `CODE` option.

When using appletviwer, you can leave the translation files and policy file in the JAR file. The applet invoked from the JAR file will find them in the JAR file.

```
<HTML>
<BODY>
<APPLET CODE=client1/R.class ARCHIVE=”applet.jar” WIDTH=300 HEIGHT=300>
</APPLET>
</BODY>
</HTML>
```

However, you do need to unpack the web pages so you can move them to their final location. The following command does this. Everything goes on one line.

```
jar xv applet.jar index.html rmiEapp.html rmiFapp.html rmiGapp.html
```

> **Note:** To run the HTML files from a browser, you need to unpack the JAR file, copy the `java.policy` file to your home directory and make sure it has the right name (`.java.policy` for UNIX and `java.policy` for Windows), and install Java Plug-In.

View Order Set of Files

The view order set of files (below) consists of the application class file and the policy file.

- RMIClient2.class
- java.policy

Compress and Pack Files

```
jar cf vieworder.jar RMIClient2.class java.policy
```

To deploy the view order client files, copy the `vieworder.jar` file to its final location.

Decompress and Unpack Files

```
jar xf vieworder.jar
```

Exercises

1. When you organize classes into a package, what is the package name the same as?
2. When do you have to make a class public?
3. How is compiling and running classes organized into packages different?
4. What are JAR files used for?
5. Can applet and server classes be executed from within a JAR file?

Code Listings

This appendix lists application code for the completed RMI application.

RMIClient1

```
//package statement
package client1;

import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import java.text.*;
import server.*;

class RMIClient1 extends JFrame implements ActionListener {
  private JLabel col1, col2;
  private JLabel totalItems, totalCost;
  private JLabel cardNum, custID;
  private JLabel applechk, pearchk, peachchk;
  private JButton purchase, reset;
  private JPanel panel;
  private JTextField appleqnt, pearqnt, peachqnt;
  private JTextField creditCard, customer;
  private JTextArea items, cost;
  private static Send send;

//Internationalization variables
  private static Locale currentLocale;
  private static ResourceBundle messages;
  private static String language, country;
  private NumberFormat numFormat;

  private RMIClient1(){ //Begin Constructor
    setTitle(messages.getString("title"));

//Create left and right column labels
    col1 = new JLabel(messages.getString("1col"));
    col2 = new JLabel(messages.getString("2col"));
```

```
//Create labels and text field components
   applechk = new JLabel(“   “ + messages.getString(“apples”));
    appleqnt = new JTextField();
    appleqnt.addActionListener(this);
    pearchk = new JLabel(“   “ + messages.getString(“pears”));
    pearqnt = new JTextField();
    pearqnt.addActionListener(this);
    peachchk = new JLabel(“   “ + messages.getString(“peaches”));
    peachqnt = new JTextField();
    peachqnt.addActionListener(this);
    cardNum = new JLabel(“   “ + messages.getString(“card”));
    creditCard = new JTextField();
    pearqnt.setNextFocusableComponent(creditCard);
    customer = new JTextField();
    custID = new JLabel(“   “ + messages.getString(“customer”));

//Create labels and text area components
    totalItems = new JLabel(“   “ + messages.getString(“items”));
    totalCost = new JLabel(“   “ + messages.getString(“cost”));
    items = new JTextArea();
    cost = new JTextArea();

//Create buttons and make action listeners
    purchase = new JButton(messages.getString(“purchase”));
    purchase.addActionListener(this);
    reset = new JButton(messages.getString(“reset”));
    reset.addActionListener(this);

//Create a panel for the components
    panel = new JPanel();

//Set panel layout to 2-column grid
//on a white background
    panel.setLayout(new GridLayout(0,2));
    panel.setBackground(Color.white);

//Add components to panel columns
//going left to right and top to bottom
    getContentPane().add(panel);
    panel.add(col1);
    panel.add(col2);
```

```
    panel.add(applechk);
    panel.add(appleqnt);
    panel.add(peachchk);
    panel.add(peachqnt);
    panel.add(pearchk);
    panel.add(pearqnt);
    panel.add(totalItems);
    panel.add(items);
    panel.add(totalCost);
    panel.add(cost);
    panel.add(cardNum);
    panel.add(creditCard);
    panel.add(custID);
     panel.add(customer);
     panel.add(reset);
     panel.add(purchase);
  } //End Constructor

  public void actionPerformed(ActionEvent event) {
   Object source = event.getSource();
   Integer applesNo, peachesNo, pearsNo, num;
   Double cost;
   String text, text2;
   DataOrder order = new DataOrder();

//If Purchase button pressed  . . .
   if (source == purchase) {
//Get data from text fields
    order.cardnum = creditCard.getText();
    order.custID = customer.getText();
    order.apples = appleqnt.getText();
    order.peaches = peachqnt.getText();
    order.pears = pearqnt.getText();

//Calculate total items
    if (order.apples.length() > 0) {
//Catch invalid number error
      try {
        applesNo = Integer.valueOf(order.apples);
        order.itotal += applesNo.intValue();
      } catch (java.lang.NumberFormatException e) {
        appleqnt.setText(messages.getString("invalid"));
```

```
      }
    } else {
      /* else no need to change the total */
    }
    if(order.peaches.length() > 0){

//Catch invalid number error
      try {
        peachesNo = Integer.valueOf(order.peaches);
        order.itotal += peachesNo.intValue();
      } catch(java.lang.NumberFormatException e) {
        peachqnt.setText(messages.getString("invalid"));
      }
    } else {
      /* else no need to change the total */
    }
    if(order.pears.length() > 0){

//Catch invalid number error
      try {
        pearsNo = Integer.valueOf(order.pears);
        order.itotal += pearsNo.intValue();
      } catch (java.lang.NumberFormatException e) {
        pearqnt.setText(messages.getString("invalid"));
      }
    } else {
      /* else no need to change the total */
    }

//Create number formatter
     numFormat = NumberFormat.getNumberInstance(currentLocale);
//Display running total
     text = numFormat.format(order.itotal);
     this.items.setText(text);
//Calculate and display running cost
     order.icost = (order.itotal * 1.25);
     text2 = numFormat.format(order.icost);
     this.cost.setText(text2);
     try {
        send.sendOrder(order);
      } catch (java.rmi.RemoteException e) {
        System.out.println(messages.getString("send"));
```

```
      } catch (java.io.IOException e) {
        System.out.println("Unable to write to file");
      }
   }
//If Reset button pressed
//Clear all fields
    if (source == reset) {
     creditCard.setText("");
     appleqnt.setText("");
     peachqnt.setText("");
     pearqnt.setText("");
     creditCard.setText("");
     customer.setText("");
     order.icost = 0;
     cost = new Double(order.icost);
     text2 = cost.toString();
     this.cost.setText(text2);
     order.itotal = 0;
     num = new Integer(order.itotal);
     text = num.toString();
     this.items.setText(text);
    }
  }

  public static void main(String[] args) {
    if (args.length != 3) {
      language = new String("en");
      country = new String ("US");
      System.out.println("English");
    } else {
      language = new String(args[1]);
      country = new String(args[2]);
      System.out.println(language + country);
    }
    currentLocale = new Locale(language, country);
                    messages = ResourceBundle.getBundle(
                    "client1" + File.separatorChar +
                    "MessagesBundle", currentLocale);
    WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
```

```
    };

  RMIClient1 frame = new RMIClient1();
    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(messages.getString(“nolookup”));
    } catch (java.rmi.RemoteException e) {
      System.out.println(messages.getString(“nolookup”));
    } catch (java.net.MalformedURLException e) {
      System.out.println(messages.getString(“nollokup”));
    }
  }
}
```

RMIClient2

```
//package statement
package client2;

import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.io.FileInputStream.*;
import java.io.RandomAccessFile.*;
import java.io.File;
import java.util.*;
import java.text.*;
```

```
import server.*;

class RMIClient2 extends JFrame implements ActionListener {
   private JLabel creditCard, custID, apples, peaches, pears, total, cost, clicked;
   private JButton view, reset;
   private JPanel panel;
   private JTextArea creditNo, customerNo, applesNo, peachesNo, pearsNo, itotal,
icost;
   private static Send send;
   private String customer;
   private Set s = new HashSet();
   private static RMIClient2 frame;

//Internationalization variables
   private static Locale currentLocale;
   private static ResourceBundle messages;
   private static String language, country;
   private NumberFormat numFormat;
   private RMIClient2(){ //Begin Constructor
   setTitle(messages.getString("title"));

//Create labels
     creditCard = new JLabel(messages.getString("card"));
     custID = new JLabel(messages.getString("customer"));
     apples = new JLabel(messages.getString("apples"));
     peaches = new JLabel(messages.getString("peaches"));
     pears = new JLabel(messages.getString("pears"));
     total = new JLabel(messages.getString("items"));
     cost = new JLabel(messages.getString("cost"));

//Create text areas
     creditNo = new JTextArea();
     customerNo = new JTextArea();
     applesNo = new JTextArea();
     peachesNo = new JTextArea();
     pearsNo = new JTextArea();
     itotal = new JTextArea();
     icost = new JTextArea();

//Create buttons
     view = new JButton(messages.getString("view"));
     view.addActionListener(this);
```

```
     reset = new JButton(messages.getString(“reset”));
     reset.addActionListener(this);

//Create panel for 2-column layout
//Set white background color
     panel = new JPanel();
     panel.setLayout(new GridLayout(0,2));
     panel.setBackground(Color.white);

//Add components to panel columns
//going left to right and top to bottom
     getContentPane().add(panel);
     panel.add(creditCard);
     panel.add(creditNo);
     panel.add(custID);
     panel.add(customerNo);
     panel.add(apples);
     panel.add(applesNo);
     panel.add(peaches);
     panel.add(peachesNo);
     panel.add(pears);
     panel.add(pearsNo);
     panel.add(total);
     panel.add(itotal);
     panel.add(cost);
     panel.add(icost);
     panel.add(view);
     panel.add(reset);
   } //End Constructor

 //Create list of customer IDs
  public void addCustomer(String custID) {
    s.add(custID);
    System.out.println("Customer ID added");
  }
//Get customer IDs
  public void getData() {
    if (s.size()!=0) {
      Iterator it = s.iterator();
      while (it.hasNext()) {
        System.out.println(it.next());
      }
```

```
      System.out.println(s);
      JOptionPane.showMessageDialog(frame, s.toString(), "Customer List",
JOptionPane.PLAIN_MESSAGE);
    } else {
      System.out.println("No customer IDs available");
    }
  }
  public void actionPerformed(ActionEvent event) {
     Object source = event.getSource();
     String unit, i;
     double cost;
     Double price;
     int items;
     Integer itms;
     DataOrder order = new DataOrder();

//If View button pressed
//Get data from server and display it
     if (source == view) {
        try {
          order  = send.getOrder();
          creditNo.setText(order.cardnum);
          customerNo.setText(order.custID);

//Add customerID to list
          addCustomer(order.custID);
          applesNo.setText(order.apples);
          peachesNo.setText(order.peaches);
          pearsNo.setText(order.pears);

//Create number formatter
          numFormat = NumberFormat.getNumberInstance(currentLocale);
          price = new Double(order.icost);
          unit = numFormat.format(price);
          icost.setText(unit);
          itms = new Integer(order.itotal);
          i = numFormat.format(order.itotal);
          itotal.setText(i);
        } catch (java.rmi.RemoteException e) {
          JOptionPane.showMessageDialog(frame, "Cannot get data from server",
"Error", JOptionPane.ERROR_MESSAGE);
        } catch (java.io.IOException e) {
```

```
          System.out.println("Unable to write to file");
        }
     //Display Customer IDs
      getData();
  }

//If Reset button pressed
//Clear all fields
     if (source == reset) {
       creditNo.setText("");
       customerNo.setText("");
       applesNo.setText("");
       peachesNo.setText("");
       pearsNo.setText("");
       itotal.setText("");
      icost.setText("");
     }
  }

  public static void main(String[] args) {
    if (args.length != 3) {
      language = new String("en");
      country = new String ("US");
      System.out.println("English");
    } else {
      language = new String(args[1]);
      country = new String(args[2]);
      System.out.println(language + country);
    }

    currentLocale = new Locale(language, country);
    messages = ResourceBundle.getBundle(
               "client2" + File.separatorChar +
               "MessagesBundle", currentLocale);
    WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };

    frame = new RMIClient2();
    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(messages.getString("nolookup"));
    } catch (java.rmi.RemoteException e) {
      System.out.println(messages.getString("nolookup"));
    } catch (java.net.MalformedURLException e) {
      System.out.println(messages.getString("nolookup"));
    }
  }
}
```

DataOrder

```
//package statement
package server;

import java.io.*;

public class DataOrder implements Serializable {
  public String apples, peaches, pears, custID, cardnum;
  public double icost;
  public int itotal;
}
```

Send

```
//package statement
package server;

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

public interface Send extends Remote {
```

```
  public void sendOrder(DataOrder order)
              throws RemoteException, java.io.IOException;
  public DataOrder getOrder()
              throws RemoteException, java.io.IOException;
}
```

RemoteServer

```
//package statement
package server;

import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;

class RemoteServer extends UnicastRemoteObject implements Send {
  Integer num = null;
  int value = 0, get = 0;
  ObjectOutputStream oos = null;

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

  public synchronized void sendOrder(DataOrder order) throws java.io.IOException{
   value += 1;
   String orders = String.valueOf(value);
   try {
      FileOutputStream fos = new FileOutputStream(orders);
      oos = new ObjectOutputStream(fos);
      oos.writeObject(order);
   } catch (java.io.FileNotFoundException e) {
      System.out.println("File not found");
   } finally {
     if (oos != null) {
       oos.close();
     }
   }
  }
```

```
public synchronized DataOrder getOrder() throws java.io.IOException{
  DataOrder order = null;
  ObjectInputStream ois = null;

  if (value == 0) {
    System.out.println("No Orders To Process");
  }
  if (value > get) {
    get += 1;
    String orders = String.valueOf(get);
    try {
      FileInputStream fis = new FileInputStream(orders);
      ois = new ObjectInputStream(fis);
      order = (DataOrder)ois.readObject();
    } catch (java.io.FileNotFoundException e) {
      System.out.println("File not found");
    } catch (java.io.IOException e) {
      System.out.println("Unable to read file");
    } catch (java.lang.ClassNotFoundException e) {
      System.out.println("No data available");
    } finally {
      if (oos != null) {
        oos.close();
      }
    }
  } else {
    System.out.println("No Orders To Process");
  }
 return order;
}

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");
    }
  }
}
```

RMIFrenchApp

```
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import java.text.*;
import java.applet.Applet;

//Make public
public class RMIFrenchApp extends Applet implements ActionListener {
   JLabel col1, col2;
   JLabel totalItems, totalCost;
   JLabel cardNum, custID;
   JLabel applechk, pearchk, peachchk;
   JButton purchase, reset;
   JTextField appleqnt, pearqnt, peachqnt;
   JTextField creditCard, customer;
   JTextArea items, cost;
   static Send send;

//Internationalization variables
   Locale currentLocale;
   ResourceBundle messages;
   static String language, country;
   NumberFormat numFormat;
   public void init(){
   language = new String(“fr”);
```

```
   country = new String (“FR”);
   if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }
    currentLocale = new Locale(language, country);
    messages = ResourceBundle.getBundle(“MessagesBundle”, currentLocale);
    Locale test = messages.getLocale();
    try {

//Path to host where remote Send object is running
       String name = “//kq6py.eng.sun.com/Send”;
       send = ((Send) Naming.lookup(name));
     } catch (java.rmi.NotBoundException e) {
       System.out.println(messages.getString(“nolookup”));
     } catch (java.rmi.RemoteException e) {
       System.out.println(messages.getString(“nolookup”));
     } catch (java.net.MalformedURLException e) {
       System.out.println(messages.getString(“nollokup”));
     }

//Create left and right column labels
     col1 = new JLabel(messages.getString(“1col”));
     col2 = new JLabel(messages.getString(“2col”));

//Create labels and text field components
     applechk = new JLabel(“   “ + messages.getString(“apples”));
     appleqnt = new JTextField();
     appleqnt.addActionListener(this);
     pearchk = new JLabel(“   “ + messages.getString(“pears”));
     pearqnt = new JTextField();
     pearqnt.addActionListener(this);
     peachchk = new JLabel(“   “ + messages.getString(“peaches”));
     peachqnt = new JTextField();
     peachqnt.addActionListener(this);
     cardNum = new JLabel(“   “ + messages.getString(“card”));
     creditCard = new JTextField();
     pearqnt.setNextFocusableComponent(creditCard);
     customer = new JTextField();
     custID = new JLabel(“   “ + messages.getString(“customer”));

//Create labels and text area components
     totalItems = new JLabel(“   “ + messages.getString(“items”));
```

```
     totalCost = new JLabel("   " + messages.getString("cost"));
     items = new JTextArea();
     cost = new JTextArea();

//Create buttons and make action listeners
     purchase = new JButton(messages.getString("purchase"));
     purchase.addActionListener(this);
     reset = new JButton(messages.getString("reset"));
     reset.addActionListener(this);

//Set panel layout to 2-column grid
//on a white background
     setLayout(new GridLayout(0,2));
     setBackground(Color.white);

//Add components to panel columns
//going left to right and top to bottom
     add(col1);
     add(col2);
     add(applechk);
     add(appleqnt);
     add(peachchk);
     add(peachqnt);
     add(pearchk);
     add(pearqnt);
     add(totalItems);
     add(items);
     add(totalCost);
     add(cost);
     add(cardNum);
     add(creditCard);
     add(custID);
     add(customer);
     add(reset);
     add(purchase);
   } //End Constructor

  public void actionPerformed(ActionEvent event) {
   Object source = event.getSource();
   Integer applesNo, peachesNo, pearsNo, num;
   Double cost;
   String text, text2;
```

```
   DataOrder order = new DataOrder();

//If Purchase button pressed  . . .
   if (source == purchase) {
//Get data from text fields
    order.cardnum = creditCard.getText();
    order.custID = customer.getText();
    order.apples = appleqnt.getText();
    order.peaches = peachqnt.getText();
    order.pears = pearqnt.getText();

//Calculate total items
    if (order.apples.length() > 0){
//Catch invalid number error
      try {
        applesNo = Integer.valueOf(order.apples);
        order.itotal += applesNo.intValue();
      } catch (java.lang.NumberFormatException e) {
        appleqnt.setText(messages.getString("invalid"));
      }
    } else {
      /* else no need to change the total */
    }

    if (order.peaches.length() > 0) {
//Catch invalid number error
      try
        peachesNo = Integer.valueOf(order.peaches);
        order.itotal += peachesNo.intValue();
      } catch (java.lang.NumberFormatException e) {
        peachqnt.setText(messages.getString("invalid"));
      }
    } else {
      /* else no need to change the total */
    }

    if (order.pears.length() > 0) {
//Catch invalid number error
      try{
        pearsNo = Integer.valueOf(order.pears);
        order.itotal += pearsNo.intValue();
      } catch(java.lang.NumberFormatException e) {
```

```
        pearqnt.setText(messages.getString(“invalid”));
      }
    } else {
      /* else no need to change the total */
    }

//Create number formatter
     numFormat = NumberFormat.getNumberInstance(currentLocale);
//Display running total
     text = numFormat.format(order.itotal);
     this.items.setText(text);
//Calculate and display running cost
     order.icost = (order.itotal * 1.25);
     text2 = numFormat.format(order.icost);
     this.cost.setText(text2);
     try {
        send.sendOrder(order);
      } catch (java.rmi.RemoteException e) {
          System.out.println(messages.getString(“send”));
      }catch (java.io.IOException e) {
       System.out.println("Unable to write to file");
      }
   }

//If Reset button pressed
//Clear all fields
    if (source == reset) {
     creditCard.setText(“”);
     appleqnt.setText(“”);
     peachqnt.setText(“”);
     pearqnt.setText(“”);
     creditCard.setText(“”);
     customer.setText(“”);
     order.icost = 0;
     cost = new Double(order.icost);
     text2 = cost.toString();
     this.cost.setText(text2);
     order.itotal = 0;
     num = new Integer(order.itotal);
     text = num.toString();
     this.items.setText(text);
    }
```

```
  }
}
```

RMIGermanApp

```
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import java.text.*;
import java.applet.Applet;

//Make public
public class RMIGermanApp extends Applet implements ActionListener {
   JLabel col1, col2;
   JLabel totalItems, totalCost;
   JLabel cardNum, custID;
   JLabel applechk, pearchk, peachchk;
   JButton purchase, reset;
   JTextField appleqnt, pearqnt, peachqnt;
   JTextField creditCard, customer;
   JTextArea items, cost;
   static Send send;

//Internationalization variables
   Locale currentLocale;
   ResourceBundle messages;
   static String language, country;
   NumberFormat numFormat;
   public void init(){
    language = new String(“de”);
    country = new String (“DE”);
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }
    currentLocale = new Locale(language, country);
```

```
    messages = ResourceBundle.getBundle(“MessagesBundle”, currentLocale);
    Locale test = messages.getLocale();
    try {

//Path to host where remote Send object is running
       String name = “//kq6py.eng.sun.com/Send”;
       send = ((Send) Naming.lookup(name));
     } catch (java.rmi.NotBoundException e) {
       System.out.println(messages.getString(“nolookup”));
     } catch java.rmi.RemoteException e) {
       System.out.println(messages.getString(“nolookup”));
     } catch (java.net.MalformedURLException e) {
       System.out.println(messages.getString(“nollokup”));
     }

//Create left and right column labels
     col1 = new JLabel(messages.getString(“1col”));
     col2 = new JLabel(messages.getString(“2col”));

//Create labels and text field components
     applechk = new JLabel(“   “ + messages.getString(“apples”));
     appleqnt = new JTextField();
     appleqnt.addActionListener(this);
     pearchk = new JLabel(“   “ + messages.getString(“pears”));
     pearqnt = new JTextField();
     pearqnt.addActionListener(this);
     peachchk = new JLabel(“   “ + messages.getString(“peaches”));
     peachqnt = new JTextField();
     peachqnt.addActionListener(this);
     cardNum = new JLabel(“   “ + messages.getString(“card”));
     creditCard = new JTextField();
     pearqnt.setNextFocusableComponent(creditCard);
     customer = new JTextField();
     custID = new JLabel(“   “ + messages.getString(“customer”));

//Create labels and text area components
     totalItems = new JLabel(“   “ + messages.getString(“items”));
     totalCost = new JLabel(“   “ + messages.getString(“cost”));
     items = new JTextArea();
     cost = new JTextArea();

//Create buttons and make action listeners
```

```
     purchase = new JButton(messages.getString("purchase"));
     purchase.addActionListener(this);
     reset = new JButton(messages.getString("reset"));
     reset.addActionListener(this);

//Set panel layout to 2-column grid
//on a white background
     setLayout(new GridLayout(0,2));
     setBackground(Color.white);

//Add components to panel columns
//going left to right and top to bottom
     add(col1);
     add(col2);
     add(applechk);
     add(appleqnt);
     add(peachchk);
     add(peachqnt);
     add(pearchk);
     add(pearqnt);
     add(totalItems);
     add(items);
     add(totalCost);
     add(cost);
     add(cardNum);
     add(creditCard);
     add(custID);
     add(customer);
     add(reset);
     add(purchase);
   } //End Constructor

  public void actionPerformed(ActionEvent event) {
   Object source = event.getSource();
   Integer applesNo, peachesNo, pearsNo, num;
   Double cost;
   String text, text2;
   DataOrder order = new DataOrder();

//If Purchase button pressed  . . .
   if (source == purchase) {
//Get data from text fields
```

```
    order.cardnum = creditCard.getText();
    order.custID = customer.getText();
    order.apples = appleqnt.getText();
    order.peaches = peachqnt.getText();
    order.pears = pearqnt.getText();

//Calculate total items
    if (order.apples.length() > 0) {
//Catch invalid number error
      try {
        applesNo = Integer.valueOf(order.apples);
        order.itotal += applesNo.intValue();
      } catch (java.lang.NumberFormatException e) {
        appleqnt.setText(messages.getString(“invalid”));
      }
    } else {
      /* else no need to change the total */
    }

    if (order.peaches.length() > 0) {
//Catch invalid number error
      try {
        peachesNo = Integer.valueOf(order.peaches);
        order.itotal += peachesNo.intValue();
      } catch (java.lang.NumberFormatException e){
        peachqnt.setText(messages.getString(“invalid”));
      }
    } else {
      /* else no need to change the total */
    }

    if (order.pears.length() > 0) {
//Catch invalid number error
      try {
        pearsNo = Integer.valueOf(order.pears);
        order.itotal += pearsNo.intValue();
      } catch (java.lang.NumberFormatException e) {
        pearqnt.setText(messages.getString(“invalid”));
      }
    } else {
      /* else no need to change the total */
    }
```

```
//Create number formatter
     numFormat = NumberFormat.getNumberInstance(currentLocale);
//Display running total
     text = numFormat.format(order.itotal);
     this.items.setText(text);
//Calculate and display running cost
     order.icost = (order.itotal * 1.25);
     text2 = numFormat.format(order.icost);
     this.cost.setText(text2);
     try{
        send.sendOrder(order);
      } catch (java.rmi.RemoteException e) {
          System.out.println(messages.getString(“send”));
      }catch (java.io.IOException e) {
       System.out.println("Unable to write to file");
      }
   }

//If Reset button pressed
//Clear all fields
    if source == reset) {
     creditCard.setText(“”);
     appleqnt.setText(“”);
     peachqnt.setText(“”);
     pearqnt.setText(“”);
     creditCard.setText(“”);
     customer.setText(“”);
     order.icost = 0;
     cost = new Double(order.icost);
     text2 = cost.toString();
     this.cost.setText(text2);
     order.itotal = 0;
     num = new Integer(order.itotal);
     text = num.toString();
     this.items.setText(text);
    }
  }
}
```

RMIEnglishApp

```
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import java.text.*;
import java.applet.Applet;

//Make public
public class RMIEnglishApp extends Applet implements ActionListener {
   JLabel col1, col2;
   JLabel totalItems, totalCost;
   JLabel cardNum, custID;
   JLabel applechk, pearchk, peachchk;
   JButton purchase, reset;
   JTextField appleqnt, pearqnt, peachqnt;
   JTextField creditCard, customer;
   JTextArea items, cost;
   static Send send;

//Internationalization variables
   Locale currentLocale;
   ResourceBundle messages;
   static String language, country;
   NumberFormat numFormat;

   public void init() {
    language = new String("en");
    country = new String ("US");
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }
    currentLocale = new Locale(language, country);
    messages = ResourceBundle.getBundle("MessagesBundle", currentLocale);
    Locale test = messages.getLocale();
```

```
    try {

//Path to host where remote Send object is running
       String name = "//kq6py.eng.sun.com/Send";
       send = ((Send) Naming.lookup(name));
     } catch (java.rmi.NotBoundException e) {
       System.out.println(messages.getString("nolookup"));
     } catch (java.rmi.RemoteException e) {
       System.out.println(messages.getString("nolookup"));
     } catch(java.net.MalformedURLException e) {
       System.out.println(messages.getString("nollookup"));
     }

//Create left and right column labels
     col1 = new JLabel(messages.getString("1col"));
     col2 = new JLabel(messages.getString("2col"));

//Create labels and text field components
     applechk = new JLabel("   " + messages.getString("apples"));
     appleqnt = new JTextField();
     appleqnt.addActionListener(this);
     pearchk = new JLabel("   " + messages.getString("pears"));
     pearqnt = new JTextField();
     pearqnt.addActionListener(this);
     peachchk = new JLabel("   " + messages.getString("peaches"));
     peachqnt = new JTextField();
     peachqnt.addActionListener(this);
     cardNum = new JLabel("   " + messages.getString("card"));
     creditCard = new JTextField();
     pearqnt.setNextFocusableComponent(creditCard);
     customer = new JTextField();
     custID = new JLabel("   " + messages.getString("customer"));

//Create labels and text area components
     totalItems = new JLabel("   " + messages.getString("items"));
     totalCost = new JLabel("   " + messages.getString("cost"));
     items = new JTextArea();
     cost = new JTextArea();

//Create buttons and make action listeners
     purchase = new JButton(messages.getString("purchase"));
     purchase.addActionListener(this);
```

```
     reset = new JButton(messages.getString("reset"));
     reset.addActionListener(this);

//Set panel layout to 2-column grid
//on a white background
     setLayout(new GridLayout(0,2));
     setBackground(Color.white);

//Add components to panel columns
//going left to right and top to bottom
     add(col1);
     add(col2);
     add(applechk);
     add(appleqnt);
     add(peachchk);
     add(peachqnt);
     add(pearchk);
     add(pearqnt);
     add(totalItems);
     add(items);
     add(totalCost);
     add(cost);
     add(cardNum);
     add(creditCard);
     add(custID);
     add(customer);
     add(reset);
     add(purchase);
   } //End Constructor

  public void actionPerformed(ActionEvent event) {
   Object source = event.getSource();
   Integer applesNo, peachesNo, pearsNo, num;
   Double cost;
   String text, text2;
   DataOrder order = new DataOrder();
//If Purchase button pressed  . . .
   if(source == purchase){
//Get data from text fields
    order.cardnum = creditCard.getText();
    order.custID = customer.getText();
    order.apples = appleqnt.getText();
```

```
    order.peaches = peachqnt.getText();
    order.pears = pearqnt.getText();
//Calculate total items
    if(order.apples.length() > 0){
//Catch invalid number error
      try {
        applesNo = Integer.valueOf(order.apples);
        order.itotal += applesNo.intValue();
      } catch (java.lang.NumberFormatException e) {
        appleqnt.setText(messages.getString(“invalid”));
      }
    } else {
      /* else no need to change the total */
    }
    if(order.peaches.length() > 0){
//Catch invalid number error
      try{
        peachesNo = Integer.valueOf(order.peaches);
        order.itotal += peachesNo.intValue();
      } catch(java.lang.NumberFormatException e) {
        peachqnt.setText(messages.getString(“invalid”));
      }
    } else {
      /* else no need to change the total */
    }
    if (order.pears.length() > 0) {
//Catch invalid number error
      try {
        pearsNo = Integer.valueOf(order.pears);
        order.itotal += pearsNo.intValue();
      } catch(java.lang.NumberFormatException e) {
        pearqnt.setText(messages.getString(“invalid”));
      }
    } else {
      /* else no need to change the total */
    }
//Create number formatter
     numFormat = NumberFormat.getNumberInstance(currentLocale);
//Display running total
     text = numFormat.format(order.itotal);
     this.items.setText(text);
//Calculate and display running cost
```

```
     order.icost = (order.itotal * 1.25);
     text2 = numFormat.format(order.icost);
     this.cost.setText(text2);
     try {
        send.sendOrder(order);
      } catch (java.rmi.RemoteException e) {
          System.out.println(“send”));
      }catch (java.io.IOException e) {
       System.out.println("Unable to write to file");
      }
   }

//If Reset button pressed
//Clear all fields
    if (source == reset) {
     creditCard.setText(“”);
     appleqnt.setText(“”);
     peachqnt.setText(“”);
     pearqnt.setText(“”);
     creditCard.setText(“”);
     customer.setText(“”);
     order.icost = 0;
     cost = new Double(order.icost);
     text2 = cost.toString();
     this.cost.setText(text2);
     order.itotal = 0;
     num = new Integer(order.itotal);
     text = num.toString();
     this.items.setText(text);
    }
  }
}
```

RMIClientView Program

```
package client1;

import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.WindowListener;
import java.awt.event.WindowAdapter;
```

```
import java.awt.event.WindowEvent;
import javax.swing.*;
import java.io.File;
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
import java.util.ResourceBundle;
import java.util.Locale;
import java.text.NumberFormat;
import server.Send;

class RMIClientView extends JFrame {
  protected JLabel col1, col2;
  protected JLabel totalItems, totalCost;
  protected JLabel cardNum, custID;
  protected JLabel applechk, pearchk, peachchk;
  protected JButton purchase, reset;
  protected JPanel panel;
  protected JTextField appleqnt, pearqnt, peachqnt;
  protected JTextField creditCard, customer;
  protected JTextArea items, cost;
  protected static Send send;

//Internationalization variables
  private static Locale currentLocale;
  private static ResourceBundle messages;
  private static String language, country;
  private NumberFormat numFormat;

  private RMIClientView(){ //Begin Constructor
    setTitle(messages.getString("title"));

//Create left and right column labels
    col1 = new JLabel(messages.getString("1col"));
    col2 = new JLabel(messages.getString("2col"));

//Create labels and text field components
   applechk = new JLabel("   " + messages.getString("apples"));
    appleqnt = new JTextField();
    pearchk = new JLabel("   " + messages.getString("pears"));
    pearqnt = new JTextField();
    peachchk = new JLabel("   " + messages.getString("peaches"));
    peachqnt = new JTextField();
```

```
    cardNum = new JLabel("   " + messages.getString("card"));
    creditCard = new JTextField();
    pearqnt.setNextFocusableComponent(creditCard);
    customer = new JTextField();
    custID = new JLabel("   " + messages.getString("customer"));

//Create labels and text area components
    totalItems = new JLabel("   " + messages.getString("items"));
    totalCost = new JLabel("   " + messages.getString("cost"));
    items = new JTextArea();
    cost = new JTextArea();

//Create buttons and make action listeners
    purchase = new JButton(messages.getString("purchase"));
    reset = new JButton(messages.getString("reset"));

//Create a panel for the components
    panel = new JPanel();

//Set panel layout to 2-column grid
//on a white background
    panel.setLayout(new GridLayout(0,2));
    panel.setBackground(Color.white);

//Add components to panel columns
//going left to right and top to bottom
    getContentPane().add(panel);
    panel.add(col1);
    panel.add(col2);
    panel.add(applechk);
    panel.add(appleqnt);
    panel.add(peachchk);
    panel.add(peachqnt);
    panel.add(pearchk);
    panel.add(pearqnt);
    panel.add(totalItems);
    panel.add(items);
    panel.add(totalCost);
    panel.add(cost);
    panel.add(cardNum);
    panel.add(creditCard);
    panel.add(custID);
```

```
    panel.add(customer);
    panel.add(reset);
    panel.add(purchase);
  } //End Constructor

  public static void main(String[] args) {
    if (args.length != 3) {
      language = new String("en");
      country = new String ("US");
      System.out.println("English");
    } else {
      language = new String(args[1]);
      country = new String(args[2]);
      System.out.println(language + country);
    }
    currentLocale = new Locale(language, country);
    messages = ResourceBundle.getBundle("client1" +
       File.separatorChar + "MessagesBundle",
       currentLocale);
    WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };

    RMIClientView frame = new RMIClientView();
    frame.addWindowListener(l);
    frame.pack();
    frame.setVisible(true);
    RMIClientController control = new RMIClientController(frame, messages,
currentLocale);
    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(messages.getString("nolookup"));
    } catch (java.rmi.RemoteException e) {
      System.out.println(messages.getString("nolookup"));
```

```
    } catch (java.net.MalformedURLException e) {
      System.out.println(messages.getString("nollokup"));
    }
  }
}
```

RMIClientController Program

```
package client1;

import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.util.ResourceBundle;
import java.util.Locale;
import java.text.NumberFormat;
import server.DataOrder;

class RMIClientController implements ActionListener {
  private RMIClientView frame;
  private ResourceBundle messages;
  private NumberFormat numFormat;
  private Locale currentLocale;

  protected RMIClientController(RMIClientView frame,
             ResourceBundle messages,
             Locale currentLocale){
    this.frame = frame;
    this.messages = messages;
    this.currentLocale = currentLocale;
//Make action listeners
    frame.purchase.addActionListener(this);
    frame.reset.addActionListener(this);
    frame.appleqnt.addActionListener(this);
    frame.peachqnt.addActionListener(this);
    frame.pearqnt.addActionListener(this);
  } //End Constructor

  public void actionPerformed(ActionEvent event) {
   Object source = event.getSource();
   Integer applesNo, peachesNo, pearsNo, num;
   Double cost;
```

```
   String text, text2;
   DataOrder order = new DataOrder();
//If Purchase button pressed  . . .
   if (source == frame.purchase) {
//Get data from text fields
    order.cardnum = frame.creditCard.getText();
    order.custID = frame.customer.getText();
    order.apples = frame.appleqnt.getText();
    order.peaches = frame.peachqnt.getText();
    order.pears = frame.pearqnt.getText();

//Calculate total items
    if (order.apples.length() > 0) {
//Catch invalid number error
      try {
        applesNo = Integer.valueOf(order.apples);
        order.itotal += applesNo.intValue();
      } catch (java.lang.NumberFormatException e) {
        frame.appleqnt.setText(messages.getString("invalid"));
      }
    } else {
      /* else no need to change the total */
    }
    if(order.peaches.length() > 0){

//Catch invalid number error
      try {
        peachesNo = Integer.valueOf(order.peaches);
        order.itotal += peachesNo.intValue();
      } catch(java.lang.NumberFormatException e) {
        frame.peachqnt.setText(messages.getString("invalid"));
      }
    } else {
      /* else no need to change the total */
    }
    if(order.pears.length() > 0){

//Catch invalid number error
      try {
        pearsNo = Integer.valueOf(order.pears);
        order.itotal += pearsNo.intValue();
      } catch (java.lang.NumberFormatException e) {
```

```
        frame.pearqnt.setText(messages.getString("invalid"));
      }
    } else {
      /* else no need to change the total */
    }

//Create number formatter
     numFormat = NumberFormat.getNumberInstance(currentLocale);
//Display running total
     text = numFormat.format(order.itotal);
     frame.items.setText(text);
//Calculate and display running cost
     order.icost = (order.itotal * 1.25);
     text2 = numFormat.format(order.icost);
     frame.cost.setText(text2);
     try {
        frame.send.sendOrder(order);
      } catch (java.rmi.RemoteException e) {
        System.out.println(messages.getString("send"));
      } catch (java.io.IOException e) {
        System.out.println("Unable to write to file");

      }
   }
//If Reset button pressed
//Clear all fields
    if (source == frame.reset) {
     frame.creditCard.setText("");
     frame.appleqnt.setText("");
     frame.peachqnt.setText("");
     frame.pearqnt.setText("");
     frame.creditCard.setText("");
     frame.customer.setText("");
     order.icost = 0;
     cost = new Double(order.icost);
     text2 = cost.toString();
     frame.cost.setText(text2);
     order.itotal = 0;
     num = new Integer(order.itotal);
     text = num.toString();
     frame.items.setText(text);
    }
```

```
  }
}
```

Index

D

E

J

K

L

M

S