Java Programming 4: Java Application Building

Application Building

Welcome to the **Java Application Building** series of Java courses. This series will focus on developing applications using the many tools available in Java. As in all OST courses, the emphasis will be on interactive instruction.

Course Objectives

When you complete this course, you will be able to:

- enhance Graphical User Interfaces in Java using views, frames, panels, and Swing.
- implement error checking, exception handling, and try/catch clauses to minimize bugs.
- catch unchecked exceptions and prepare for problems through graceful degradation.
- create and manipulate threads for concurrent programming.
- connect with databases using the JDBC API, factory design patterns, and view controllers.
- document and tag code using Javadoc and API pages.

In this course, you will achieve an understanding of the structure and purposes for many of the classes in the Java API. In-depth experience with user-interfaces, event and exception handling, database connectivity, multiple threads and synchronization will provide you with a toolkit for both implementing applications as well as understanding source code of others. Programs designed in the course using Java Threads, Client/Server Sockets and Database Connectivity provide a solid basis for application building.

From beginning to end, you will learn by doing your own Java projects, within our Eclipse Learning Sandbox we affectionately call "Ellipse". These projects will add to your portfolio and provide needed experience. Besides a browser and internet connection, all software is provided online by the O'Reilly School of Technology.

Review

In this series, we assume that you have a general foundation of Java and object-oriented programming knowledge, so basic programming skills won't be covered in this course. We'll work now to grow and refine your existing Java programming skills. If you are unable to follow the code that we use for illustrations and examples in this course, we recommend that you take OST's first Java series of courses to gain those basic programming skills; then you'll be able to reap the full benefits of the materials presented here.

If you are new to OST courses, read this overview before you go further.

In the previous course, we went over these concepts:

- Code flexibility
- Package declaration and usage
- Separation of classes to model the Model/View/Controller (MVC) design pattern
- Interfaces
- Casting
- Declaration and use of inner classes

We'll apply these ideas and more as we improve our Sales Report application.

Preview

In the first new version of our application, we'll bring in **LayoutManager**s to provide a better user interface. Except for the various **LayoutManager**s and **Panel**s, we'll also be using familiar code and techniques. In upcoming lessons, we will learn additional techniques that will enable us to:

- provide exception handling.
- make better GUIs.
- allow application deployment using jars and executables.
- add input from other sources (such as databases and "offsite" machines).
- allow multiple people to use the same applications at the same time (threads).

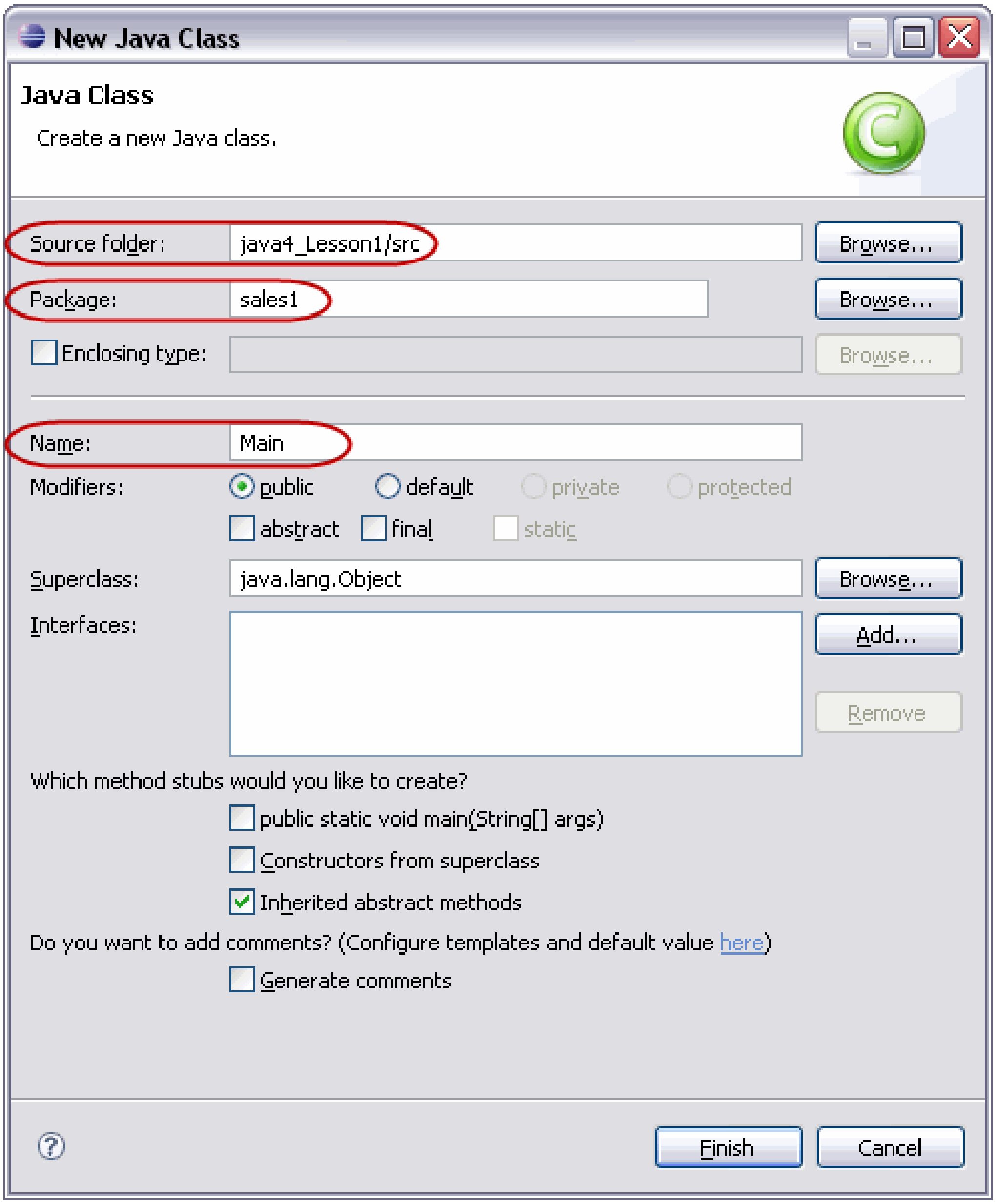
- add other useful features to our applications.

Ready? Alright then!

Improving Your Code

Let's get to work using the same application we developed in the previous series. The application prompts users for the number of salespeople and their sales performance figures, and then displays the top performer. (If you took the earlier course, **type** the code in and run it.)

Create a new **java4_Lesson1** project (it may help to take a look at the overview). Now, create a new **main** class in this project as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson1/src

Package: sales1

□ Enclosing type:

Name: Main

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

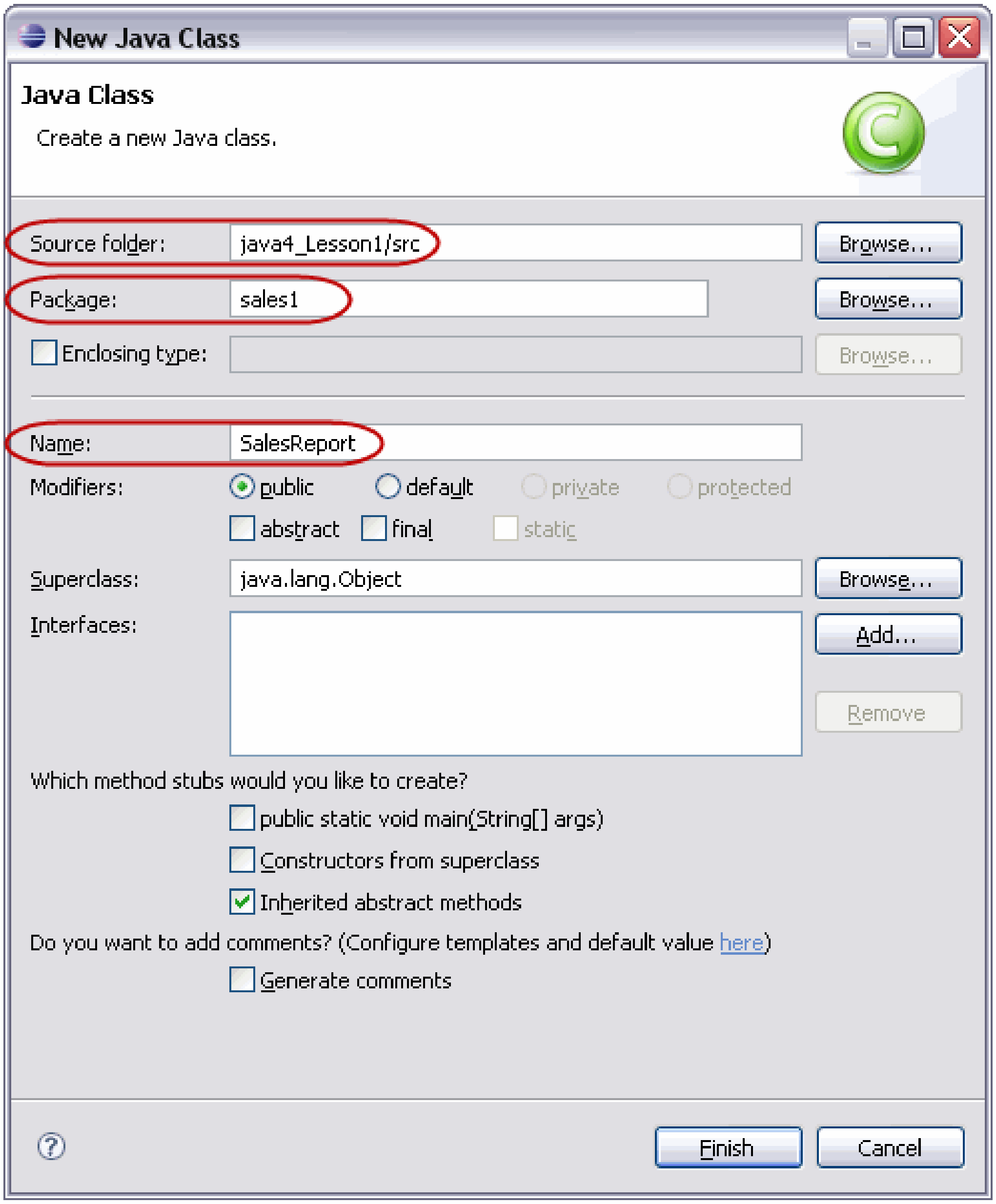
Type **main** as shown in **blue**:

CODE TO TYPE: Main

```
// ****************************************************************
// Main.java
//
// Instantiates and starts the SalesReport class
//
// ****************************************************************
package sales1;

public class Main {

    public static void main(String[] args){
        if (args.length > 0)
        {
            int argIn = Integer.parseInt(args[0]);
            SalesReport mySalesInfo = new SalesReport(argIn);
            mySalesInfo.testMe();
        }
        else
        {
            SalesReport mySalesInfo = new SalesReport();  // instantiate the class
            mySalesInfo.testMe();                        // start the application
        }
    }
}
```

This class instantiates and starts our application. There will be errors in **Main**, because we still haven't created the class that it instantiates.

In java4_Lesson1, create the **SalesReport** class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson1/src

Package: sales1

□ Enclosing type:

Name: SalesReport

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

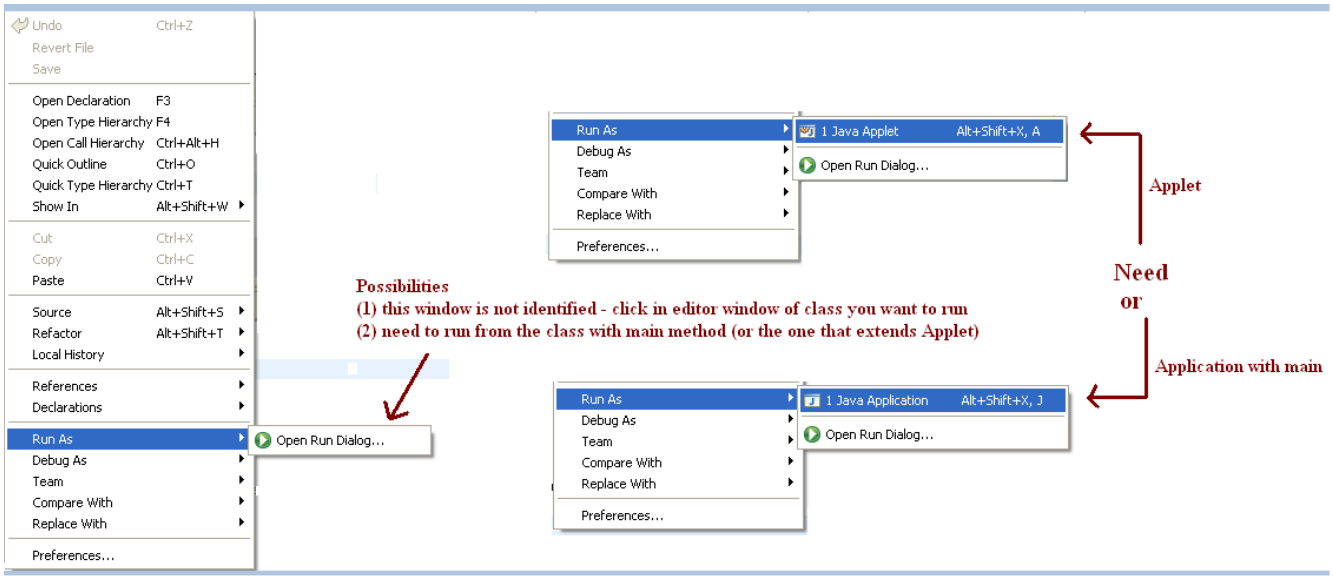
□ Generate comments

Type **SalesReport** as shown in **blue** below:

CODE TO TYPE: SalesReport

```java
package sales1;

import java.util.Scanner;

public class SalesReport{
    int SALESPEOPLE;
    int sum;
    int sales[];
    Scanner scan = new Scanner(System.in);

    public SalesReport(){
        System.out.print("Enter the number of salespersons: ");
        this.SALESPEOPLE = scan.nextInt();
        this.sales = new int[SALESPEOPLE];
    }

    public SalesReport(int howMany){
        this.SALESPEOPLE = howMany;
        this.sales = new int[SALESPEOPLE];
    }

    public void testMe(){
        getSalesInput();
        provideSalesOutput();
        findMax();
    }

    public void getSalesInput(){
        Scanner scan = new Scanner(System.in);

        for (int i=0; i < sales.length;  i++)
        {
            System.out.print("Enter sales for salesperson " + (i+1) + ": ");
            sales[i] = scan.nextInt();
        }
    }

    public void provideSalesOutput(){
         System.out.println("\nSalesperson   Sales");
         System.out.println("--------------------");
         sum = 0;
         for (int i=0; i < sales.length;  i++)
         {
             System.out.println("     " + (i+1) + "         " + sales[i]);
             sum = sum + sales[i];
         }
         System.out.println("\nTotal sales: " + sum);
    }

    public void findMax(){
        int max = sales[0];  // this way we are assured that value for the initial max
is in the collection
        int who = 0;         // and the initial index is the first so we visit all
        for (int i=0; i < sales.length;  i++)
        {
            if (max < sales[i])
            {
                max = sales[i];
                who = i;
            }
        }
        System.out.println("\nSalesPerson " + (who+1) + " had the highest sale with $"
+ max );
    }
}
```

Save and Run it.



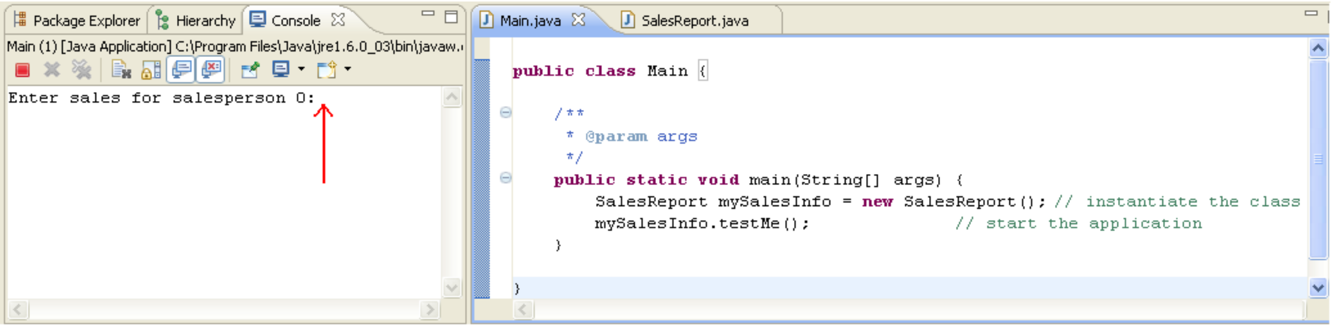
Possibilities
(1) this window is not identified - click in editor window of class you want to run
(2) need to run from the class with main method (or the one that extends Applet)

Since **SalesReport** does not extend **Applet**, it is not an Applet, and since **SalesReport** does not have a **main()** method, Java is not sure what to do. We created the **Main** class to instantiate and start this application, so we should go to that class to run it.

Note In Eclipse, if you choose **Run As**, but neither **Java Applet** nor **Java Application** appear as options, click *in the Editor Window*. This lets Eclipse know that you are running the .java file.

Click on the **Main.java** class. (**SalesReport** is no longer unknown, because we have defined it now.)

Save and Run it. The Console opens and is ready for you to provide input:


Package Explorer
Hierarchy
Console
Main (1) [Java Application] C:\Program Files\Java\jre1.6.0_03\bin\javaw.e
Main.java
SalesReport.java

```
Enter sales for salesperson 0:
```

```
public class Main {

    /**
     * @param args
     */
    public static void main(String[] args) {
        SalesReport mySalesInfo = new SalesReport(); // instantiate the class
        mySalesInfo.testMe();                // start the application
    }

}
```

Click in the console window, type **2**, and press **Enter**. You're asked for sales numbers for salespersons 1 and 2--enter any number for each and press **Enter**. Trace the code from the instantiation in **Main()**.

The code works fine, but we can improve it. Previously, we wrote code to find the average and minimum sales, and to allow the user to set a number as a goal and determine which salespeople reached this goal. The new versions of the **SalesReport** application we'll write in this course will call upon many of the added potentials we learned earlier.

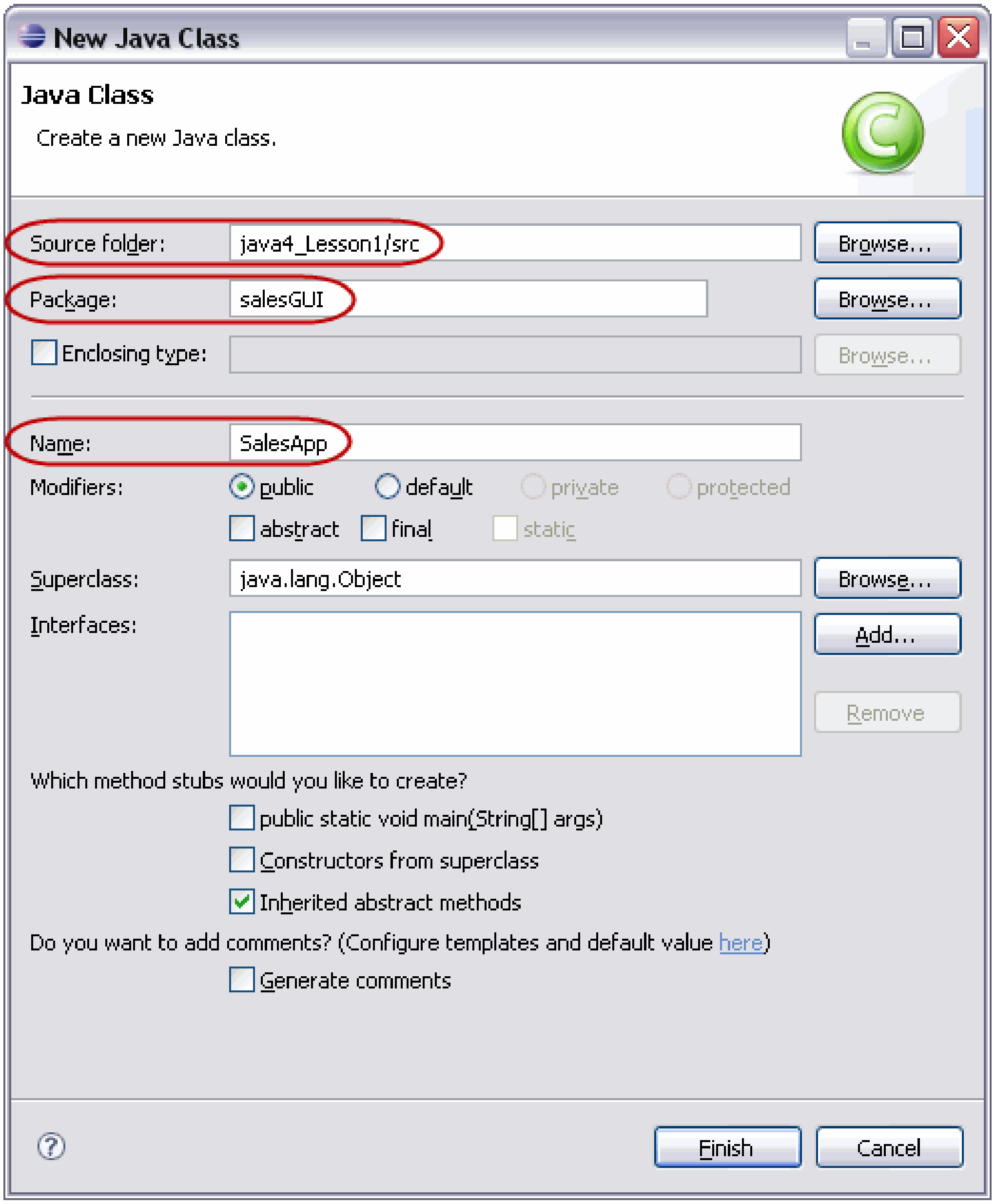
Design Pattern: Model/View/Controller

A common and well-known design pattern in object-oriented programming involves separating the various components of an application based on function. In keeping with object-oriented principles of modularity and the MVC design pattern, we'll separate the components in our construction. First, we will make the application, or *Model* class. The model class holds the code that defines a particular application. The application functionality required for **SalesReport** was the ability to determine the average sales, the minimum and maximum sales, and which salespeople surpassed a goal set by the user.

Separating Application from User Interface

Let's put our new version of the application in a different package.

In the java4_Lesson1 project, add a new class as shown:



Source folder: java4_Lesson1/src

Package: salesGUI

☐ Enclosing type:

Name: SalesApp

Modifiers: ● public ○ default ○ private ○ protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Type **SalesApp** as shown in **blue**:

CODE TO TYPE: SalesApp

```
package salesGUI;

public class SalesApp {

    //array to hold sales of each salesperson
    private int[] sales;
    //variable for sales goal (to be established by user)
    private int salesBar;
    private int totalSales;
    //why not average = totalSales/sales.length; here?
    private double average;
    private int minIndex = 0;
    private int maxIndex = 0;
    SalesUserInterface myUserInterface;
}
```

Save it.

OBSERVE:

```
private int[] sales;
private int salesBar;
private int totalSales;
private double average;
private int minIndex = 0;
private int maxIndex = 0;
SalesUserInterface myUserInterface;
```

Here we set up variables for the **model** of our Sales Report Application. We created the sales array **sales** to hold each salesperson's sales figures. The **salesBar** variable will hold our sales goal. The **totalSales** and **average** variables will keep the totals and average sales. The **minIndex** and **maxIndex** will hold the locations in the **sales** array for the minimum sales and maximum sales.

SalesUserInterface myUserInterface is a reference to the GUI for the *SalesUserInterface* application that we'll be making in Lesson 3. We'll use it to send information to and from the GUI.

Now add the rest of the setters for the private variables. Add the **blue** code as shown:

CODE TO EDIT: SalesApp

```java
package salesGUI;

public class SalesApp {

    //array to hold sales of each salesperson
    private int[] sales;
    //variable for sales goal (to be established by user)
    private int salesBar;
    //sales of all sales people together
    private int totalSales;
    //why not average = totalSales/sales.length; here?
    private double average;
    private int minIndex = 0;
    private int maxIndex = 0;
    SalesUserInterface myUserInterface;

    public void setMyUserInterface(SalesUserInterface myGUI){
        myUserInterface = myGUI;
    }

    public void setSales(int[] sales) {
        this.sales = sales;
        for (int i = 0; i < sales.length; i++)
            // checking to see if it's working
            System.out.println(sales[i]);
        // data consistency
        setTotalSales();
    }

    public void setTotalSales() {
        totalSales = 0;
        for (int x = 0; x < sales.length; x++)
            totalSales += sales[x];
        setAverage(); // data consistency
    }

    public void setAverage() {
        if (sales.length != 0)
            average = (double) (totalSales / sales.length);
        System.out.println("totalSales is " + totalSales + " and sales.length is
 "
            + sales.length + " making average "
            + ((double) totalSales / sales.length));
    }

    public void setSalesBar(int goal){
        salesBar = goal;
    }
}
```

Save it.

OBSERVE: Setters

```
public void setMyUserInterface(SalesUserInterface myGUI)){
    myUserInterface = myGUI;
}

public void setSales(int[] sales) {
    this.sales = sales;
    for (int i = 0; i < sales.length; i++)
        // just checking to see if working
        System.out.println(sales[i]);
    // data consistency
    setTotalSales();
}

public void setTotalSales() {
    totalSales = 0;
    for (int x = 0; x < sales.length; x++)
        totalSales += sales[x];
    setAverage(); // data consistency
}

public void setAverage() {
    if (sales.length != 0)
        average = (double) (totalSales / sales.length);
    System.out.println("totalSales is " + totalSales + " and sales.length is "
        + sales.length + " making average "
        + ((double) totalSales / sales.length));
}

public void setSalesBar(int goal){
    salesBar = goal;
}
```

The five **methods** above are *setters* for the variables **sales**, **totalSales**, and **average**. They are chained together; if we call **setSales()**, it calls **setTotalSales()**, which in turn calls **setAverage()**. This ensures that when we set the sales, the totalSales and average are up to date and consistent with the current sales array data. Finally, we set the salesBar variable with **setSaleBar(int goal)**. The *goal* will be an integer that is set by the end user when we build our User Interface in a future lesson.

Now, add the getters. Add the code shown in **blue**:

CODE TO EDIT: SalesApp

```
package salesGUI;

public class SalesApp {

    //array to hold sales of each salesperson
    private int[] sales;
    //variable for sales goal (to be established by user)
    private int salesBar;
    private int totalSales;
    //why not average = totalSales/sales.length; here?
    private double average;
    private int minIndex = 0;
    private int maxIndex = 0;
    SalesUserInterface myUserInterface;

    public void setMyUserInterface(SalesUserInterface myGUI){
        myUserInterface = myGUI;
    }

    public void setSales(int[] sales) {
        this.sales = sales;
        for (int i = 0; i < sales.length; i++)
            // just checking to see if working
            System.out.println(sales[i]);
        // data consistency
        setTotalSales();
    }

    public void setTotalSales() {
        totalSales = 0;
        for (int x = 0; x < sales.length; x++)
            totalSales += sales[x];
        setAverage(); // data consistency
    }

    public void setAverage() {
        if (sales.length != 0)
            average = (double) (totalSales / sales.length);
        System.out.println("totalSales is " + totalSales + " and sales.length is
 "
            + sales.length + " making average "
            + ((double) totalSales / sales.length));
    }

    public void setSalesBar(int goal){
        salesBar = goal;
    }

    public int[] getSales() {
        return sales;
    }

    public double getAverage() {
        if (sales.length != 0)
            // cast so does not truncate int division
            return ((double) totalSales / sales.length);
        else
            return average;
    }

    public int getBar() {
        return salesBar;
    }

    public int getTotalSales() {
        return totalSales;
    }

    public int getMin() {
        return minIndex;
    }

    public int getMax() {
        return maxIndex;
    }
}
```

💾 Save it.

Let's take a closer look at the **getAverage()** getter:

OBSERVE

```
public double getAverage() {
    if (sales.length != 0)
        return ((double) totalSales / sales.length);
    else
        return average;
}
```

If the sales array length (the user-entered number of salespersons) is not 0, **getAverage()** calculates the average before returning its value; otherwise, it returns the value of the **average** variable.

Now create a method that calculates the minimum and maximum sales, using comparisons. Add the code shown in **blue**:

CODE TO EDIT: SalesApp

```
package salesGUI;

public class SalesApp {

    //array to hold sales of each salesperson
    private int[] sales;
    //variable for sales goal (to be established by user)
    private int salesBar;
    private int totalSales;
    //why not average = totalSales/sales.length; here?
    private double average;
    private int minIndex = 0;
    private int maxIndex = 0;
    SalesUserInterface myUserInterface;

    public void setMyUserInterface(SalesUserInterface myGUI){
        myUserInterface = myGUI;
    }

    public void setSales(int[] sales) {
        this.sales = sales;
        for (int i = 0; i < sales.length; i++)
            // just checking to see if working
            System.out.println(sales[i]);
        // data consistency
        setTotalSales();
    }

    public void setTotalSales() {
        totalSales = 0;
        for (int x = 0; x < sales.length; x++)
            totalSales += sales[x];
        setAverage(); // data consistency
    }

    public void setAverage() {
        if (sales.length != 0)
            average = (double) (totalSales / sales.length);
        System.out.println("totalSales is " + totalSales + " and sales.length is
 "
            + sales.length + " making average "
            + ((double) totalSales / sales.length));
    }

    public void setSalesBar(int goal){
        salesBar = goal;
    }

    public int[] getSales() {
        return sales;
    }

    public double getAverage() {
        if (sales.length != 0)
            // cast so does not truncate int division
            return ((double) totalSales / sales.length);
        else
            return average;
    }

    public int getBar() {
        return salesBar;
    }

    public int getTotalSales() {
        return totalSales;
    }

    public int getMin() {
        return minIndex;
    }

    public int getMax() {
        return maxIndex;
    }

    public void calculateMinMax() {
        int minimum = sales[0];
        int maximum = sales[0];
        // loop through the sales array to see each sales amount
        for (int x = 0; x < sales.length; x++) {
            //Check for max sale
            if (sales[x] > maximum) {
                maximum = sales[x];
                maxIndex = x;
            }
            else if (sales[x] < minimum) //Check for min sale
            {
                minimum = sales[x];
                minIndex = x;
            }
        }
        System.out.println("Maximum value is at index " + maxIndex
            + " (Salesperson " + (maxIndex + 1) + ") with value " + maximum);
        System.out.println("Minimum value is at index " + minIndex
            + " (Salesperson " + (minIndex + 1) + ") with value " + minimum);
        setAverage();
    }
}
```

Save it.

The calculateMinMax() method

```
public void calculateMinMax() {
    int minimum = sales[0];
    int maximum = sales[0];
    // loop through the sales array to see each sales amount
    for (int x = 0; x < sales.length; x++) {
        //Check for max sale
        if (sales[x] > maximum) {
            maximum = sales[x];
            maxIndex = x;
        }
        else if (sales[x] < minimum) //Check for min sale
        {
            minimum = sales[x];
            minIndex = x;
        }
    }
    System.out.println("Maximum value is at index " + maxIndex
        + " (Salesperson " + (maxIndex + 1) + ") with value " + maximum);
    System.out.println("Minimum value is at index " + minIndex
        + " (Salesperson " + (minIndex + 1) + ") with value " + minimum);
    setAverage();
}
```

The **calculateMaxMin()** sets the index of the maximum (**maxIndex**) and minimum (**minIndex**) values in the sales array. We set local variables **minimum** and **maximum** to the value in the sales[0] element as a starting point, then loop through the array. If the value in a particular index is greater than the previous maximum, we set maximum to that value. If the value in a particular index is not greater than the previous maximum, we check to see if the value is less than the previous minimum, and if so, we set minimum to the new value. We also keep track of the location of the indexes that contain the current minimum (**minIndex**) and maximum (**maxIndex**) values in the array.

Okay, now we'll add a method to determine who the top sales people are, so we can praise them and then give them even more work! Edit your code as shown in **blue**:

CODE TO EDIT: SalesApp

```
package salesGUI;

public class SalesApp {

    //array to hold sales of each salesperson
    private int[] sales;
    //variable for sales goal (to be established by user)
    private int salesBar;
    private int totalSales;
    //why not average = totalSales/sales.length; here?
    private double average;
    private int minIndex = 0;
    private int maxIndex = 0;
    SalesUserInterface myUserInterface;

    public void setMyUserInterface(SalesUserInterface myGUI){
        myUserInterface = myGUI;
    }

    public void setSales(int[] sales) {
        this.sales = sales;
        for (int i = 0; i < sales.length; i++)
            // just checking to see if working
            System.out.println(sales[i]);
        // data consistency
        setTotalSales();
    }

    public void setTotalSales() {
        totalSales = 0;
        for (int x = 0; x < sales.length; x++)
            totalSales += sales[x];
        setAverage(); // data consistency
    }

    public void setAverage() {
        if (sales.length != 0)
            average = (double) (totalSales / sales.length);
        System.out.println("totalSales is " + totalSales + " and sales.length is
 "
            + sales.length + " making average "
            + ((double) totalSales / sales.length));
    }

    public void setSalesBar(int goal){
        salesBar = goal;
    }

    public int[] getSales() {
        return sales;
    }

    public double getAverage() {
        if (sales.length != 0)
            // cast so does not truncate int division
            return ((double) totalSales / sales.length);
        else
            return average;
    }

    public int getBar() {
        return salesBar;
    }

    public int getTotalSales() {
        return totalSales;
    }

    public int getMin() {
        return minIndex;
    }

    public int getMax() {
        return maxIndex;
    }

    public void calculateMinMax() {
        int minimum = sales[0];
        int maximum = sales[0];
        // loop through the sales array to see each sales amount
        for (int x = 0; x < sales.length; x++) {
            //Check for max sale
            if (sales[x] > maximum) {
                maximum = sales[x];
                maxIndex = x;
            }
            else if (sales[x] < minimum) //Check for min sale
            {
                minimum = sales[x];
                minIndex = x;
            }
        }
        System.out.println("Maximum value is at index " + maxIndex
            + " (Salesperson " + (maxIndex + 1) + ") with value " + maximum);
        System.out.println("Minimum value is at index " + minIndex
            + " (Salesperson " + (minIndex + 1) + ") with value " + minimum);
        setAverage();
    }

    //method returns performance array to indicate success at reaching goal
    public int[] determineTopSalesPeople() {
        // System.out prints to console to be sure we got here--debugging tool
        System.out.println("I'm here and salesBar is " + salesBar);

        // an array with values of -1, 0, 1 to indicate success at reaching goal
        int[] performance = new int[sales.length];

        // Loop through the sales array and see who sold more than the sales bar
        for (int x = 0; x < sales.length; x++)
        {
            if (sales[x] > salesBar) {
                performance[x] = 1;
            }
            else if (sales[x] == salesBar) {
                performance[x] = 0;
            }
            else {
                performance[x] = -1;
            }
        }
        return performance;
    }
}
```

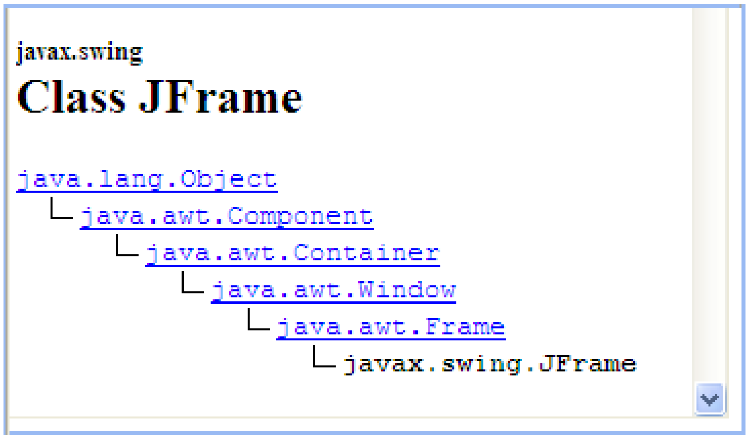
Save it.

The method **determineTopSalesPeople()** will return the top-performing salespeople in the sales array. It returns an integer array, associated with the sales array. If a salesperson's performance is below the salesBar, then we place a -1 in the corresponding slot of the integer array. If performance is equal to the salesBar, then we place a 0 in that slot. And if a salesperson's performance is above the salesBar, then we place a +1 in that slot.

Coming Attractions: The View

The "View" is exactly that: the View or GUI used to interact with the Model.

Eventually we'll create the Graphical User Interface (GUI) using Swing components, so in the next lesson we'll cover some Swing basics. Using **javax.swing** package is similar to using **java.awt** components. In fact, Swing components usually inherit from the awt components:

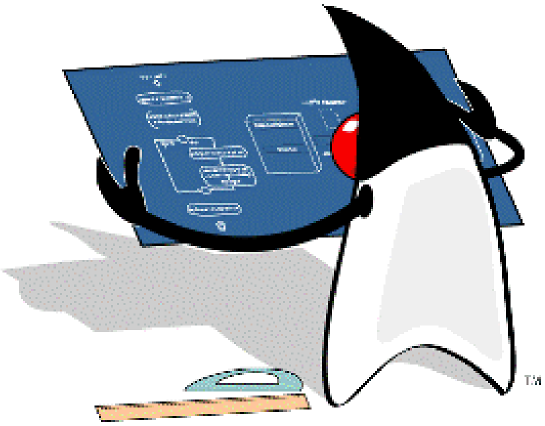


javax.swing

Class JFrame

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └java.awt.Window
              └java.awt.Frame
                  └javax.swing.JFrame
```

Duke will help you work out the design.



Swing: A Very Brief Overview

AWT vs. Swing

This lesson will give you a brief overview of the Swing package, and in particular, compare the AWT package to the Swing package. The similarities between the components of these two packages will allow you to assimilate the new material and ultimately incorporate more options to control the appearance of your GUI.

According to the freejavaguide.com page on Java Swing: Free Java Tutorials:

***Java Swing** is a GUI toolkit for Java. Swing is one part of the Java Foundation Classes (JFC). Swing includes graphical user interface (GUI) widgets such as text boxes, buttons, split-panes, and tables.*

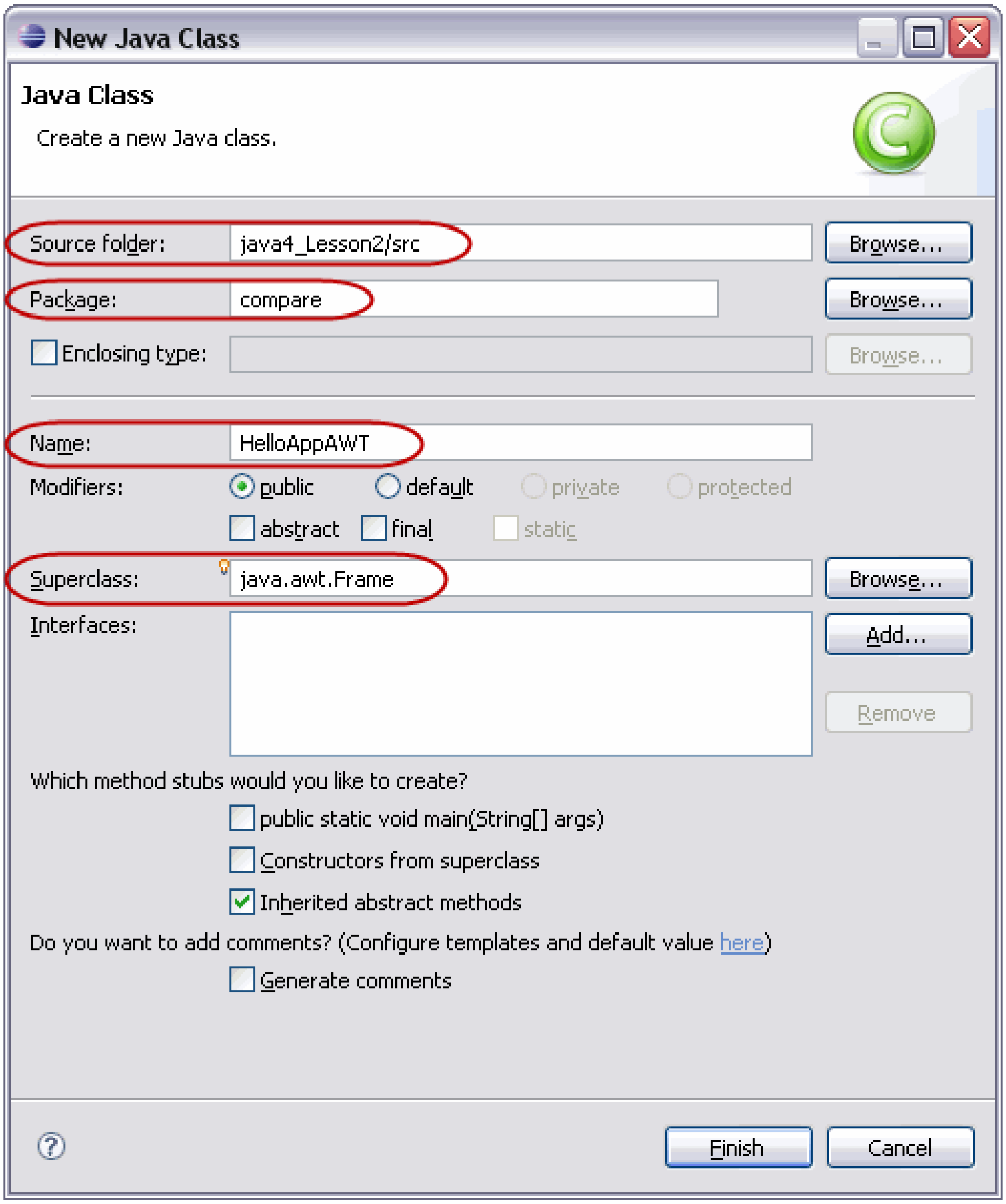
Swing widgets provide more sophisticated GUI components than the earlier Abstract Window Toolkit. Since they are written in pure Java, they run the same on all platforms, unlike the [first] AWT which is tied to the underlying platform's windowing system. Swing supports pluggable look and feel – not by using the native platform's facilities, but by roughly emulating them. This means you can get any supported look and feel on any platform. The disadvantage of lightweight components is possibly slower execution. The advantage is uniform behavior on all platforms.

Note Please note, where possible, we have updated our links to point to the new Oracle site for Java. Oracle bought Sun Microsystems some time ago. Some of Oracle's links point to locations that no longer exist. We have no contol over that. We are sorry for any inconvenience. If you are directed to the **java.sun.com** domain from our course, it is because we could not find a corresponding **oracle.com** URL for that particular resource. Oracle has indicated that they want to shut down **java.sun.com**; however, they have, at least for the time being, delayed that decision, partly due to outcry from the Java community.

HelloWorld in AWT and Swing

To illustrate the similarities between AWT and Swing, we'll use a HelloWorld **Frame** and **JFrame**. Later we'll use a HelloWorld **JApplet**, which makes use of **Threads**. Create a new **java4_Lesson2** project. If you're given the option to "Open Associated Perspective", click **No**.

In your new project, create a **HelloAppAWT** class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson2/src

Package: compare

□ Enclosing type:

Name: HelloAppAWT

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.awt.Frame

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **HelloAppAWT** as shown in **blue**:

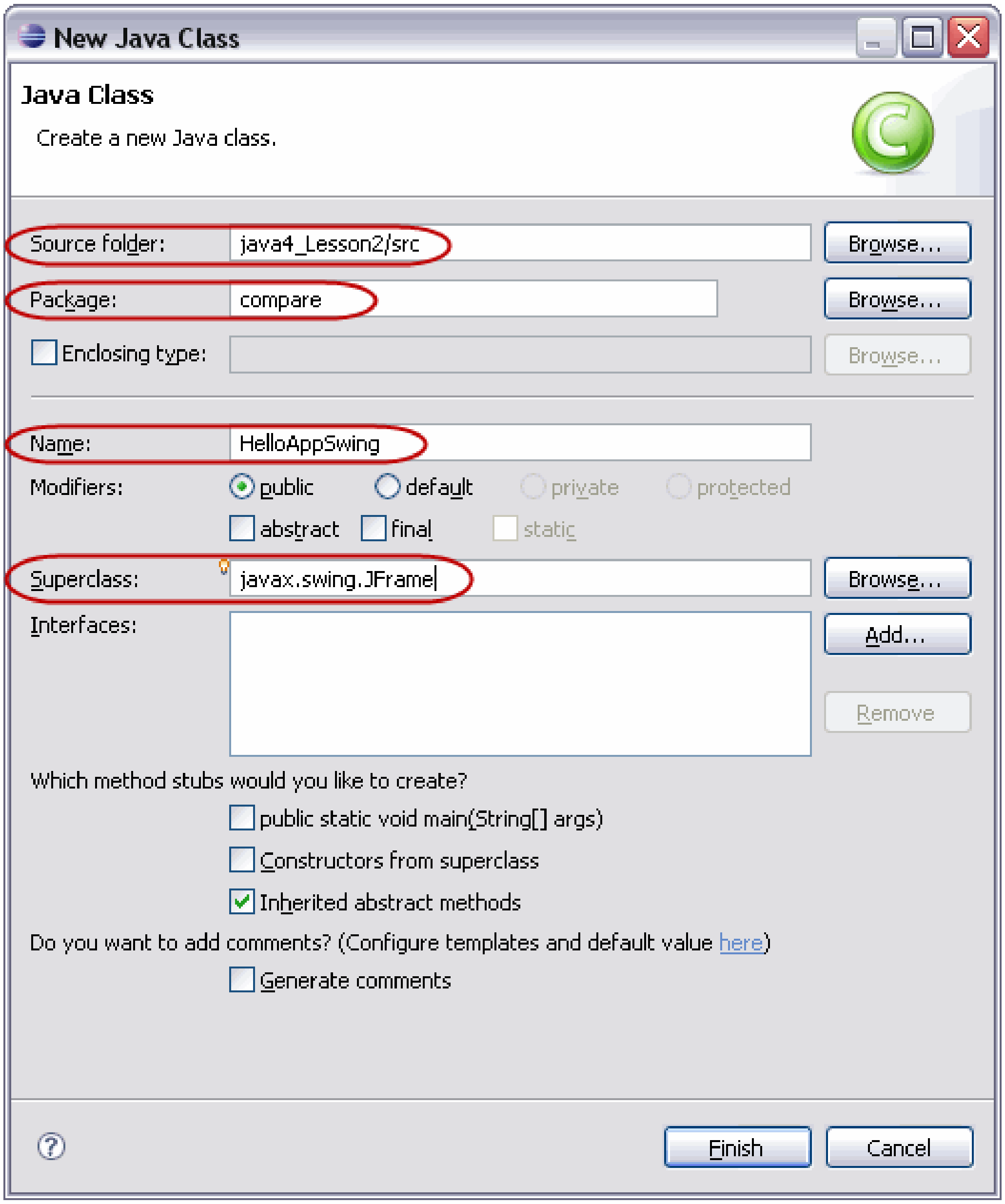
CODE TO TYPE: HelloAppAWT

```
package compare;

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

public class HelloAppAWT extends Frame {
    public HelloAppAWT() {
        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
        }});
        add(new Label("Hello, world!"));
        pack();
    }

    public static void main(String[] args) {
        new HelloAppAWT().setVisible(true);
    }
}
```

Save and run it.

Now we'll write the small application in Swing. In the comments, you can see how it differs from AWT.

In the java4_Lesson2 project, add the **HelloAppSwing** class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson2/src

Package: compare

□ Enclosing type:

Name: HelloAppSwing

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: javax.swing.JFrame

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

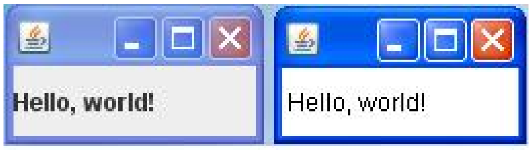
□ Generate comments

Type **HelloAppSwing** as shown in **blue** below:

CODE TO TYPE: HelloAppSwing

```
package compare;

import javax.swing.*;

public class HelloAppSwing extends JFrame {
    public HelloAppSwing() {
        setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
        add(new JLabel("Hello, world!"));
        pack();
    }

    public static void main(String[] args) {
        new HelloAppSwing().setVisible(true);
    }
}
```

Save and run it.


Hello, world!

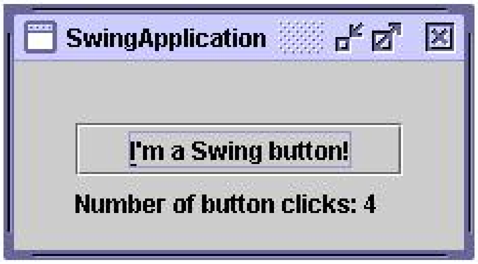
Hello, world!

Can you tell which is which?

Let's take a look at the **import** in our Swing example. The first line imports only the main Swing package: **import javax.swing.*** This is the only package that your application needs. However, if your application had any **Listeners** (for user input), your Swing program might have also needed to import the AWT packages **java.awt.*** and **java.awt.event.***. These packages are often required because Swing components use the AWT infrastructure, including the AWT event model as well. They use the same Listeners and Listener API Tables.

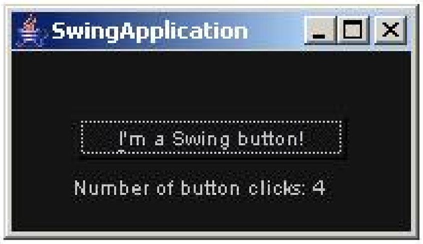
Changing Appearance

Even though the differences in appearance are often subtle, you'll still want to control what your GUI's look like. You can use any of these four platform types:



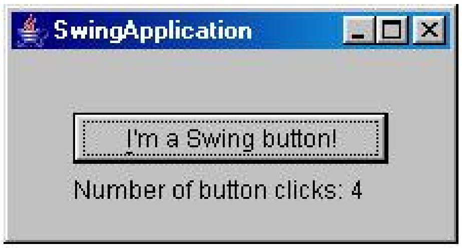
Number of button clicks: 4

Java look and feel



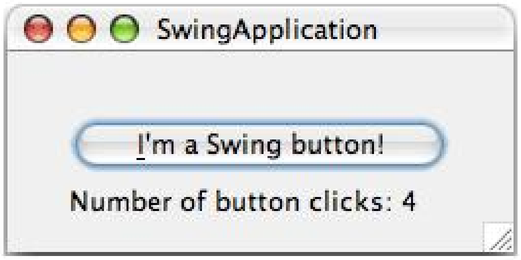
Number of button clicks: 4

GTK+ look and feel



Number of button clicks: 4

Windows look and feel

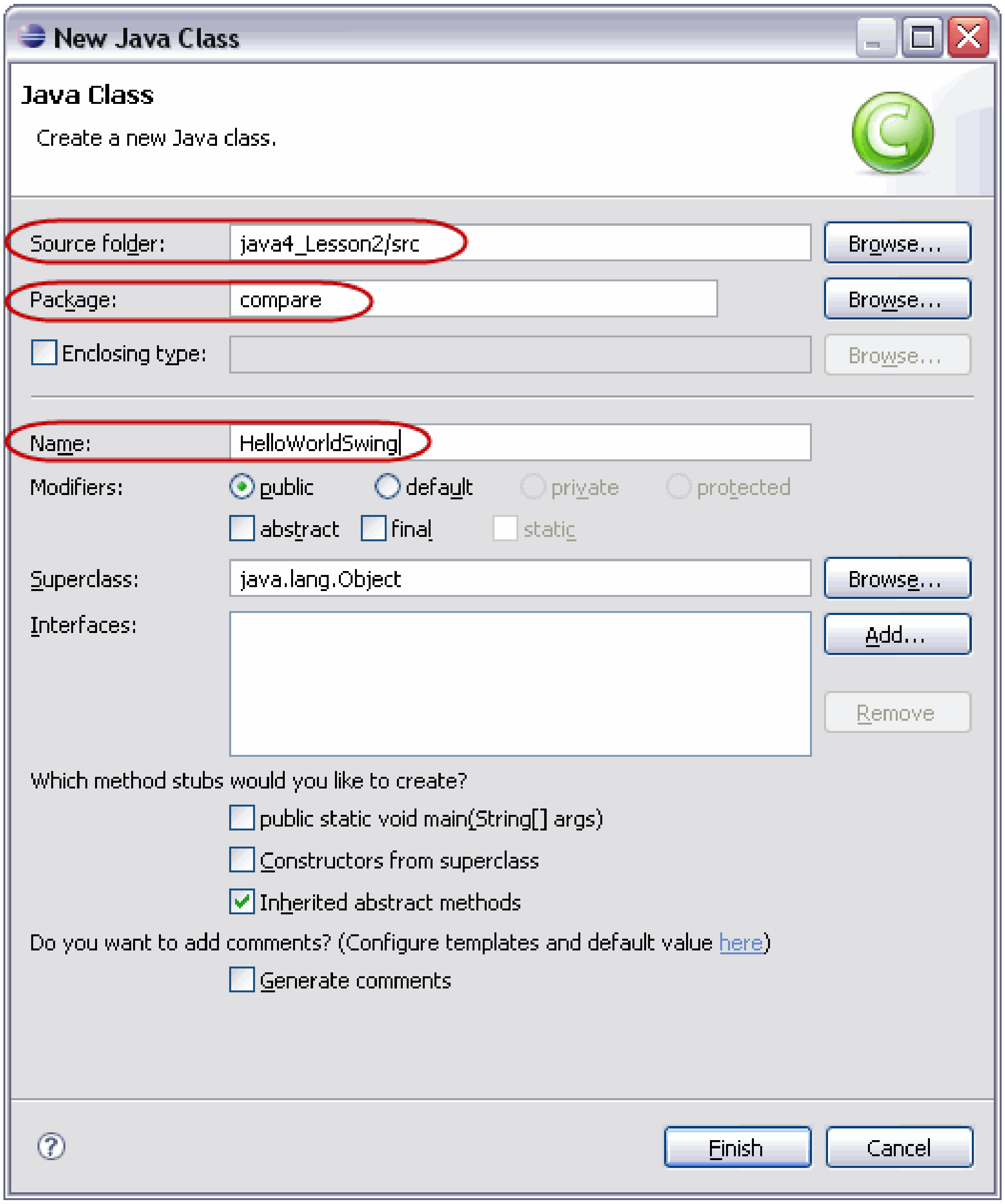


Number of button clicks: 4

Mac OS look and feel

The Swing tutorial has an example of a more decked out GUI. It is replicated exactly here so that you can see the comments and the copyright notice as well (you don't need to type the copyright notice though).

In the java4_Lesson2 project, add a **HelloWorldSwing** class as shown:



Java Class

Create a new Java class.

Source folder: java4_Lesson2/src

Package: compare

□ Enclosing type:

Name: HelloWorldSwing

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **HelloWorldSwing** as shown in **blue** below:

CODE TO TYPE: HelloWorldSwing

```
package compare;

import javax.swing.*;

public class HelloWorldSwing {
    /**
     * Create the GUI and show it.  For thread safety, this method should be invoked fr
om the
     * event-dispatching thread.
     */
    private static void createAndShowGUI() {
        //Make sure we have nice window decorations.
        JFrame.setDefaultLookAndFeelDecorated(true);

        //Create and set up the window.
        JFrame frame = new JFrame("HelloWorldSwing");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Add the "Hello World" label.
        JLabel label = new JLabel("Hello World");
        frame.getContentPane().add(label);

        //Display the window.
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        //Schedule a job for the event-dispatching thread:
        //creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

Save and run it. You might need to resize the window. Now, THAT looks different.

API Go to the **javax.swing.JFrame** class in the API. Look at the **setDefaultLookAndFeelDecorated()** method. Click on it to see the detailed description, and read the discussion of the **LookAndFeel**.

To learn more about LookAndFeel for **JFrame**s, see How to Make Frames (Main Windows), Using Swing Components, and also the page on Pluggable Look and Feel.

JApplets, JFrames, and Threads

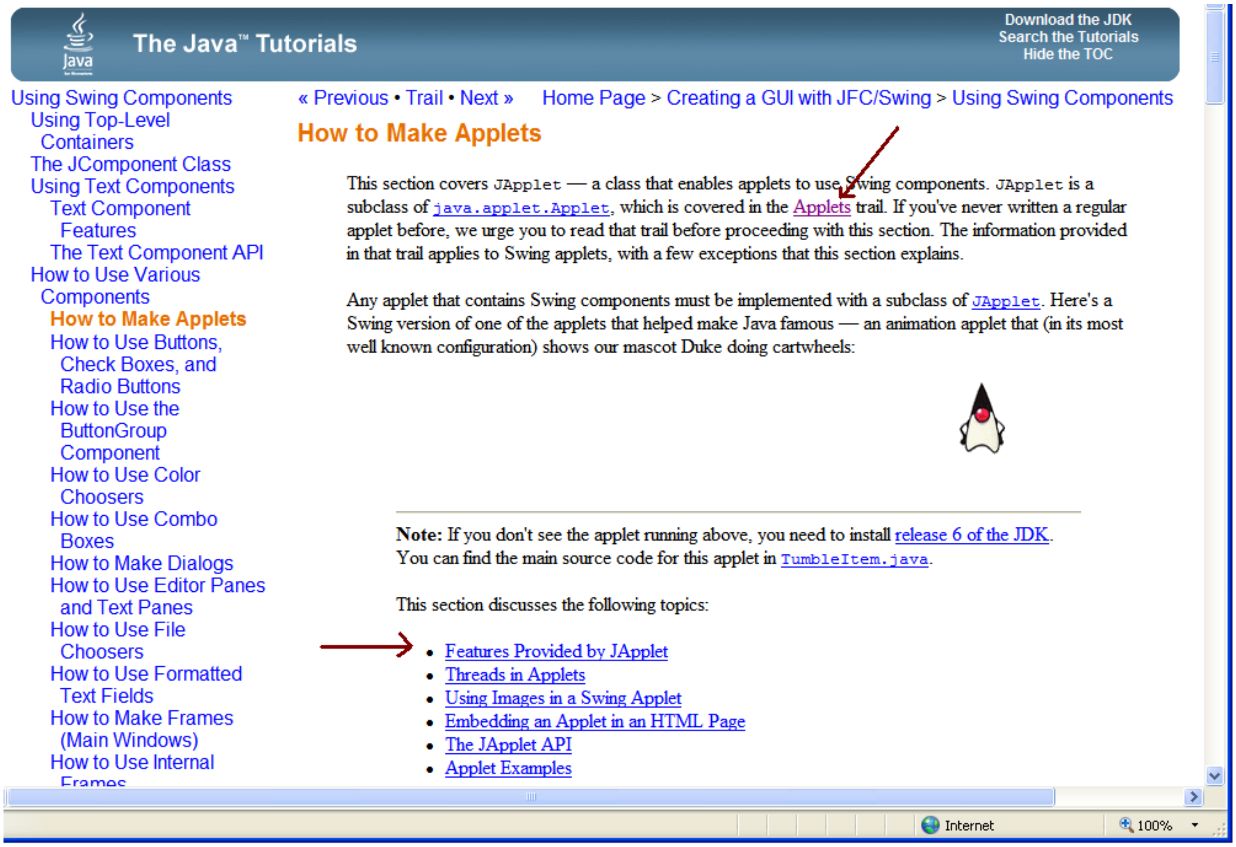
So, what was going on with the **main()** method and **javax.swing.SwingUtilities.invokeLater()** call with the **Runnable** interface parameter?

In our first examples with the AWT and Swing applications, we did not use a **Runnable** interface to access another thread from our **main()** method. This can lead to race conditions in the class's **constructors** and/or **init()** methods.

Because of these differences between the Swing and the AWT packages, Oracle suggests making **JFrames** for applications and **JApplets** differently.

More Information on Applets

Oracle's Swing Tutorial link, How to Make Applets, is really useful:


The Java™ Tutorials
Download the JDK
Search the Tutorials
Hide the TOC
Using Swing Components
Using Top-Level Containers
The JComponent Class
Using Text Components
Text Component Features
The Text Component API
How to Use Various Components
How to Make Applets
How to Use Buttons, Check Boxes, and Radio Buttons
How to Use the ButtonGroup Component
How to Use Color Choosers
How to Use Combo Boxes
How to Make Dialogs
How to Use Editor Panes and Text Panes
How to Use File Choosers
How to Use Formatted Text Fields
How to Make Frames (Main Windows)
How to Use Internal Frames
« Previous • Trail • Next »
Home Page > Creating a GUI with JFC/Swing > Using Swing Components
Internet
100%

How to Make Applets

This section covers `JApplet` — a class that enables applets to use Swing components. `JApplet` is a subclass of `java.applet.Applet`, which is covered in the Applets trail. If you've never written a regular applet before, we urge you to read that trail before proceeding with this section. The information provided in that trail applies to Swing applets, with a few exceptions that this section explains.

Any applet that contains Swing components must be implemented with a subclass of `JApplet`. Here's a Swing version of one of the applets that helped make Java famous — an animation applet that (in its most well known configuration) shows our mascot Duke doing cartwheels:

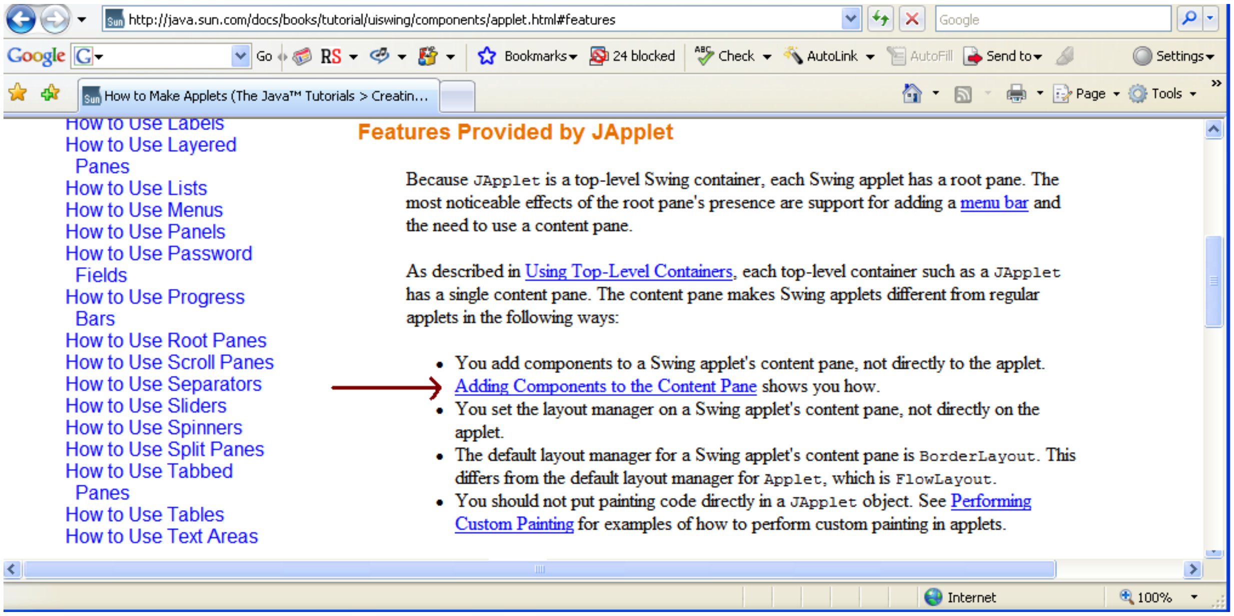
Note: If you don't see the applet running above, you need to install release 6 of the JDK. You can find the main source code for this applet in `TumbleItem.java`.

This section discusses the following topics:

- Features Provided by JApplet
- Threads in Applets
- Using Images in a Swing Applet
- Embedding an Applet in an HTML Page
- The JApplet API
- Applet Examples

Because so much of the Swing material uses the AWT infrastructure and event model, the tutorial first points to a tutorial for Getting Started with Applets.

Back on the How to Make Applets page, we have the link, Features Provided by JApplet, which shows us how to add components to the Content Pane. Go ahead and read this whole page to become familiar with the tools and concepts there.



Features Provided by JApplet

Because `JApplet` is a top-level Swing container, each Swing applet has a root pane. The most noticeable effects of the root pane's presence are support for adding a menu bar and the need to use a content pane.

As described in Using Top-Level Containers, each top-level container such as a `JApplet` has a single content pane. The content pane makes Swing applets different from regular applets in the following ways:

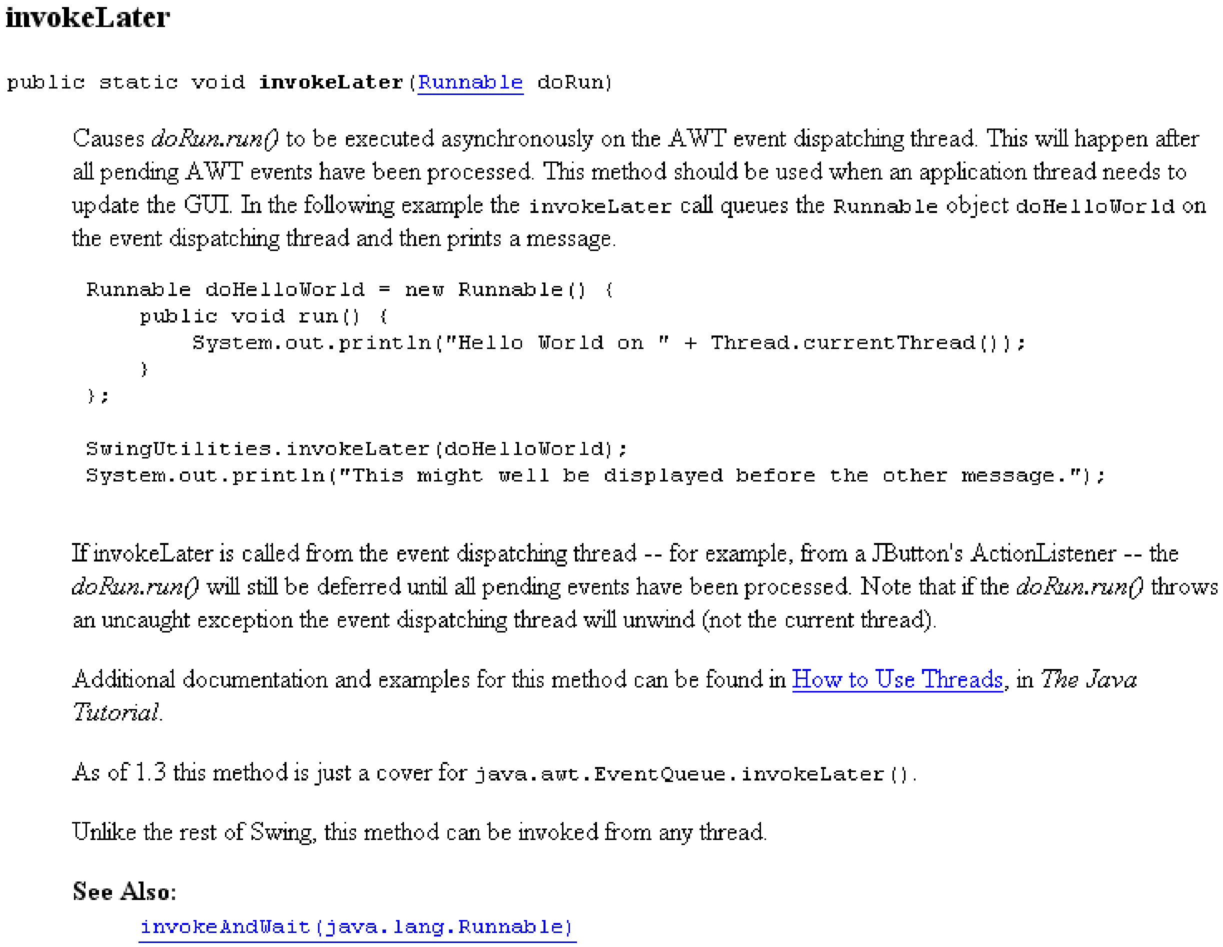
- You add components to a Swing applet's content pane, not directly to the applet. Adding Components to the Content Pane shows you how.
- You set the layout manager on a Swing applet's content pane, not directly on the applet.
- The default layout manager for a Swing applet's content pane is `BorderLayout`. This differs from the default layout manager for `Applet`, which is `FlowLayout`.
- You should not put painting code directly in a `JApplet` object. See Performing Custom Painting for examples of how to perform custom painting in applets.

The **ContentPane** is a **Container** that works similarly to the way double-buffering does when we paint on the **Graphics** area in applets. In the same way that the **setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE)** handles the Window listener for you, the **ContentPane** will take care of double-buffering for you and help graphics run more smoothly.

Applets aren't much different from applications. The main difference between them is in the ways they are started and in the way you produce a GUI for an application. In order to have a GUI for an application, you need to instantiate the Frame (or JFrame).

Let's compare the two. Go back to the How to Make Applets page, then to the section Threads in Applets. It has an example of an **init()** method that looks a lot like the **main()** method of their **JFrame** application.

API Now go to **javax.swing.SwingUtilities** and read over the full description of the **invokeLater(Runnable doRun)** method.



invokeLater

```
public static void invokeLater(Runnable doRun)
```

Causes *doRun.run()* to be executed asynchronously on the AWT event dispatching thread. This will happen after all pending AWT events have been processed. This method should be used when an application thread needs to update the GUI. In the following example the `invokeLater` call queues the `Runnable` object `doHelloWorld` on the event dispatching thread and then prints a message.

```
Runnable doHelloWorld = new Runnable() {
    public void run() {
        System.out.println("Hello World on " + Thread.currentThread());
    }
};

SwingUtilities.invokeLater(doHelloWorld);
System.out.println("This might well be displayed before the other message.");
```

If invokeLater is called from the event dispatching thread -- for example, from a JButton's ActionListener -- the *doRun.run()* will still be deferred until all pending events have been processed. Note that if the *doRun.run()* throws an uncaught exception the event dispatching thread will unwind (not the current thread).

Additional documentation and examples for this method can be found in How to Use Threads, in *The Java Tutorial*.

As of 1.3 this method is just a cover for `java.awt.EventQueue.invokeLater()`.

Unlike the rest of Swing, this method can be invoked from any thread.

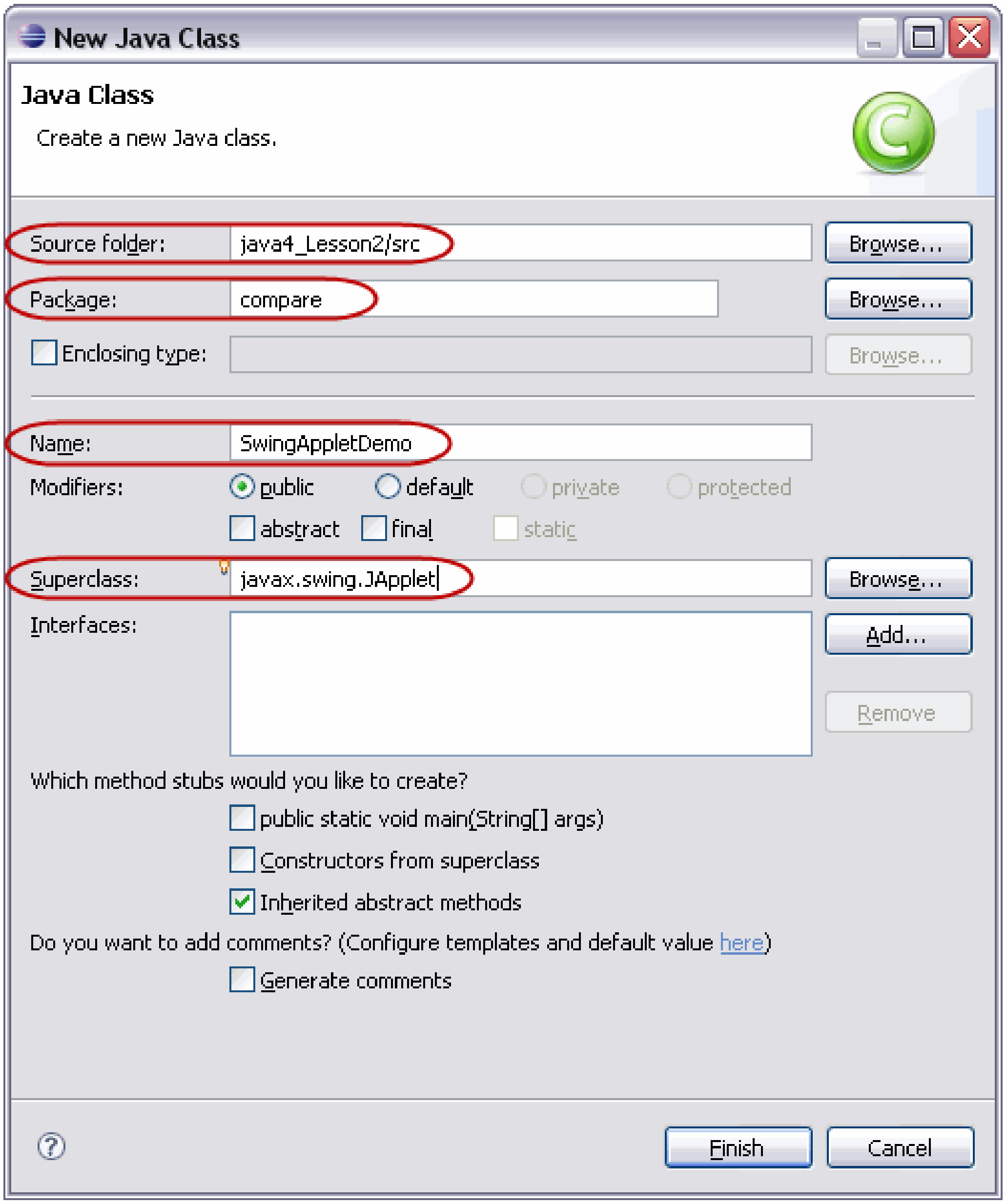
See Also:

`invokeAndWait(java.lang.Runnable)`

More Information on JApplets

We'll demonstrate JApplets using the example methods **init()** and **createGUI()**) from the Swing tutorial How to Make Applets.

In the java4_Lesson2 project, add a **SwingAppletDemo** class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson2/src

Package: compare

□ Enclosing type:

Name: SwingAppletDemo

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: javax.swing.JApplet

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

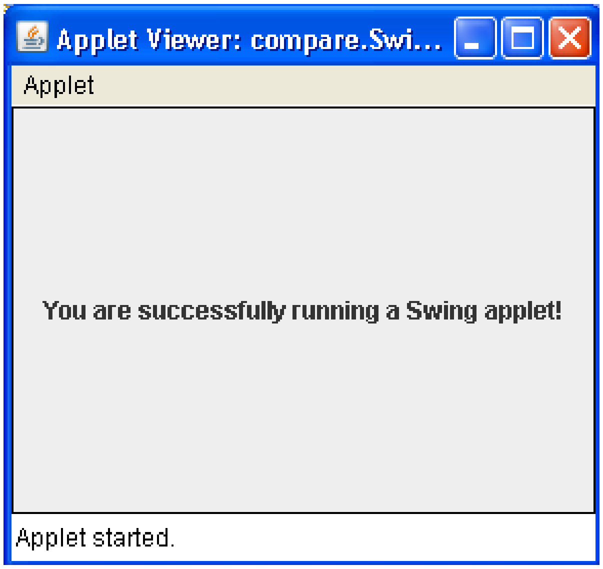
□ Generate comments

Type **SwingAppletDemo** as shown in **blue** below:

CODE TO TYPE: SwingAppletDemo

```
package compare;

import javax.swing.*; // Change from javax.swing.JApplet
import java.awt.*;

public class SwingAppletDemo extends JApplet {
    public void init() {
        //Execute a job on the event-dispatching thread:
        //creating this applet's GUI.
        try {
            javax.swing.SwingUtilities.invokeAndWait(new Runnable() {
                public void run() {
                    createGUI();
                }
            });
        } catch (Exception e) {
            System.err.println("createGUI didn't finish successfully");
        }
    }

    private void createGUI() {
        JLabel label = new JLabel("You are successfully running a Swing applet!"
);
        label.setHorizontalAlignment(JLabel.CENTER);
        label.setBorder(BorderFactory.createMatteBorder(1,1,1,1,Color.black));
        getContentPane().add(label, BorderLayout.CENTER);
    }
}
```

Save and run it (you might need to resize the Applet window).



You are successfully running a Swing applet!

Applet started.

Our example JApplet code shows two methods: **init()** and **createGUI()**. These methods are similar to the application's **main()** and **createAndShowGUI()** methods; starting JApplets is very similar to starting applications. But the JApplet's **init()** method with its **javax.swing.SwingUtilities.invokeAndWait()** call, is different from the application's **javax.swing.SwingUtilities.invokeLater()** call.

The invokeLater method is not appropriate for some JApplets because it could allow **init()** to return before initialization is complete. This could cause applet problems that are difficult to debug (such as constructors that mistakenly have a return type).

Take a look at the class LabelDemo.java (from How to Use Labels), which extends JPanel. **main()** invokes **createAndShowGUI()**, which instantiates a JFrame, then **adds** a **LabelDemo** (which is a JPanel), then gives the frame the ContentPane of this JPanel. LabelDemo's constructor adds all of the components to the JPanel.

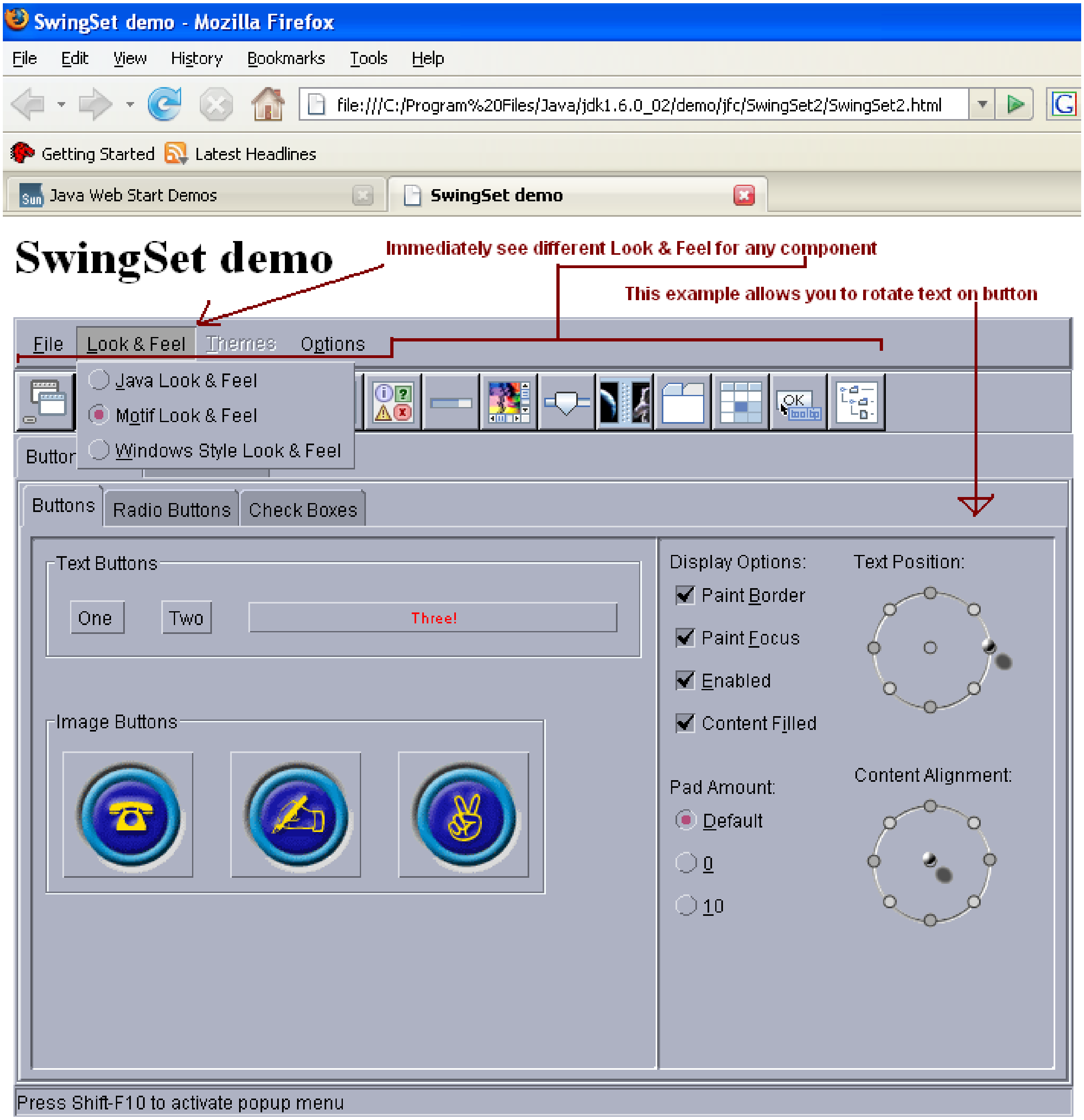
Here's an example that's a bit more complex: IconDemoApp (from Icon Demo), and an application. While we're at it, here's the Table of Examples for a tutorial on Swing Components.

Even More Swing

Explore on your own. Have fun--Swing has some awesome looks and capabilities! Here are a few more resources for you:

- The Oracle Swing tutorial includes Creating a GUI with JFC/Swing.
- The Swing Second Edition Book has a link to free version of the first edition.
- O'Reilly Media has published several books on Swing including Java Swing, Second Edition, by Marc Loy, Robert Eckstein, Dave Wood, James Elliott, and Brian Cole.
- Oracle provides Java Look and Feel Design Guidelines.
- Our perpetual source of Java knowledge, the API includes documentation on the javax.swing package and sub-packages.
- Oracle has a list of Training and Tutorials: Graphical User Interfaces and Printing.

Finally, check out the Swing Set Demo. You can test it from this web page or, if you downloaded Java and the demos on to your own machine, you have Swing Set Demo in the java directory. Play around with this demo, all it takes is a few mouse clicks!



Immediately see different Look & Feel for any component

This example allows you to rotate text on button

file:///C:/Program%20Files/Java/jdk1.6.0_02/demo/jfc/SwingSet2/SwingSet2.html

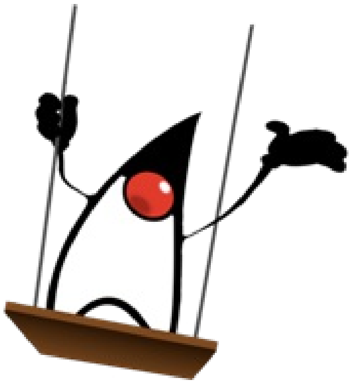
○ Java Look & Feel
● Motif Look & Feel
○ Windows Style Look & Feel

Display Options:
☑ Paint Border
☑ Paint Focus
☑ Enabled
☑ Content Filled

Pad Amount:
● Default
○ 0
○ 10

Press Shift-F10 to activate popup menu

We still have a lot more Swing coming up in the next lesson. See you there!

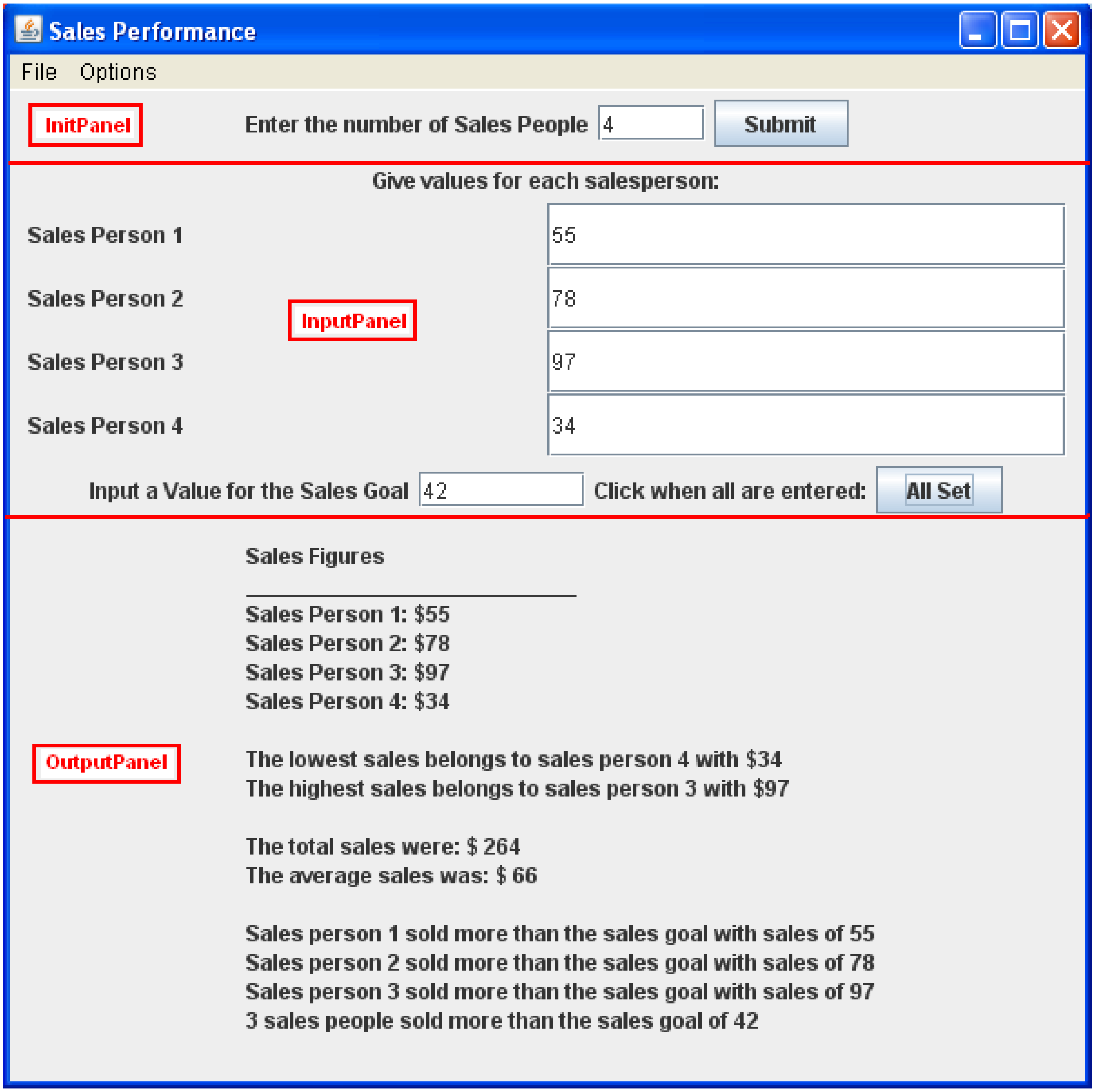


Graphical User Interfaces

Views

In this lesson, we'll create a user interface and learn about the **Layout Manager**. Most of the constructs in this code were demonstrated in the previous Java course series, so you'll probably recognize them. But using the **Panel** and the various LayoutManagers for GUI Frame's **Panel**s will be new. To learn more about layout managers, check out the java.awt.LayoutManager in the API. If you want to dig deeper still into layout managers, visit the Visual Guide to Layout Managers Tutorial.

When it runs to completion, our application will have three separate JPanels (InitPanel, InputPanel, and OutputPanel); one for each stage of the run. In the picture below, they are separated by **red** lines so you can differentiate between them:



Enter the number of Sales People: 4

Give values for each salesperson:

Sales Person 1: 55
Sales Person 2: 78
Sales Person 3: 97
Sales Person 4: 34

Input a Value for the Sales Goal: 42
Click when all are entered:

```
Sales Figures
_______________________
Sales Person 1: $55
Sales Person 2: $78
Sales Person 3: $97
Sales Person 4: $34

The lowest sales belongs to sales person 4 with $34
The highest sales belongs to sales person 3 with $97

The total sales were: $ 264
The average sales was: $ 66

Sales person 1 sold more than the sales goal with sales of 55
Sales person 2 sold more than the sales goal with sales of 78
Sales person 3 sold more than the sales goal with sales of 97
3 sales people sold more than the sales goal of 42
```

JFrames and JPanels

A **JFrame** (similar to a **Frame** in AWT) is a **Window** for the user. We know that JFrame is a Swing Component, because it is preceded by **J**. All Swing component names are preceded by a **J** in order to avoid confusing them with AWT Components. We can add menus and **Panel**s (JMenus and JPanels) to suit our needs. In fact, both **JFrame** and **JPanel** inherit from **Container**, so we can **add** other **Component**s to both of them.

We'll provide the user with a top-level **JFrame** and add various **JPanel**s to it.

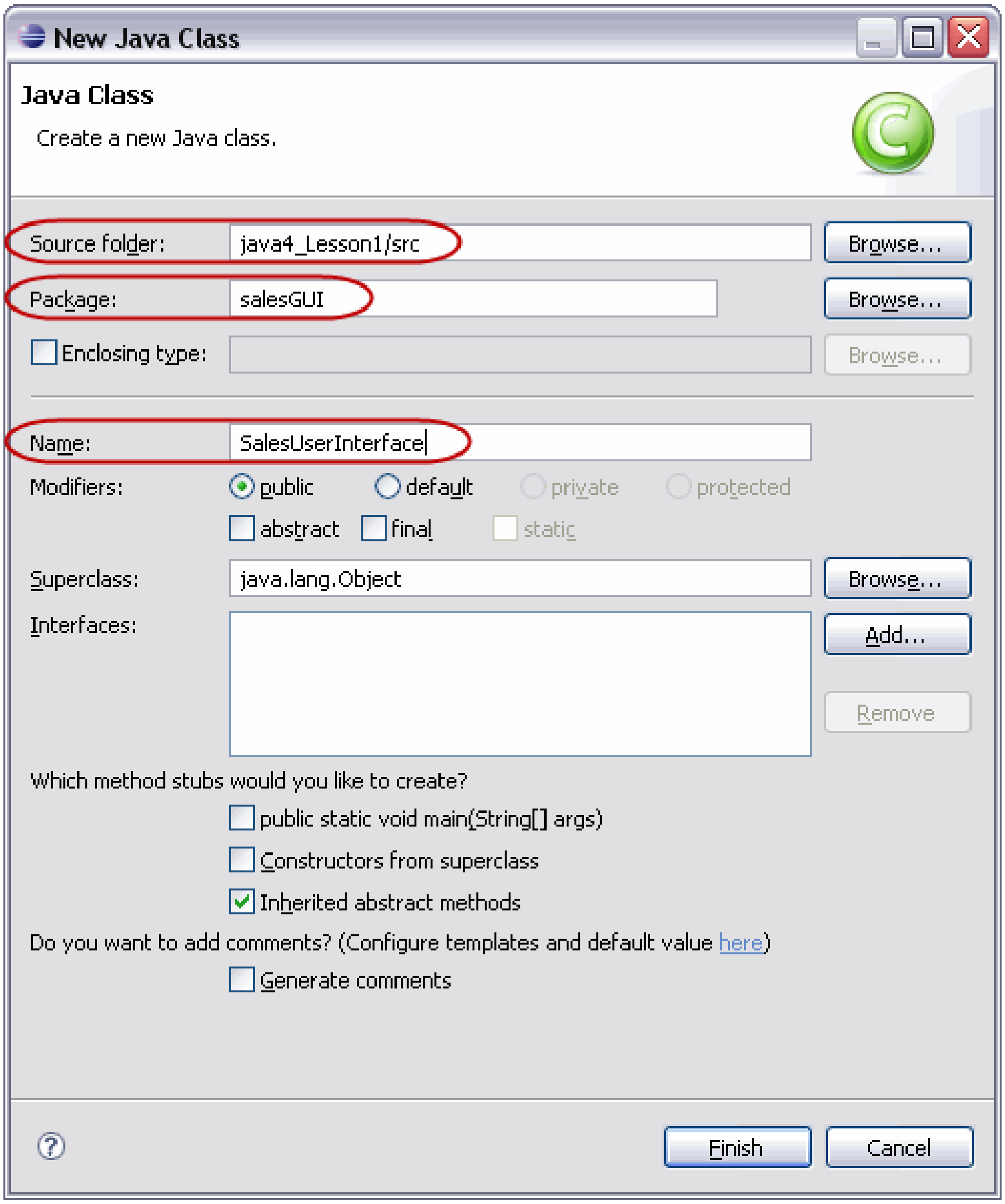
JFrames: The Top-Level View

Let's create the **JFrame** (Window) that will hold the JPanels and other components. Then we'll add a Menu Bar (JMenuBar) and a Menu item "File" (JMenuItem) with the menu option "Exit." Then we'll capture the click event, so we can close the window when we test it. If we didn't do this, we'd have to use the Console to end the program.

Swing components are referred to as "light weight" (as opposed to AWT components which are "heavy weight"), meaning that the components use the operating system to create components like Checkboxes and Choices. Swing components are created and drawn by the Swing library rather than relying on the operating system to draw them. This gives Java applications and applets a uniform look and feel across multiple operating systems. And when using Swing, the look and feel of your applications can be changed by altering a few lines of code.

Okay, time to get busy!

In the java4_Lesson1 project, create a new **SalesUserInterface** class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson1/src

Package: salesGUI

□ Enclosing type:

Name: SalesUserInterface

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Go to the **SalesUserInterface** editor window and edit it as shown in **blue**:

CODE TO TYPE: SalesUserInterface

```
package salesGUI;

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

public class SalesUserInterface extends JFrame{
    SalesApp app;
    JMenuBar mb;
    JMenu m;
    JMenuItem q, r, s, t;

    public SalesUserInterface(SalesApp myApp) {
        app = myApp;
        app.setMyUserInterface(this);
        setLayout(new BorderLayout());
        setPreferredSize(new Dimension(600, 600));
        mb = new JMenuBar();
        setJMenuBar(mb);
        m = new JMenu("File");
        mb.add(m);
        m.add(q = new JMenuItem("Exit"));
        q.addActionListener(new ActionListener(){
             public void actionPerformed(ActionEvent e) {
                 System.exit(0);
             }
        });

        pack();
        setVisible(true);
    }
}
```

Save it. It won't run, because we haven't created a **Main** class yet. We'll do that, but first let's take a closer look at the code we do have:

OBSERVE: SalesUserInterface

```
package salesGUI;

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

public class SalesUserInterface >extends JFrame>{
    SalesApp app;
    JMenuBar mb;
    JMenu m;
    JMenuItem q, r, s, t;

    public SalesUserInterface(SalesAPP myApp) {
        app = myApp;
        app.setMyUserInterface(this);
        setLayoutManager(new BorderLayout());
        setPreferredSize(new Dimension(600, 600));
        mb = new JMenuBar();
        setJMenuBar(mb);
        m = new JMenu("File");
        mb.add(m);
        m.add(q = new JMenuItem("Exit"));
        q.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });

        pack();
        setVisible(true);
    }
}
```

So let's go over our code piece by piece. We imported **javax.swing.***, **java.awt.event**, the **java.BorderLayout**, and **java.awt.Dimension**. The **JFrame** container is extended from Swing. Our SalesUserInterface class extends **JFrame** from Swing. We imported **java.awt.event** to allow us to capture events. We imported **java.BorderLayout** as our chosen Layout Manager. Finally, we imported **java.awt.Dimension** in order to size the window in this instance.

Next, we set up the variables for this application to use. The **app** variable is a reference to the **SalesApp** class that we created earlier. The other variables are of type JMenuBar, JMenu, and JMenuOption, all of which are Swing components we'll use for our menu bar. Again, the "J" is used here to differentiate Swing components from AWT components.

The constructor for this class accepts a **SalesAPP** object as a parameter. This enables the SalesApp object to make computations from this GUI.

We call **app.setMyUserInterface(this)**, which passes the SalesUserInterface object to the SaleApp instance. (This may be a bit confusing right now, because we haven't used this handle yet. Don't worry, we'll get there. Patience grasshopper.)

Next, we call the JFrame method **setLayoutManager(new BorderLayout())**, in which we create a new **BorderLayout()**--we'll use the BorderLayout manager. Well, we aren't actually using it just yet, but it will be used to lay out the JPanels when we add them. For now, let's get the window up with the File menu and Exit option.

In the **dark red** code above, we instantiate a **JMenuBar** called **mb**, and then set the menu bar on the SalesUserInterface JFrame with **setJMenuBar(mb);**. Then we add the JMenu "File" to the menu bar, and add the "Exit" JMenuItem to that.

We add an **ActionLister** to the Exit Menu Item to catch the click event. To do that, we use the anonymous inner class technique. Then we call **System.exit(0);** in the implemented interface method ActionPerformed() to kill the Application process.

Finally, we call **pack();** and **setVisible(true);** to show the GUI. The method **pack()** is actually inherited from Window, and causes the window to be set to its preferred size. **setVisible()** makes the window visible on the screen. These two methods should always be called when using a JFrame.

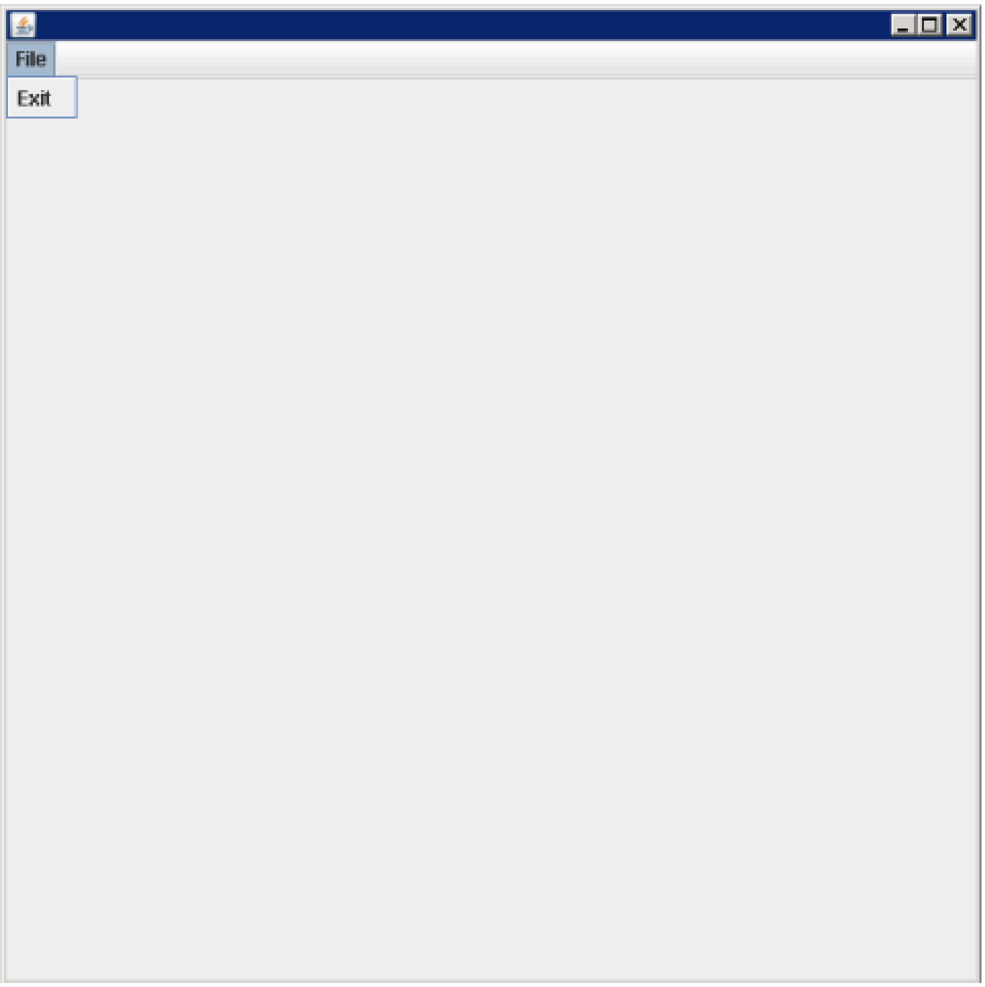
Okay, let's make a Main Class and get this application running!

Start a new Main Class file in the same location as your **SalesUserInterface** class. Type the **blue** code as shown:

CODE TO TYPE: Main

```
package salesGUI;

public class Main {
    public static void main(String[] args) {

        SalesApp newApp = new SalesApp();
        SalesUserInterface appFrame = new SalesUserInterface(newApp);
    }
}
```

Save and run it. Select **File | Exit**:



OBSERVE: Breaking Down the Main

```
package salesGUI;

public class Main {
    public static void main(String[] args) {

        SalesAPP newApp = new SalesApp();
        SalesUserInterface appFrame = new SalesUserInterface(newApp);
    }
}
```

Here we instantiate a **SalesApp** object we call **newApp**. Then we instantiate a SalesUserInterface object and pass the **newApp** object to it as a parameter. In the SalesUserInterface object, we get that **newApp** object and it becomes **app**, which will ultimately make calculations for us.

Now, add the first JPanel: **InitPanel**. We'll add this class as an Inner Class to SalesUserInterface. We do that for two reasons:

- This class is specific to the SalesUserInterface and we don't plan to reuse it.
- It will make it easier to access aspects of the SalesUserInterface later in this lesson.

Add the **blue** code to SalesUserInterface:

CODE TO EDIT: Inner Class InitPanel in SalesUserInterface

```
package salesGUI;

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

public class SalesUserInterface extends JFrame{
    SalesApp app;
    JMenuBar mb;
    JMenu m;
    JMenuItem q, r, s, t;
    JLabel peopleLabel;
    JTextField peopleField;
    JButton jbNumPeople, done;

    public SalesUserInterface(SalesAPP myApp) {
        app = myApp;
        app.setMyUserInterface(this);
        setLayout(new BorderLayout());
        setPreferredSize(new Dimension(600, 600));
        mb = new JMenuBar();
        setJMenuBar(mb);
        m = new JMenu("File");
        mb.add(m);
        m.add(q = new JMenuItem("Exit"));
        q.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });
        InitPanel specifyNumber = new InitPanel();
        add("North", specifyNumber);
        pack();
        setVisible(true);
    }

    private class InitPanel extends JPanel{
        public InitPanel() {
            peopleLabel = new JLabel("Enter the number of sales people");
            add(peopleLabel);
            peopleField = new JTextField(5);
            add(peopleField);
            jbNumPeople = new JButton("Submit");
            add(jbNumPeople);
        }
    }
}
```

Save this SalesUserInterface and run the **Main.java**.

You'll see the "Enter the Number of Sales People" JLabel, an input JTextfield, and a Submit JButton. Of course, it doesn't do anything just yet.

OBSERVE: InitPanel

```
package salesGUI;

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

public class SalesUserInterface extends JFrame {
    SalesApp app;
    JMenuBar mb;
    JMenu m;
    JMenuItem q, r, s, t;
        JLabel peopleLabel;
        JTextField peopleField;
        JButton jbNumPeople, done;

    public SalesUserInterface(SalesAPP myApp) {
        app = myApp;
        app.setMyUserInterface(this);
        setLayout(new BorderLayout());
        setPreferredSize(new Dimension(600, 600));
        mb = new MenuBar();
        setMenuBar(mb);
        m = new Menu("File");
        mb.add(m);
        m.add(q = new JMenuItem("Exit"));
        q.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });
        InitPanel specifyNumber = new InitPanel();
        add("North", specifyNumber);
        pack();
        setVisible(true);

    }

    public class InitPanel extends JPanel{
        public InitPanel() {
            peopleLabel = new JLabel("Enter the number of sales people");
            add(peopleLabel);
            peopleField = new JTextField(5);
            add(peopleField);
            jbNumPeople = new JButton("Submit");
            add(jbNumPeople);
        }
    }
}
```

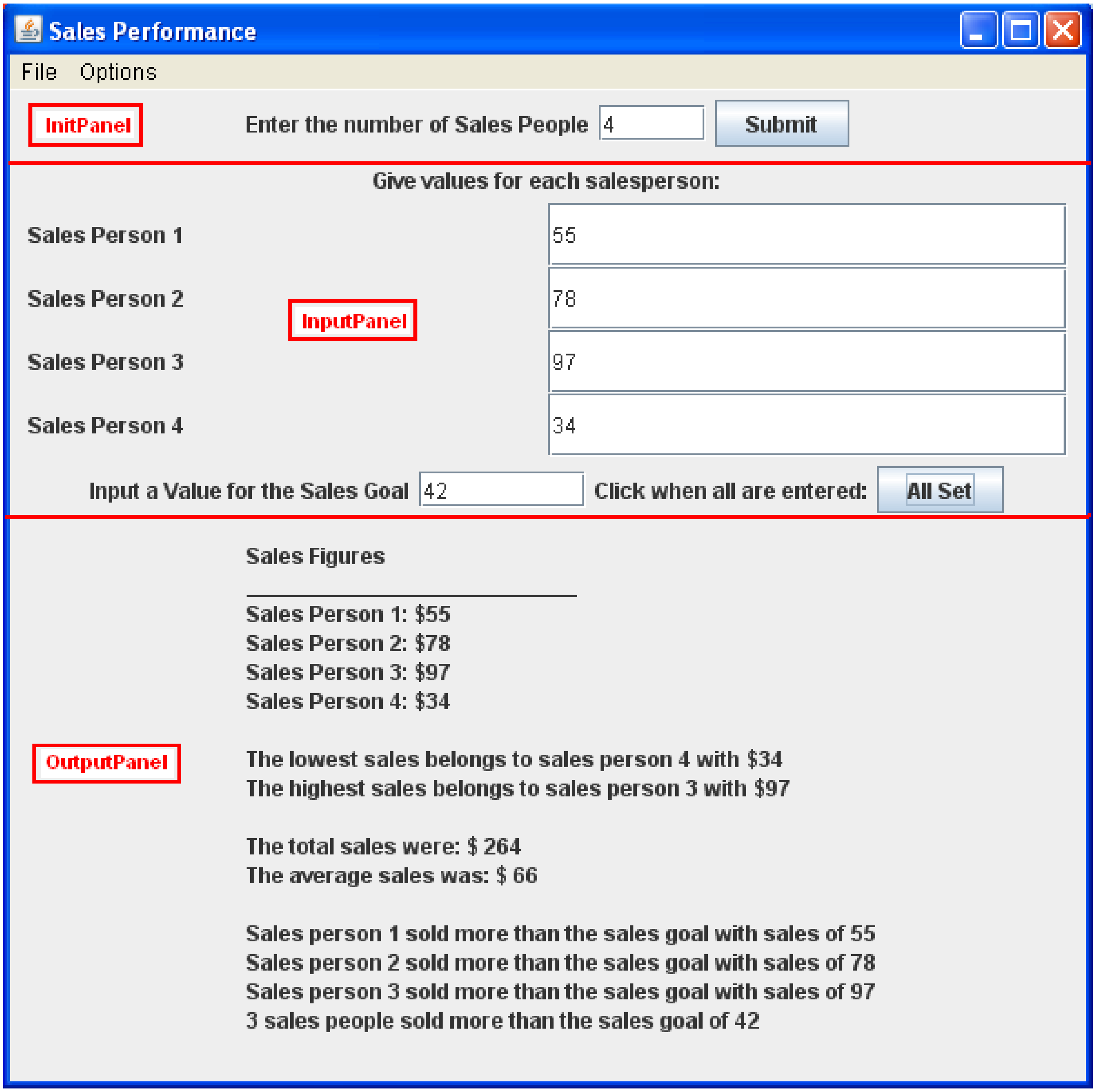
We created a new class called **InitPanel**, which **extends JPanel**. We added a *JLabel, JTextfield,* and *JButton*. We used the **add()** method from JPanel to add each of those instantiated objects to our JPanel. Then we added the variables used in InitPanel to the global scope of SalesUserInterface. You'll see why we did that later in this lesson.

OBSERVE

```
InitPanel specifyNumber = new InitPanel();
add("North", specifyNumber);
```

Just for practice, try changing "North" to "East" or "South" and running the program again to observe the effect it has.

We want our GUI to look like this:



Enter the number of Sales People 4

Give values for each salesperson:

Sales Person 1 55

Sales Person 2 78

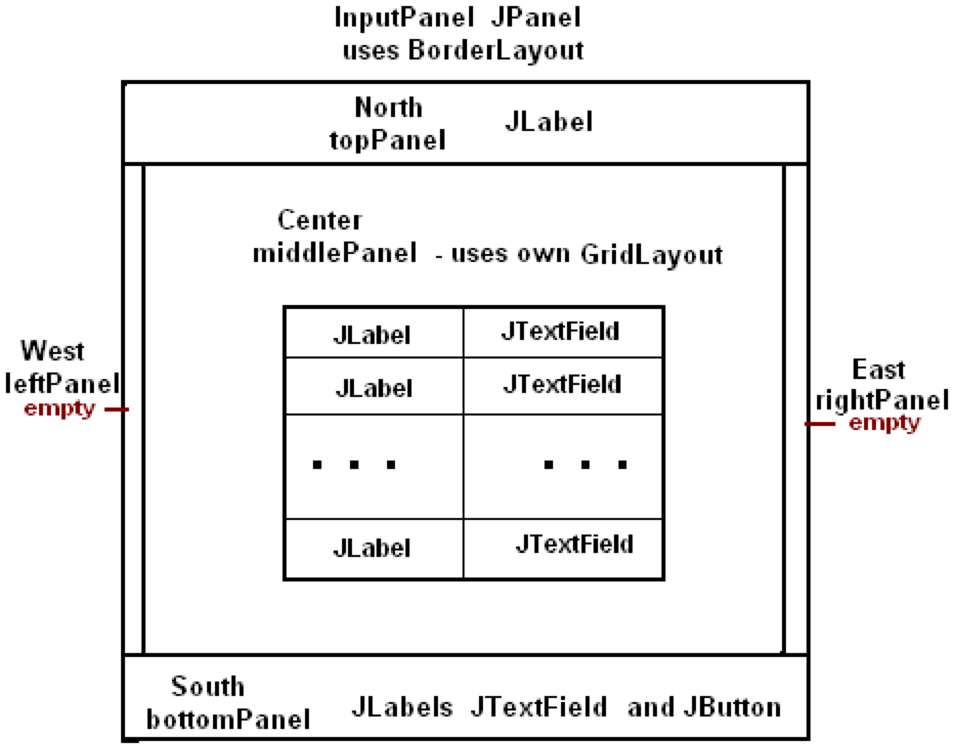
Sales Person 3 97

Sales Person 4 34

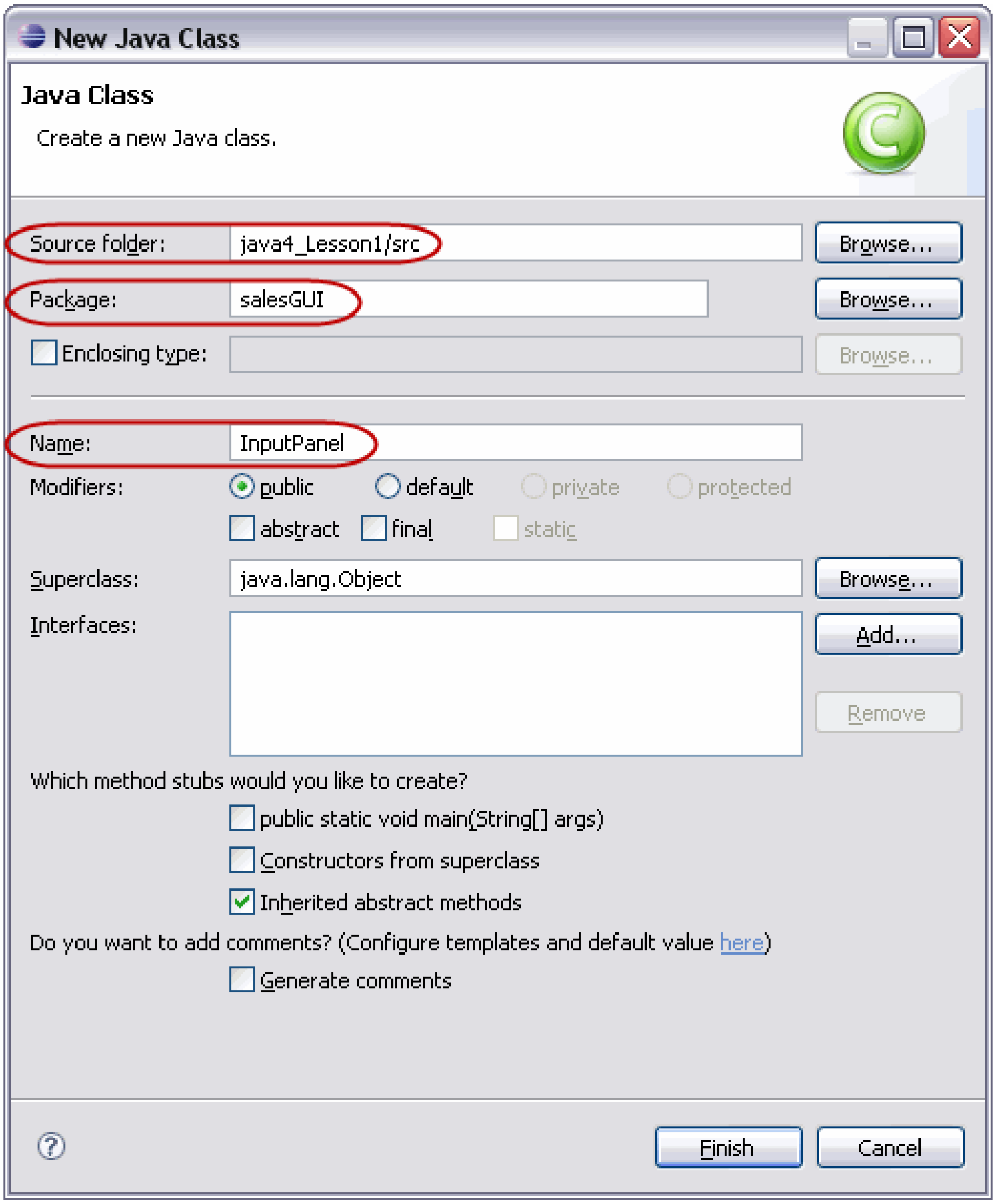
Input a Value for the Sales Goal 42 Click when all are entered:

```
Sales Figures
_______________________
Sales Person 1: $55
Sales Person 2: $78
Sales Person 3: $97
Sales Person 4: $34

The lowest sales belongs to sales person 4 with $34
The highest sales belongs to sales person 3 with $97

The total sales were: $ 264
The average sales was: $ 66

Sales person 1 sold more than the sales goal with sales of 55
Sales person 2 sold more than the sales goal with sales of 78
Sales person 3 sold more than the sales goal with sales of 97
3 sales people sold more than the sales goal of 42
```

So next, we need to create the input panel to add to our JFrame. We'll make a class called **InputPanel**, and implement it as inputPanel on our SalesUserInterface. Let's make it a separate class (we might want to reuse it someday). In this particular class, we'll extend JPanel and layer more JPanels onto it. Here's a graphical representation of what we'll be adding to our InputPanel:



In the java4_Lesson1 project, create an **InputPanel** class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson1/src

Package: salesGUI

□ Enclosing type:

Name: InputPanel

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **InputPanel** as shown in **blue**:

CODE TO TYPE: InputPanel

```
package salesGUI;

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

public class InputPanel extends JPanel {
    JPanel topPanel;
    SalesApp app;
    JLabel prompt;

    public InputPanel(SalesApp container) {
        this.app = container;
        this.setLayout(new BorderLayout());
        topPanel = new JPanel();
        topPanel.setLayout(new FlowLayout());
        add("North", topPanel);
        prompt = new JLabel("Give values for each salesperson:");
        topPanel.add(prompt);
    }
}
```

Save it. We can't view it on our SalesUserInterface yet, because we haven't added this InputPanel to our SalesUserInterface JFrame. Let's do that now!

In your SalesUserInterface.java file, add the code shown in **blue**:

CODE TO EDIT: SalesUserInterface

```
package salesGUI;

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

public class SalesUserInterface extends JFrame{
    SalesApp app;
    JMenuBar mb;
    JMenu m;
    JMenuItem q, r, s, t;
    JLabel peopleLabel;
    JTextField peopleField;
    JButton jbNumPeople, done;

    public SalesUserInterface(SalesAPP myApp) {
        app = myApp;
        app.setMyUserInterface(this);
        setLayout(new BorderLayout());
        setPreferredSize(new Dimension(600, 600));
        mb = new JMenuBar();
        setJMenuBar(mb);
        m = new JMenu("File");
        mb.add(m);
        m.add(q = new JMenuItem("Exit"));
        q.addActionListener(new ActionListener(){
             public void actionPerformed(ActionEvent e) {
                 System.exit(0);
             }
        });
        InitPanel specifyNumber = new InitPanel();
        add("North", specifyNumber);
        InputPanel inputPanel = new InputPanel(app);
        add("Center",inputPanel);
        pack();
        setVisible(true);
    }

    public class InitPanel extends JPanel {
        public InitPanel() {
            peopleLabel = new JLabel("Enter the number of sales people");
            add(peopleLabel);
            peopleField = new JTextField(5);
            add(peopleField);
            jbNumPeople = new JButton("Submit");
            add(jbNumPeople);
        }
    }
}
```

Save this SalesUserInterface and run the **Main**. You'll see the prompt **Give values for each salesperson**. That's our **topPanel**!

So far, we've added the topPanel, a JPanel, into InputPanel. We still need to make the middle panel to do all of the work in InputPanel. We'll add some code to prepare our InputPanel to accept the number of sales people entered from the InitPanel as well.

CODE TO EDIT: InputPanel

```
package salesGUI;

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

public class InputPanel extends JPanel {

    JPanel topPanel, middlePanel, bottomPanel, leftPanel, rightPanel;
    SalesApp app;
    JLabel prompt, doneLabel, jlSalesBar;
    JLabel[] jlSales;
    JButton done;
    JTextField[] jtfSales;
    JTextField jtfSalesBar;
    int numPeople;

    public InputPanel(SalesApp container, int numPeople, int gridX) {
        this.app = container;
        this.numPeople = numPeople;
        this.setLayout(new BorderLayout());
        topPanel = new JPanel();
        topPanel.setLayout(new FlowLayout());
        middlePanel = new JPanel(new GridLayout(numPeople, gridX));
        bottomPanel = new JPanel();
        bottomPanel.setLayout(new FlowLayout());
        leftPanel = new JPanel();
        rightPanel = new JPanel();
        add("North", topPanel);
        add("Center", middlePanel);
        add("South", bottomPanel);
        add("East", rightPanel);
        add("West", leftPanel);
        jlSales = new JLabel[numPeople];
        jtfSales = new JTextField[numPeople];
        prompt = new JLabel("Give values for each salesperson:");
        topPanel.add(prompt);

        for (int x = 0; x < numPeople; x++)
        {
            jlSales[x] = new JLabel("Sales Person " + (x+1));
            jtfSales[x] = new JTextField("0",8);
            middlePanel.add(jlSales[x]);
            middlePanel.add(jtfSales[x]);
        }
        jlSalesBar = new JLabel("Enter a value for the sales goal");
        bottomPanel.add(jlSalesBar);
        jtfSalesBar = new JTextField("0",8);
        bottomPanel.add(jtfSalesBar);
        doneLabel = new JLabel("Click when all are entered:");
        bottomPanel.add(doneLabel);
        done = new JButton("All Set");
        bottomPanel.add(done);
    }
}
```

Save it. We've added lots of new code here. Let's just get it working so we can actually see it in action first, then we'll go over it in detail.

In order for the button in the InitPanel to work, we'll create a new private inner class in SalesUserInterface.java that will serve as the button's listener. Let's call it **NumPeopleSalesListener**.

In SalesUserInterface.java, add the code shown in **blue**:

CODE TO EDIT: SalesUserInterface

```
package salesGUI;

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

public class SalesUserInterface extends JFrame{
    SalesApp app;
    JMenuBar mb;
    JMenu m, m1;
    JMenuItem q,r,s,t;
    InputPanel inputPanel;
    JLabel peopleLabel;
    JTextField peopleField;
    JButton jbNumPeople, done;
    int numPeople;
    boolean processed = false;

    public SalesUserInterface(SalesAPP myApp) {
        app = myApp;
        app.setMyUserInterface(this);
        setLayout(new BorderLayout());
        setPreferredSize(new Dimension(600, 600));
        mb = new JMenuBar();
        setJMenuBar(mb);
        m = new JMenu("File");
        mb.add(m);
        m.add(q = new JMenuItem("Exit"));
        q.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });

        InitPanel specifyNumber = new InitPanel();
        add("North", specifyNumber);
        // REMOVE the next two lines.
        InputPanel inputPanel = new InputPanel(app);
        add("Center",inputPanel);
        pack();
        setVisible(true);
    }

    private class InitPanel extends JPanel {
        public InitPanel() {
            peopleLabel = new JLabel("Enter the number of sales people");
            add(peopleLabel);
            peopleField = new JTextField(5);
            add(peopleField);
            jbNumPeople = new JButton("Submit");
            add(jbNumPeople);
            jbNumPeople.addActionListener(new NumSalesPeopleListener());
        }
    }

    private class NumSalesPeopleListener implements ActionListener {
        public void actionPerformed(ActionEvent event){
            if (inputPanel != null)
            {
                remove(inputPanel);
                app = new SalesApp();
            }
            numPeople = Integer.parseInt(peopleField.getText());
            inputPanel = new InputPanel(app, numPeople, 2);
            add("Center", inputPanel);
            SalesUserInterface.this.validate();
        }
    }
}
```

Save it and run Main.java. When your application appears, type in a number and press **Submit**. You'll see the input fields for your salespeople.

Now let's go over this, bit by bit. First we'll look over the code we added to **InputPanel**:

OBSERVE: InputPanel

```
package salesGUI;

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

public class InputPanel extends JPanel {
    JPanel topPanel, middlePanel, bottomPanel, leftPanel, rightPanel;
    SalesAPP app;
    JLabel prompt, doneLabel, jlSalesBar;
    JLabel[] jlSales;
    JButton done;
    JTextField[] jtfSales;
    JTextField jtfSalesBar;
    int numPeople;

    public InputPanel(SalesApp container,int numPeople , int gridX) {
        this.app = container;
        this.numPeople = numPeople;
        this.setLayout(new BorderLayout());
        topPanel = new JPanel();
        topPanel.setLayout(new FlowLayout());
        middlePanel = new JPanel(new GridLayout(numPeople, gridX));

        bottomPanel = new JPanel();
        bottomPanel.setLayout(new FlowLayout());
        leftPanel = new JPanel();
        rightPanel = new JPanel();
        add("North", topPanel);
        add("Center", middlePanel);
        add("South", bottomPanel);
        add("East", rightPanel);
        add("West", leftPanel);
        jlSales = new JLabel[numPeople];
        jtfSales = new JTextField[numPeople];

        add("North", topPanel);
        prompt = new JLabel("Give values for each salesperson:");
        topPanel.add(prompt);

        for (int x = 0; x < numPeople; x++)
        {
            jlSales[x] = new JLabel("Sales Person " + (x+1));
            jtfSales[x] = new JTextField("0",8);
            middlePanel.add(jlSales[x]);
            middlePanel.add(jtfSales[x]);
        }
        jlSalesBar = new JLabel("Enter a value for the sales goal");
        bottomPanel.add(jlSalesBar);
        jtfSalesBar = new JTextField("0",8);
        bottomPanel.add(jtfSalesBar);
        doneLabel = new JLabel("Click when all are entered:");
        bottomPanel.add(doneLabel);
        done = new JButton("All Set");
        bottomPanel.add(done);
    }
}
```

The code in blue is already familiar, so we'll turn our attention to the new stuff. When we call this **InputPanel()** constructor in SalesUserInterface, we will supply the parameters **numPeople** and **gridX**. Those parameters are used in the **GridLayout(rows, cols)** constructor (go ahead and look up GridLayout in the API). **numPeople** is the number of **rows** that our grid will have and **gridX** is the total number of **columns** our grid will have.

The code in **red** is a **for** statement that is indexed by **x** up to **numPeople**. We create an array of both **JLabel** and **JTextField** components, then **add** them to the **middlePanel**. When we add components using **add**, the GridLayout layout manager automatically adds them from left-to-right and top-to-bottom, and allows them each the same amount of space. Nice.

Now, look at the code we added to **SalesUserInterface**:

OBSERVE: SalesUserInterface.java

```
package salesGUI;

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

public class SalesUserInterface extends JFrame{
    SalesApp app;
    JMenuBar mb;
    JMenu m, m1;
    JMenuItem q,r,s,t;
    InputPanel inputPanel;
    JLabel peopleLabel;
    JTextField peopleField;
    JButton jbNumPeople, done;
    int numPeople;
    boolean processed = false;

    public SalesUserInterface(SalesApp myApp) {
        app = myApp;
        app.setMyUserInterface(this);
        setLayout(new BorderLayout());
        setPreferredSize(new Dimension(600, 600));
        mb = new JMenuBar();
        setJMenuBar(mb);
        m = new JMenu("File");
        mb.add(m);
        m.add(q = new JMenuItem("Exit"));
        q.addActionListener(new ActionListener(){
             public void actionPerformed(ActionEvent e) {
                 System.exit(0);
             }
        });

        InitPanel specifyNumber = new InitPanel();
        add("North", specifyNumber);

        pack();
        setVisible(true);
    }

    private class InitPanel extends JPanel {
        public InitPanel() {
            peopleLabel = new JLabel("Enter the number of sales people");
            add(peopleLabel);
            peopleField = new JTextField(5);
            add(peopleField);
            jbNumPeople = new JButton("Submit");
            add(jbNumPeople);
            jbNumPeople.addActionListener(new NumSalesPeopleListener());
        }
    }

    private class NumSalesPeopleListener implements ActionListener {
        public void actionPerformed(ActionEvent event){
            if (inputPanel != null)
            {
                remove(inputPanel);
                app = new SalesAPP();
            }
            numPeople = Integer.parseInt(peopleField.getText());
            inputPanel = new InputPanel(app,numPeople, 2);
            add("Center", inputPanel);
            SalesUserInterface.this.validate();
            }
        }
}
```

We've added the actionlistener **NumSalesPeopleListener** to the JButton **jbNumPeople**. And we've added the **NumSalesPeopleListener** class that implements **ActionListener**. In the implemented actionPerformed() method, we've added an **if** statement that checks to see **if** an inputPanel exists already. If we change the number of sales people, then we rebuild that existing inputPanel. We **remove(inputPanel)** and create a new SalesApp to pass to the new InputPanel. We retrieve the **numPeople** using the **getText()** method in peopleField (a JTextField object). Now, when we create an instance of InputPanel, we pass it **numPeople** (the number of rows we want) and **2** (the number of columns we want).

Next, we add the inputPanel, position it in the **"Center"**, and then call **SalesUserInterface.this.validate()**. We call JFrame's **validate()** method when we rebuild and re-add the inputPanel component. In our code, we called **pack()** before we called **setVisible()**. Once a panel is visible on a JFrame, we call **validate()** to get our application to redraw. Because **NumSalesPeopleListener** is an inner class, we can use **SalesUserInterface.this** to access the JFrame's **validate()** method.

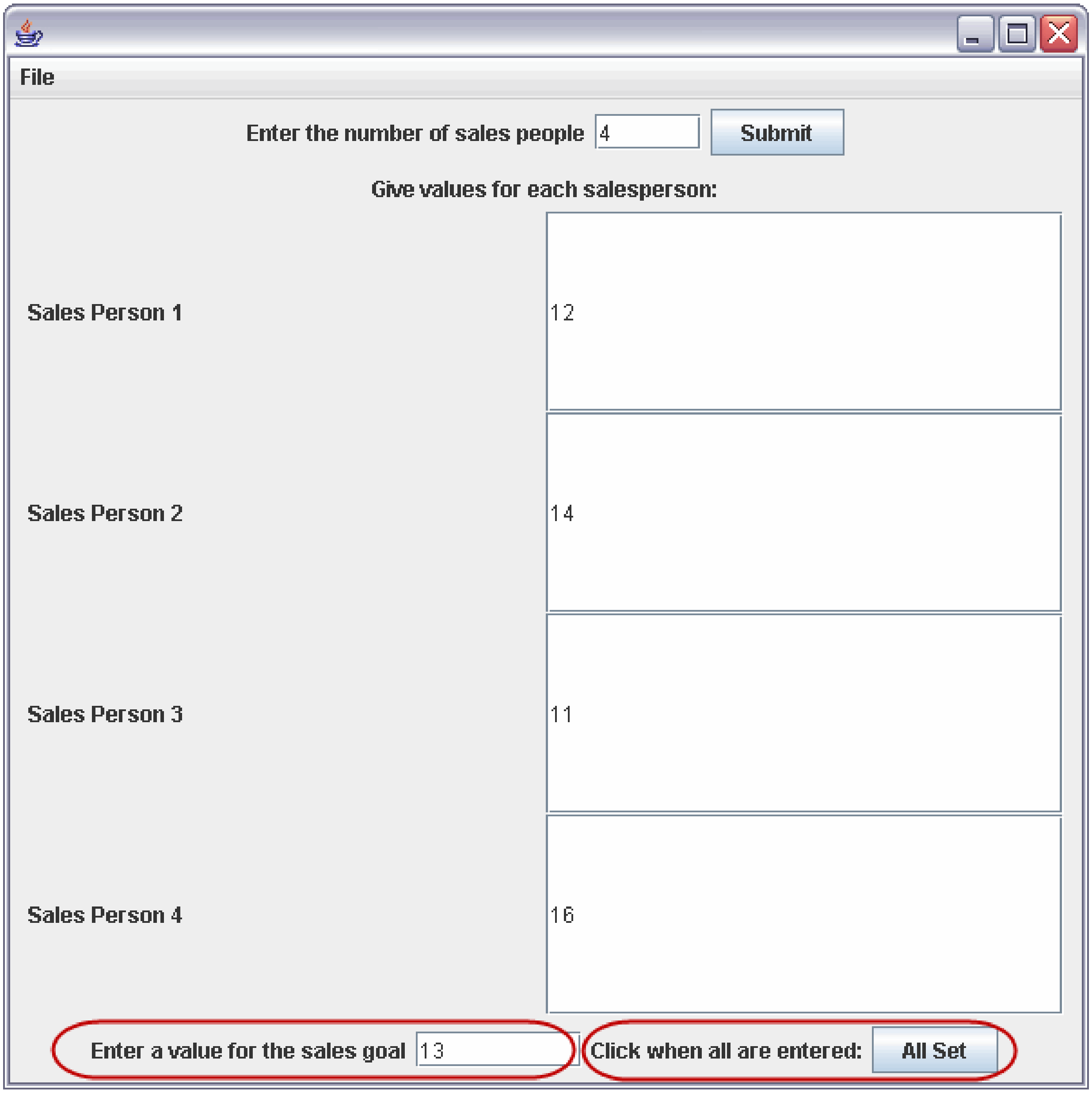
So far, so good. Now let's turn our InputPanel into a listener and make our **done** JButton grab the numbers the user enters and send them to our SalesApp.

Edit **InputPanel.java** as shown in **blue**:

CODE TO EDIT: InputPanel

```
package salesGUI;

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

public class InputPanel extends JPanel implements ActionListener {
    JPanel topPanel, middlePanel, bottomPanel, leftPanel, rightPanel;
    JLabel[] jlSales;
    JButton done;
    SalesApp app;
    JLabel prompt, doneLabel, jlSalesBar;
    JTextField[] jtfSales;
    JTextField jtfSalesBar;
    int numPeople;
    int [] sales;
    int goal;

    public InputPanel(SalesAPP container, int numPeople, int gridX) {
        this.app = container;
        this.numPeople = numPeople;
        sales = new int[numPeople];
        this.setLayout(new BorderLayout());
        topPanel = new JPanel();
        topPanel.setLayout(new FlowLayout());
        middlePanel = new JPanel(new GridLayout(numPeople, gridX));
        bottomPanel = new JPanel();
        bottomPanel.setLayout(new FlowLayout());
        leftPanel = new JPanel();
        rightPanel = new JPanel();
        add("North", topPanel);
        add("Center", middlePanel);
        add("South", bottomPanel);
        add("East", rightPanel);
        add("West", leftPanel);
        jlSales = new JLabel[numPeople];
        jtfSales = new JTextField[numPeople];
        prompt = new JLabel("Give values for each salesperson:");
        topPanel.add(prompt);

        for (int x = 0; x < numPeople; x++)
        {
            jlSales[x] = new JLabel("Sales Person " + (x+1));
            jtfSales[x] = new JTextField("0",8);
            middlePanel.add(jlSales[x]);
            middlePanel.add(jtfSales[x]);
        }
        jlSalesBar = new JLabel("Enter a value for the sales goal");
        bottomPanel.add(jlSalesBar);
        jtfSalesBar = new JTextField("0",8);
        bottomPanel.add(jtfSalesBar);
        doneLabel = new JLabel("Click when all are entered:");
        bottomPanel.add(doneLabel);
        done = new JButton("All Set");
        bottomPanel.add(done);
        done.addActionListener(this);
    }

    public void actionPerformed(ActionEvent event) {
        if(event.getSource() instanceof JButton)
        {
            if ((JButton)event.getSource() == done)
            {
                for (int x = 0; x < numPeople; x++)
                {
                    sales[x] = Integer.parseInt(jtfSales[x].getText());
                }
                app.setSales(sales);
                goal = Integer.parseInt(jtfSalesBar.getText());
                app.setSalesBar(goal);
            }
        }
     }
}
```

Save and run it (run the Main.java). Now type in a number (we used **4**) and when the input panel comes up, enter numbers into that as well; include the sales goal. Then click **All Set** as shown:



Enter the number of sales people 4

Give values for each salesperson:

Sales Person 1 12

Sales Person 2 14

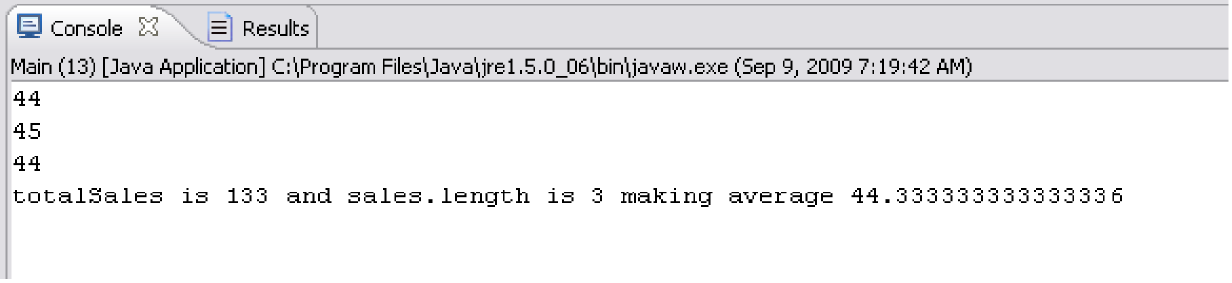
Sales Person 3 11

Sales Person 4 16

Enter a value for the sales goal 13

Click when all are entered:

The System.out.print output appears in the Console:



```
Main (13) [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Sep 9, 2009 7:19:42 AM)
44
45
44
totalSales is 133 and sales.length is 3 making average 44.333333333333336
```

OBSERVE: Changes to InputPanel

```
package salesGUI;

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

public class InputPanel extends JPanel implements ActionListener {
    Panel topPanel, middlePanel, bottomPanel, leftPanel, rightPanel;
    JLabel[] jlSales;
    JLabel  prompt, doneLabel, jlSalesBar;
    JTextField[] jtfSales;
    JTextField jtfSalesBar;
    JButton done;
    SalesApp app;
    int numPeople;
    int [] sales;
    int goal;

    public InputPanel(SalesApp container, int numPeople, int gridX){
        this.app = container;
        this.numPeople = numPeople;
        sales = new int[numPeople];
        this.setLayout(new BorderLayout());
        topPanel = new Panel();
        topPanel.setLayout(new FlowLayout());
        middlePanel = new Panel();
        middlePanel.setLayout(new GridLayout(numPeople, gridX));
        bottomPanel = new Panel();
        bottomPanel.setLayout(new FlowLayout());
        leftPanel = new Panel();
        rightPanel = new Panel();
        add("North", topPanel);
        add("Center", middlePanel);
        add("South", bottomPanel);
        add("East", rightPanel);
        add("West", leftPanel);

        jlSales = new JLabel[numPeople];
        jtfSales = new JTextField[numPeople];
        prompt = new JLabel("Give values for each salesperson:");
        topPanel.add(prompt);
        for (int x = 0; x < numPeople; x++)
        {
            jlSales[x] = new JLabel("Sales Person " + (x+1));
            jtfSales[x] = new JTextField("0",8);
            middlePanel.add(jlSales[x]);
            middlePanel.add(jtfSales[x]);
        }
        jlSalesBar = new JLabel("Enter a value for the sales goal");
        bottomPanel.add(jlSalesBar);
        jtfSalesBar = new JTextField("0",8);
        bottomPanel.add(jtfSalesBar);
        doneLabel = new JLabel("Click when all are entered:");
        bottomPanel.add(doneLabel);
        done = new JButton("All Set");
        bottomPanel.add(done);
        done.addActionListener(this);
    }

    public void actionPerformed(ActionEvent event){
        if (event.getSource() instanceof JButton)
        {
            if ((JButton)event.getSource() == done)
            {
                for (int x = 0; x < numPeople; x++)
                {
                    sales[x] = Integer.parseInt(jtfSales[x].getText());
                }
                app.setSales(sales);
                goal = Integer.parseInt(jtfSalesBar.getText());
                app.setSalesBar(goal);
            }
        }
    }
}
```

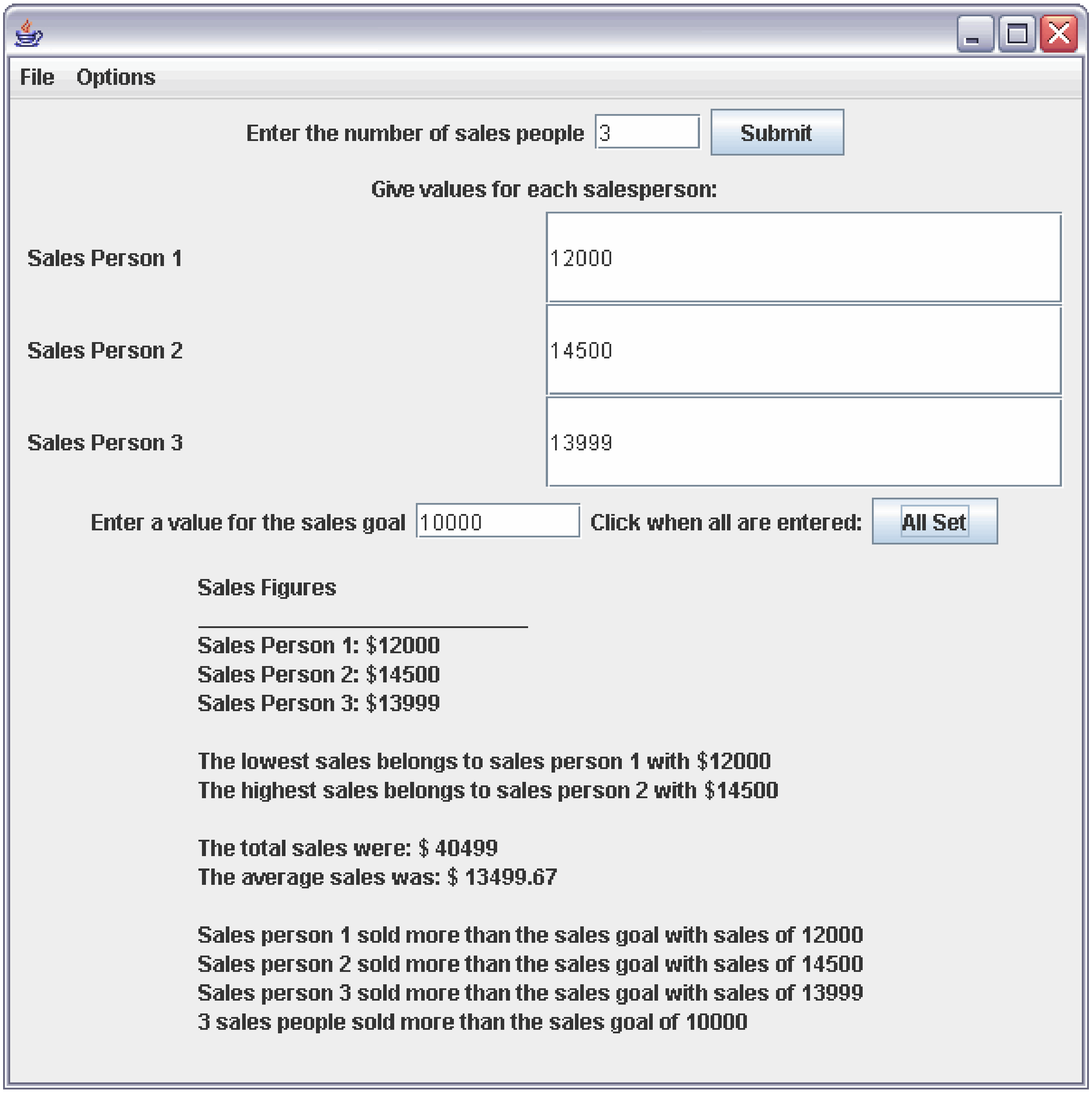
The **actionPerformed()** method of this InputPanel class determines whether the source of the **event** is a **JButton**. If it is, we use a **for loop** to go through the number of people and look in each of the TextFields defined by the **jtfSales[]** array. We use **getText()** to get the text the user typed in and we use **Integer.parseInt()** to convert that text into an integer. Then we call the **setSales()** method of the SalesApp object. We determine the value of the **goal** and set the salesBar to that value.

Okay, nice work! Take a break, pat yourself on the back, and bask in the glory of your accomplishments so far! We'll do the Output Panel for this user interface in the next lesson. See you there!

Graphical User Interfaces, continued

Making the Output Panel

Now that we've created the input panel and have output going to the console, let's create a **Panel** to display the output:



Enter the number of sales people 3

Give values for each salesperson:

Sales Person 1 12000

Sales Person 2 14500

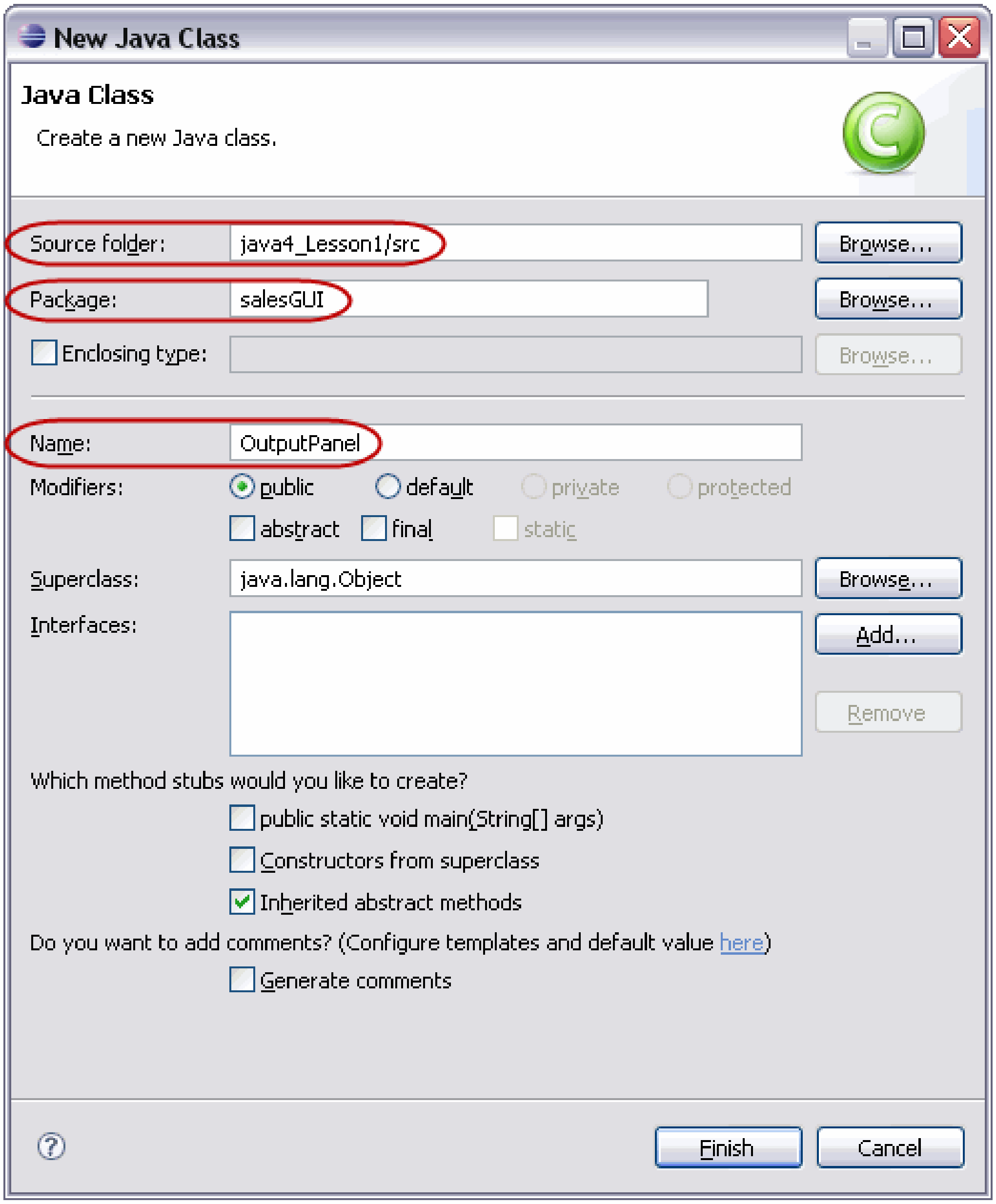
Sales Person 3 13999

Enter a value for the sales goal 10000 Click when all are entered:

```
Sales Figures
_______________________
Sales Person 1: $12000
Sales Person 2: $14500
Sales Person 3: $13999

The lowest sales belongs to sales person 1 with $12000
The highest sales belongs to sales person 2 with $14500

The total sales were: $ 40499
The average sales was: $ 13499.67

Sales person 1 sold more than the sales goal with sales of 12000
Sales person 2 sold more than the sales goal with sales of 14500
Sales person 3 sold more than the sales goal with sales of 13999
3 sales people sold more than the sales goal of 10000
```

We can take advantage of JLabel's ability to display HTML to format the output.

Using the same java4_Lesson1 project we used in the previous lesson, create an **OutputPanel** class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson1/src

Package: salesGUI

□ Enclosing type:

Name: OutputPanel

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

The output panel we're going to build will actually consist of two panels: one in the **East** area and one in the **West** area of the layout. They will be used to display the output of the application to the user.

Type **OutputPanel** as shown in **blue**:

CODE TO TYPE: OutputPanel

```
package salesGUI;

import javax.swing.*;
import java.text.DecimalFormat;

public class OutputPanel extends JPanel {

    JLabel jlSalesOutput;
    JPanel leftPanel, rightPanel;
    JLabel jlSalesBar;
    JTextField jtfSalesBar;
    JButton done;
    SalesApp app;
    int salesBar;
    int [] sales;

    public OutputPanel(SalesApp container) {
        app = container;
        sales = app.getSales();
        leftPanel = new JPanel();
        rightPanel = new JPanel();
        add("East", rightPanel);
        add("West", leftPanel);
        jlSalesOutput = new JLabel();
        rightPanel.add(jlSalesOutput);
        jlSalesOutput.setText("");
    }
}
```

Next we'll add methods to format and display the results. The **writeOutput()** method uses concatenation (**+=**) to build a **+=txtOutput** String that contains all of our output. We are using JLabel's ability to display HTML content, and displaying the data as we would in an HTML document.

Note HTML break tags (<br>) are used for new lines in the **txtOutput String**s for the output JLabel.

CODE TO EDIT: OutputPanel

```
package salesGUI;

import java.awt.Panel;
import javax.swing.*;
import java.text.DecimalFormat;

public class OutputPanel extends JPanel {

    JLabel jlSalesOutput;
    Panel leftPanel, rightPanel;
    JLabel jlSalesBar;
    JTextField jtfSalesBar;
    JButton done;
    SalesApp app;
    int salesBar;
    int [] sales;

    public OutputPanel(SalesApp container) {
        app = container;
        sales = app.getSales();
        leftPanel = new Panel();
        rightPanel = new Panel();
        add("East", rightPanel);
        add("West", leftPanel);
        jlSalesOutput = new JLabel();
        rightPanel.add(jlSalesOutput);
        jlSalesOutput.setText("");
    }

    public void refreshOutput(){
        jlSalesOutput.setText("");
    }

    protected void writeOutput(){
        app.calculateMinMax();
        DecimalFormat df1 =  new DecimalFormat("####.##");
        // Build the output string like an HTML doc
        String txtOutput =
           "<html>Sales Figures<br>__________________________<br>";
        for (int x = 0; x < sales.length; x++)
        {
            txtOutput += "Sales Person " + (x + 1) + ": $" + sales[x] + "<br>";
        }

        txtOutput += "<br>The lowest sales belongs to sales person " +
            (app.getMin() + 1) + " with $" + sales[app.getMin()] + "<br>";
        txtOutput += "The highest sales belongs to sales person " +
            (app.getMax() + 1) + " with $" + sales[app.getMax()] + "<br>";
        txtOutput += "<br>The total sales were: $ " +
            app.getTotalSales() + "<br>";
        txtOutput += "The average sales was: $ " + df1.format(app.getAverage()) +
            "<br><br>";
        txtOutput += createSalesBarInfo();
        txtOutput += "</html>";

        jlSalesOutput.setText(txtOutput);
        validate();
        repaint();
    }


    protected String createSalesBarInfo(){

        String salesBarOutput = "";
        int overSalesBar = 0;
        int [] performance = app.determineTopSalesPeople();
        int [] sales = app.getSales();

        for (int x = 0; x < sales.length; x++)
        {
            if (performance[x] ==1)
            {
                overSalesBar++;
                salesBarOutput += "Sales person " + (x + 1) +
                    " sold more than the sales goal with sales of "+ sales[x]+ "<br>" ;
            }
            else if (performance[x] ==0)
            {
               salesBarOutput += "Sales person " + (x + 1) +
                   " exactly reached the sales goal with sales of "+ sales[x]+ "<br>" ;
            }
        }
        if (overSalesBar ==1)
            salesBarOutput += "Only " + overSalesBar +
                " sales person sold more than the sales goal of " + app.getBar() + "<br
><br>";
        else
            salesBarOutput += overSalesBar +
                " sales people sold more than the sales goal of " + app.getBar() + "<br
><br>";
        return salesBarOutput;
    }

}
```

Here we use JLabel's ability to display HTML to display the sales totals of each salesperson, and which were greater than, lower than, or equal to the sales bar.

API For more information on the javax.swing.JLabel, look at the API, as well as How to Use Labels in the Java Tutorial on *Using Swing Components*.

💾 Save the **OutputPanel** class.

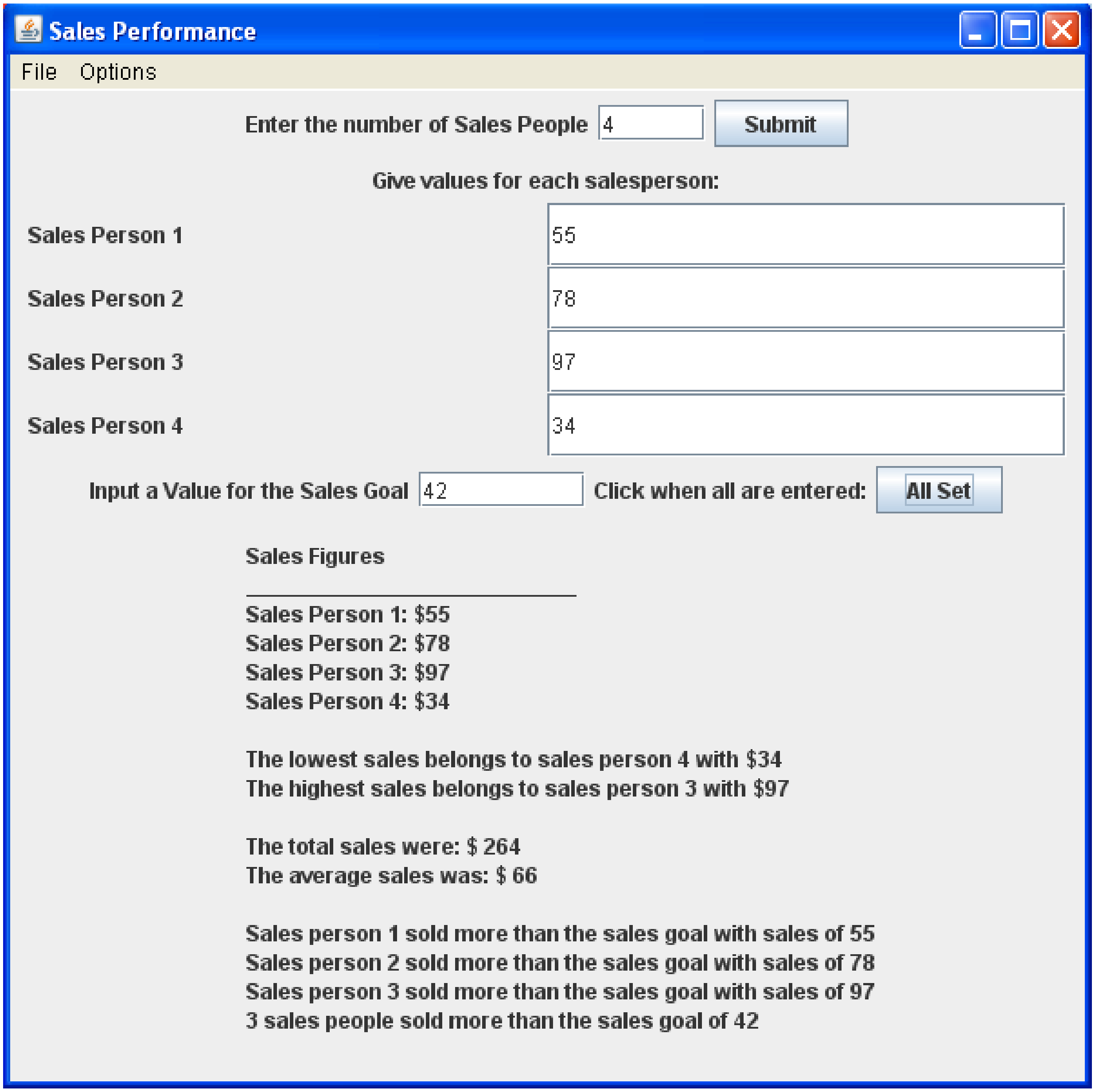
Click on **InputPanel.java**. If you haven't done so, 💾 save it now--any errors you have should go away.

Click on **SalesUserInterface.java**; it should be free of errors as well. Now let's add the Results option and OutputPanel. Edit SalesUserInterface as shown in **blue** below:

CODE TO EDIT: SalesUserInterface

```
package salesGUI;

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

public class SalesUserInterface extends JFrame {
    SalesApp app;
    JMenuBar mb;
    JMenu m, m1;
    JMenuItem q, r, s, t;
    InputPanel inputPanel;
    JLabel peopleLabel;
    JTextField peopleField;
    JButton jbNumPeople, done;
    int numPeople;
    OutputPanel results;
    boolean processed = false;

    public SalesUserInterface(SalesApp myApp) {
        app = myApp;
        app.setMyUserInterface(this);
        setLayout(new BorderLayout());
        setPreferredSize(new Dimension(600, 600));
        mb = new JMenuBar();
        setJMenuBar(mb);
        m = new JMenu("File");
        m1 = new JMenu ("Options");
        mb.add(m);
        mb.add(m1);
        m.add(q = new JMenuItem("Exit"));
        q.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });

        m1.add(t= new JMenuItem("Results"));
        t.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if (processed)
                {
                    remove(results);
                }
                results = new OutputPanel(app);
                add("South", results);
                processed = true;
                results.writeOutput();}
        });

        InitPanel specifyNumber = new InitPanel();
        add("North", specifyNumber);
        //InputPanel inputPanel = new InputPanel(app, numPeople, 2);
        //add("Center", inputPanel);
        pack();
        setVisible(true);
    }

    private class InitPanel extends JPanel {
        public InitPanel() {
            peopleLabel = new JLabel("Enter the number of sales people");
            add(peopleLabel);
            peopleField = new JTextField(5);
            add(peopleField);
            jbNumPeople = new JButton("Submit");
            add(jbNumPeople);
            jbNumPeople.addActionListener(new NumSalesPeopleListener());
        }
    }

    private class NumSalesPeopleListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            if (inputPanel != null)
            {
                remove(inputPanel);
                app = new SalesApp();
            }
            numPeople = Integer.parseInt(peopleField.getText());
            inputPanel = new InputPanel(app, numPeople, 2);
            add("Center", inputPanel);
            SalesUserInterface.this.validate();
        }
    }
}
```

Save it and run the **Main** class. Enter a number of salespeople, their sales numbers and the goal, and then click **All Set**. Select **Options | Results**, and you should see something like this:



Enter the number of Sales People 4

Give values for each salesperson:

Sales Person 1 55

Sales Person 2 78

Sales Person 3 97

Sales Person 4 34

Input a Value for the Sales Goal 42 Click when all are entered:

```
Sales Figures
_______________________
Sales Person 1: $55
Sales Person 2: $78
Sales Person 3: $97
Sales Person 4: $34

The lowest sales belongs to sales person 4 with $34
The highest sales belongs to sales person 3 with $97

The total sales were: $ 264
The average sales was: $ 66

Sales person 1 sold more than the sales goal with sales of 55
Sales person 2 sold more than the sales goal with sales of 78
Sales person 3 sold more than the sales goal with sales of 97
3 sales people sold more than the sales goal of 42
```

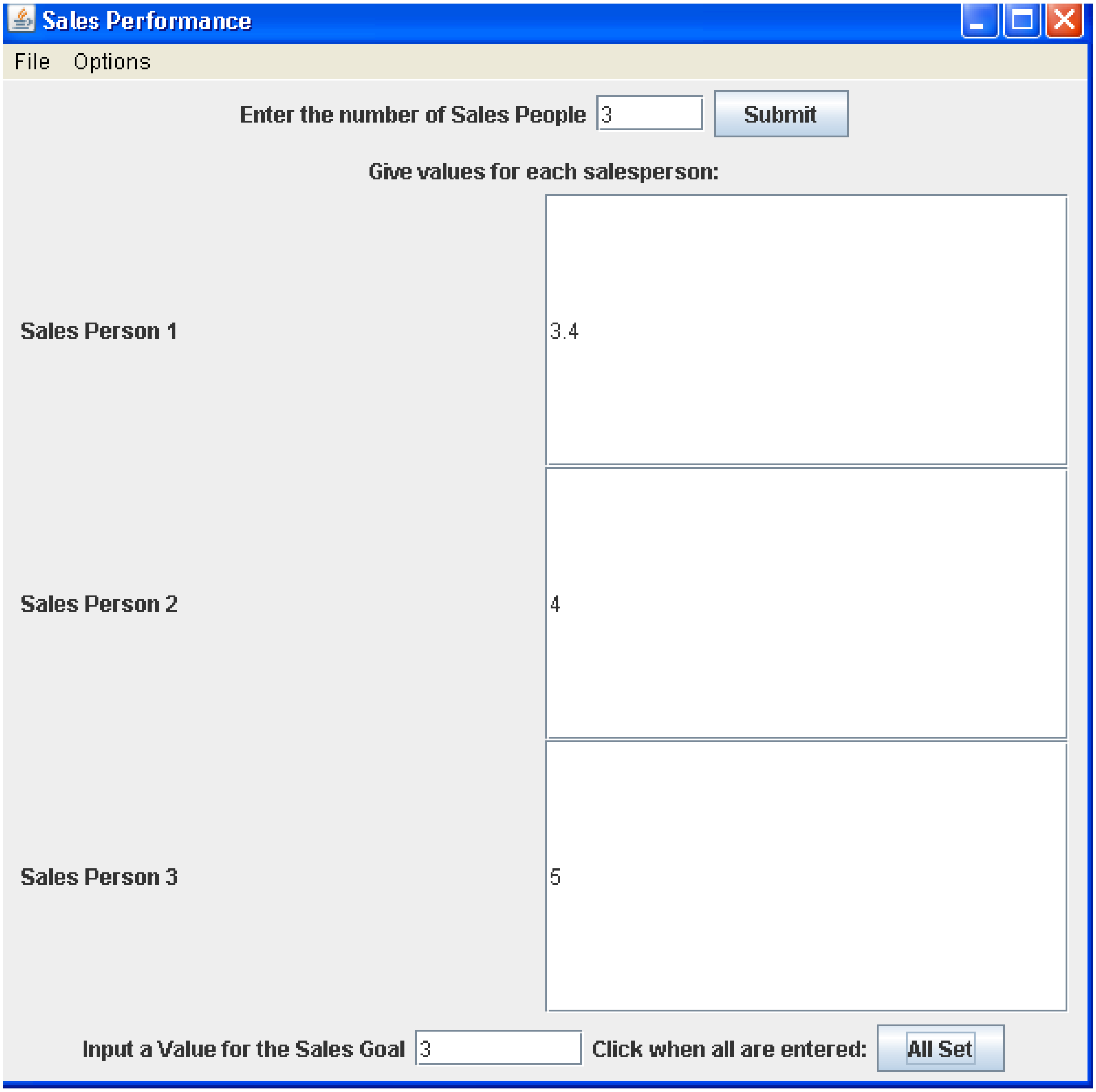
Error Checking and Exception Handling

Being Prepared for Users

Creating applications involves three basic tasks: writing the code that performs the desired function (creating the *model*), providing a clear and easy to use interface (or *view*), and making sure that users don't break the application.

Crashes

In the **java4_Lesson1** project, go to the **salesGUI** package, open the **Main.java** class, and ▶ Run it. Now give it these values:



Enter the number of Sales People: 3

Give values for each salesperson:

Sales Person 1: 3.4

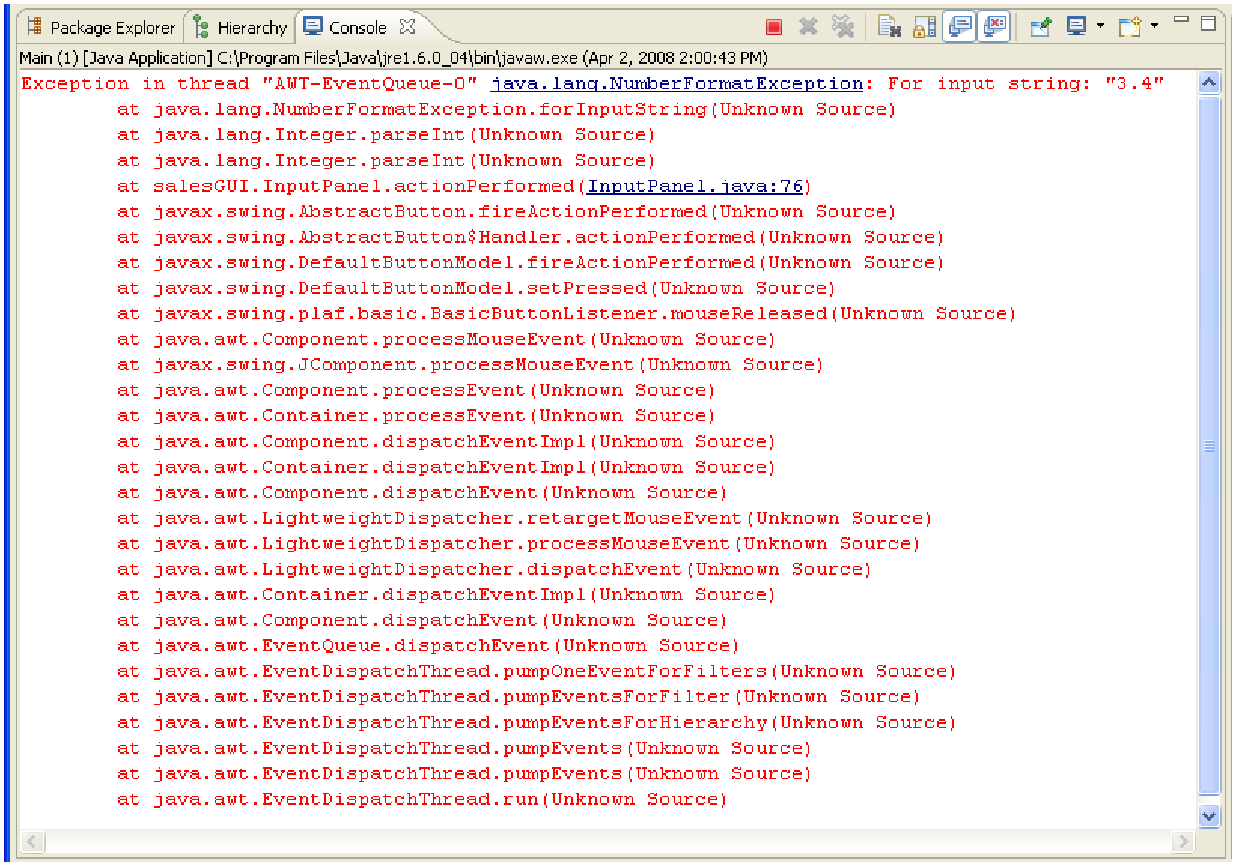
Sales Person 2: 4

Sales Person 3: 5

Input a Value for the Sales Goal: 3

Click when all are entered:

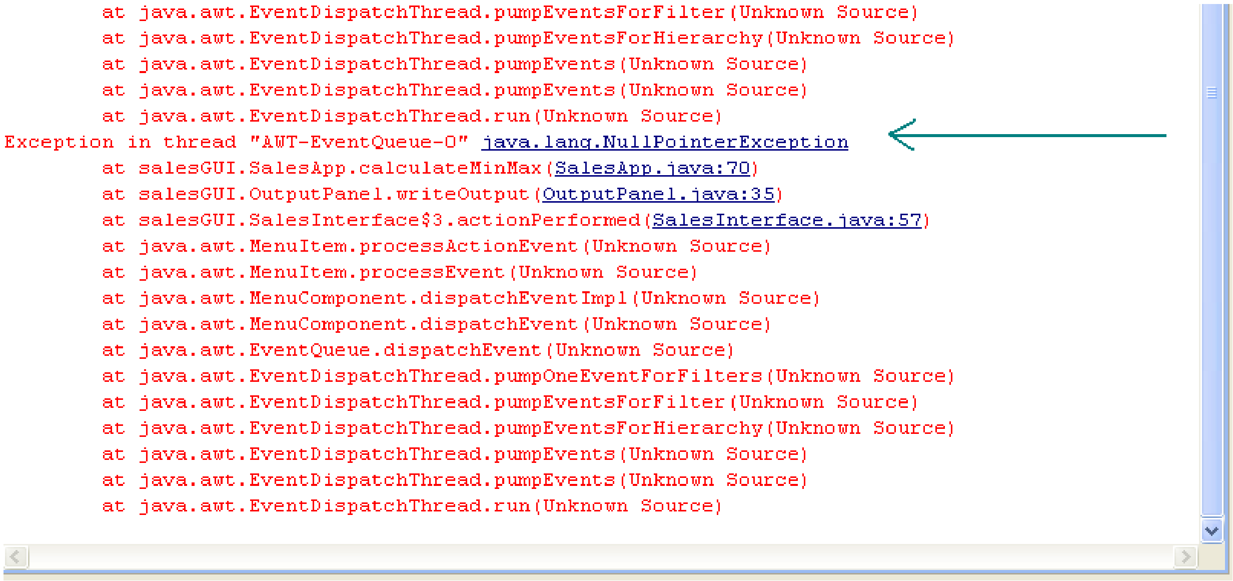
Click **All Set**. You'll see a lot of **red** in the Console:



```
Main (1) [Java Application] C:\Program Files\Java\jre1.6.0_04\bin\javaw.exe (Apr 2, 2008 2:00:43 PM)
Exception in thread "AWT-EventQueue-0" java.lang.NumberFormatException: For input string: "3.4"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at salesGUI.InputPanel.actionPerformed(InputPanel.java:76)
        at javax.swing.AbstractButton.fireActionPerformed(Unknown Source)
        at javax.swing.AbstractButton$Handler.actionPerformed(Unknown Source)
        at javax.swing.DefaultButtonModel.fireActionPerformed(Unknown Source)
        at javax.swing.DefaultButtonModel.setPressed(Unknown Source)
        at javax.swing.plaf.basic.BasicButtonListener.mouseReleased(Unknown Source)
        at java.awt.Component.processMouseEvent(Unknown Source)
        at javax.swing.JComponent.processMouseEvent(Unknown Source)
        at java.awt.Component.processEvent(Unknown Source)
        at java.awt.Container.processEvent(Unknown Source)
        at java.awt.Component.dispatchEventImpl(Unknown Source)
        at java.awt.Container.dispatchEventImpl(Unknown Source)
        at java.awt.Component.dispatchEvent(Unknown Source)
        at java.awt.LightweightDispatcher.retargetMouseEvent(Unknown Source)
        at java.awt.LightweightDispatcher.processMouseEvent(Unknown Source)
        at java.awt.LightweightDispatcher.dispatchEvent(Unknown Source)
        at java.awt.Container.dispatchEventImpl(Unknown Source)
        at java.awt.Component.dispatchEvent(Unknown Source)
        at java.awt.EventQueue.dispatchEvent(Unknown Source)
        at java.awt.EventDispatchThread.pumpOneEventForFilters(Unknown Source)
        at java.awt.EventDispatchThread.pumpEventsForFilter(Unknown Source)
        at java.awt.EventDispatchThread.pumpEventsForHierarchy(Unknown Source)
        at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
        at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
        at java.awt.EventDispatchThread.run(Unknown Source)
```

This isn't a problem for programmers--we can see the console, so we can see the **Exception** too. But it's a problem for users, because the application view doesn't change at all, so they aren't even aware that they've made an error. They can carry on and try the menu items, but they won't get their results.

In the application that's currently running, select **Options | Results**. You'll notice even more **red** in the Console:



```
        at java.awt.EventDispatchThread.pumpEventsForFilter(Unknown Source)
        at java.awt.EventDispatchThread.pumpEventsForHierarchy(Unknown Source)
        at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
        at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
        at java.awt.EventDispatchThread.run(Unknown Source)
Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException
        at salesGUI.SalesApp.calculateMinMax(SalesApp.java:70)
        at salesGUI.OutputPanel.writeOutput(OutputPanel.java:35)
        at salesGUI.SalesInterface$3.actionPerformed(SalesInterface.java:57)
        at java.awt.MenuItem.processActionEvent(Unknown Source)
        at java.awt.MenuItem.processEvent(Unknown Source)
        at java.awt.MenuComponent.dispatchEventImpl(Unknown Source)
        at java.awt.MenuComponent.dispatchEvent(Unknown Source)
        at java.awt.EventQueue.dispatchEvent(Unknown Source)
        at java.awt.EventDispatchThread.pumpOneEventForFilters(Unknown Source)
        at java.awt.EventDispatchThread.pumpEventsForFilter(Unknown Source)
        at java.awt.EventDispatchThread.pumpEventsForHierarchy(Unknown Source)
        at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
        at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
        at java.awt.EventDispatchThread.run(Unknown Source)
```

Again, the user still doesn't see any of this; they just know that the lousy program isn't working!

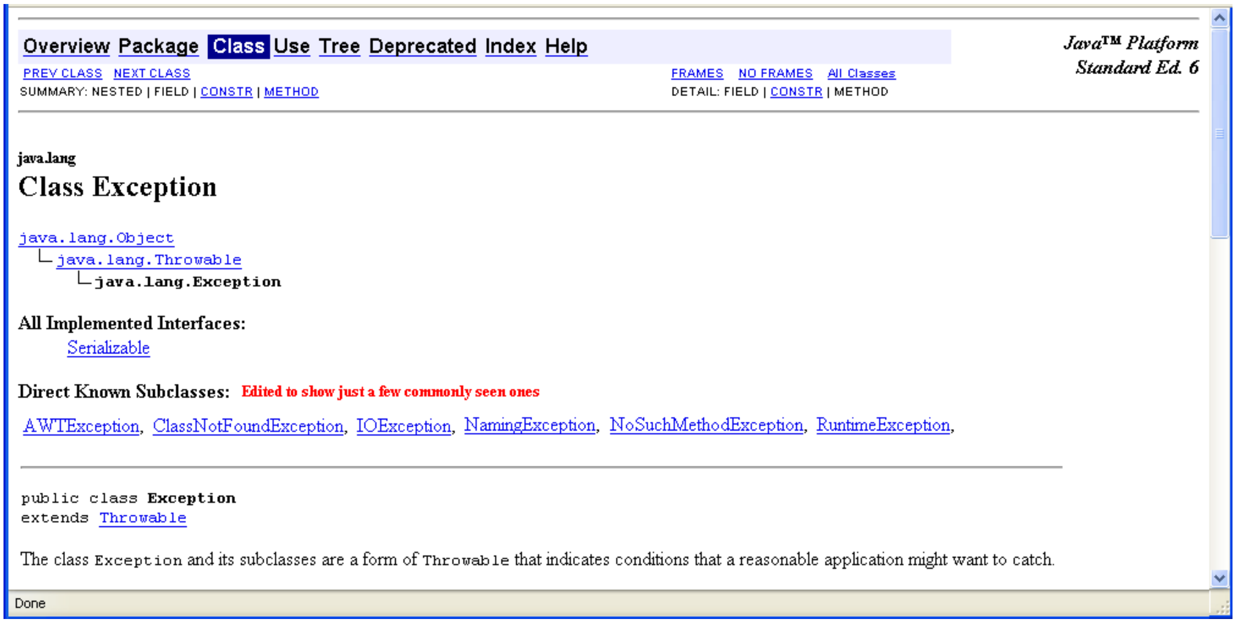
Close the running application using the menu item **File | Exit**.

If an application is open and running, it will continue to use the same **.class** (*old* compiled code) that it was opened with, even if you make changes to the application and save it again. If you edit, resave, and rerun an application, but still have the older (erroneous) code running, it can cause frustration. Always make sure to close an application properly, before editing and running a new version.

Expect the Unexpected

Exceptions occur even to our most well-thought-out Java code plans. In fact, they are so common that Java has a class *named* **Exception**.

Go to the **java.lang** package. Scroll down to the **Exception Summary**, then to the **Exception** class. There are quite a few **Direct Known Subclasses** (we edited most of them out in the image below though, because the list was so long):


Overview Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
Java™ Platform
Standard Ed. 6
Done

java.lang

Class Exception

```
java.lang.Object
  └ java.lang.Throwable
      └ java.lang.Exception
```

All Implemented Interfaces:

Serializable

Direct Known Subclasses: Edited to show just a few commonly seen ones

AWTException, ClassNotFoundException, IOException, NamingException, NoSuchMethodException, RuntimeException,

```
public class Exception
extends Throwable
```

The class `Exception` and its subclasses are a form of `Throwable` that indicates conditions that a reasonable application might want to catch.

And that's just the beginning. Go back to **java.lang**'s **Exception Summary**. Scroll down to **RuntimeException** (just one of the Direct Known Subclasses of **Exception**). Click on **RuntimeException** and check out all of its Direct Known Subclasses.

Programs and users may behave in an infinite number of unexpected ways. When the unexpected happens, our code (with the help of Java) will **throw** an **Exception** (the **java.lang.Exception** class extends (or inherits from) the class **java.lang.Throwable**).

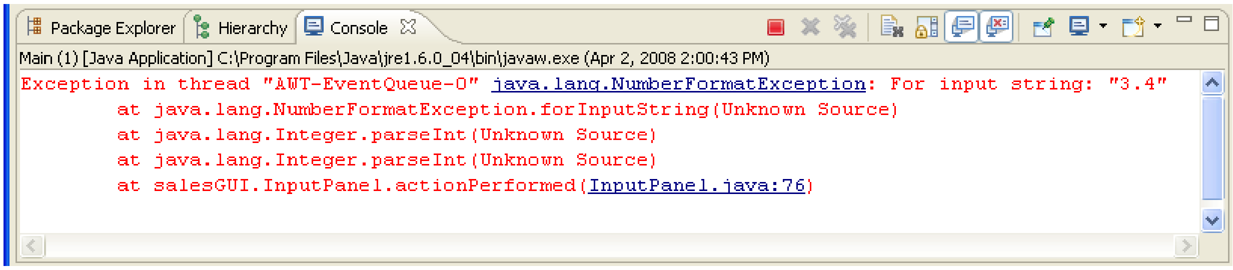
So, what's an exception? Oracle's Java tutorial says, "An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions." If we don't want our programs to crash and cause our users to become frustrated, then we need to plan for all potential **Exceptions**.

Handling Exceptions

Finding the Problem

When Java throws an **Exception**, it tells us which type of **Exception** it was and where it occurred.

In our first exception, Java provided a long *debugging trace* of its location. Usually, the last couple of lines in a trace are the most important for programmers. For example, when the user entered the value of 3.4 in the **InputPanel**, we saw:



Main (1) [Java Application] C:\Program Files\Java\jre1.6.0_04\bin\javaw.exe (Apr 2, 2008 2:00:43 PM)

```
Exception in thread "AWT-EventQueue-0" java.lang.NumberFormatException: For input string: "3.4"
	at java.lang.NumberFormatException.forInputString(Unknown Source)
	at java.lang.Integer.parseInt(Unknown Source)
	at java.lang.Integer.parseInt(Unknown Source)
	at salesGUI.InputPanel.actionPerformed(InputPanel.java:76)
```

We can tell from the **Exception** that an input string of **3.4** caused a **java.lang.NumberFormatException**. We can also determine that the exception occurred in the **InputPanel.java** class at line number **76**.

Open the InputPanel.java class and display the line numbers (on the left side bar, **right-click** and choose **Show Line Numbers**). Go to **line 76**. You should see **sales[x] = Integer.parseInt(jtfSales[x].getText());**.

Do you recognize the problem? The **sales[ ]** array is declared as **Integer**. We told Java to expect an **int**, but the user gave us a decimal. A decimal is not an **int**; it's a **double** or a **float**. So, how do we remedy this?

Fixing the Problem

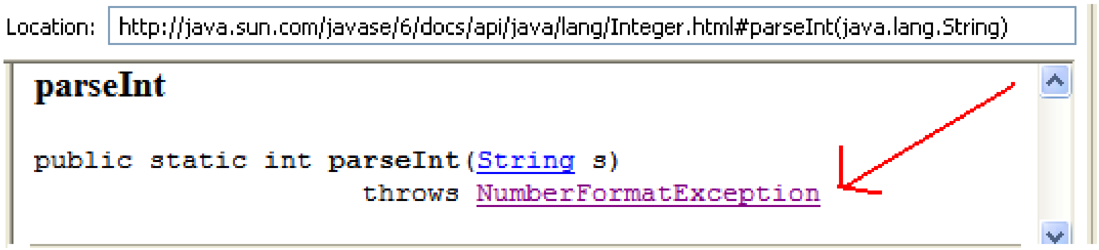
Java provides a specific structure to handle **Exceptions**. We put potential problems into **try/catch** clauses. If certain code could throw exceptions, we place it in a **try** clause, and then provide a **catch** clause to make the appropriate corrections.

Try/Catch Clauses

Anticipating Exceptions

If we want to catch exceptions, we need to know when they might occur. Sometimes code provided in the API indicates that it will throw various types of exceptions. We'll address those exceptions in greater detail later, but for now, let's get a handle on the general idea. If we had researched the methods we were using carefully in the API beforehand, we could have anticipated potential problems before writing the code.

API In the API, go to **java.lang.Integer**. Go to the **parseInt(String s)** method.



Location: http://java.sun.com/javase/6/docs/api/java/lang/Integer.html#parseInt(java.lang.String)

parseInt

```
public static int parseInt(String s)
                    throws NumberFormatException
```

We could have anticipated that a user might enter a decimal number rather than an integer. Programmers need to be ready for all kinds of potentially unexpected situations.

So, how can we be prepared? Well, if the user behaves as we would like them to, the code works great. But if the user doesn't, we need to **catch** the **Exception**. If a **method** throws an **Exception**, then we should instruct our code to **try** that method's piece of code.

If our programs do not catch exceptions, then Java is forced to **throw** them farther. Java will keep throwing exceptions until something catches it, or until it gets to the "top of the stack" (more on this in a later lesson). At that point, if an exception has not been caught, it will cause errors in the console.

Making It Right: Dialog Boxes

In our example, the problem is in the type of input given by the user, so let's tell the user when something they've entered needs to be changed. We'll do that using a dialog box.

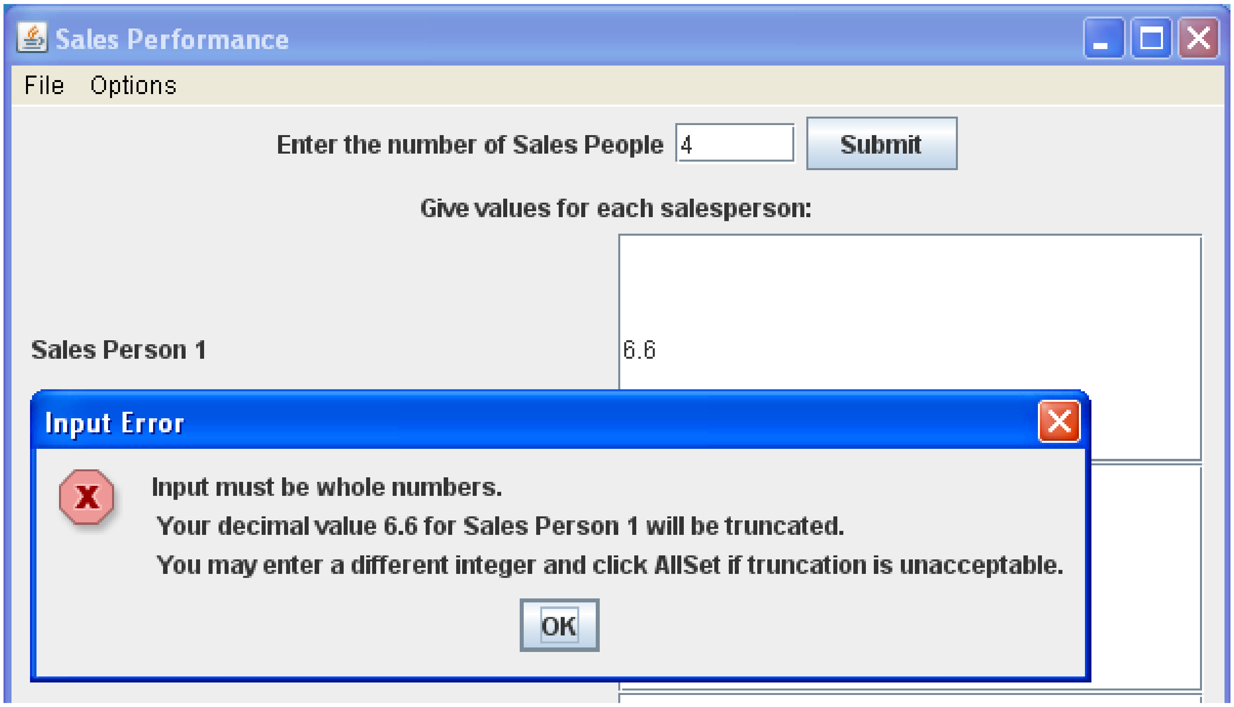
In the **InputPanel** class, where the exception occurred (around line 76), edit the **actionPerformed()** method as shown in **blue**:

CODE TO EDIT: InputPanel

```
package salesGUI;

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

public class InputPanel extends JPanel implements ActionListener {
    JPanel topPanel, middlePanel, bottomPanel, leftPanel, rightPanel;
    JLabel[] jlSales;
    JButton done;
    SalesApp app;
    JLabel prompt, doneLabel, jlSalesBar;
    JTextField[] jtfSales;
    JTextField jtfSalesBar;
    int numPeople;
    int [] sales;
    int goal;

    public InputPanel(SalesApp container, int numPeople, int gridX) {
        this.app = container;
        this.numPeople = numPeople;
        sales = new int[numPeople];

        this.setLayout(new BorderLayout());
        topPanel = new JPanel();
        topPanel.setLayout(new FlowLayout());
        middlePanel = new JPanel(new GridLayout(numPeople, gridX));
        bottomPanel = new JPanel();
        bottomPanel.setLayout(new FlowLayout());
        leftPanel = new JPanel();
        rightPanel = new JPanel();
        add("North", topPanel);
        add("Center", middlePanel);
        add("South", bottomPanel);
        add("East", rightPanel);
        add("West", leftPanel);

        jlSales = new JLabel[numPeople];
        jtfSales = new JTextField[numPeople];
        prompt = new JLabel("Give values for each salesperson:");
        topPanel.add(prompt);

        for (int x = 0; x < numPeople; x++) {
            jlSales[x] = new JLabel("Sales Person " + (x+1));
            jtfSales[x] = new JTextField("0", 8);
            middlePanel.add(jlSales[x]);
            middlePanel.add(jtfSales[x]);
        }
        jlSalesBar = new JLabel("Enter a value for the sales goal");
        bottomPanel.add(jlSalesBar);
        jtfSalesBar = new JTextField("0",8);
        //jtfSalesBar.addActionListener(new GoalButtonListener());
        bottomPanel.add(jtfSalesBar);
        doneLabel = new JLabel("Click when all are entered:");
        bottomPanel.add(doneLabel);
        done = new JButton("All Set");
        bottomPanel.add(done);
        done.addActionListener(this);
    }

    public void actionPerformed(ActionEvent event){
        if (event.getSource() instanceof JButton)
        {
            if ((JButton)event.getSource() == done)
            {
                for (int x = 0; x < numPeople; x++)
                {
                    try
                    {
                        sales[x] = Integer.parseInt(jtfSales[x].getText());  // 
throws NumberFormatException
                    }
                    catch(NumberFormatException e)
                    {
                        String messageLine1 = "Input must be whole numbers.\n ";
                        String messageLine2 = "Your decimal value " + jtfSales[x
].getText() + " for Sales Person " + (x+1) +" will be truncated.\n ";
                        String messageLine3 = "You may enter a different integer
 and click AllSet if truncation is unacceptable.";

                        JOptionPane.showMessageDialog(this, messageLine1+message
Line2+messageLine3,"Input Error", JOptionPane.ERROR_MESSAGE);

                        sales[x]= (int)Double.parseDouble(jtfSales[x].getText())
;
                        jtfSales[x].setText(Integer.toString(sales[x]));
                    }
                }

                app.setSales(sales);
                goal = Integer.parseInt(jtfSalesBar.getText());  // so don't hav
e to be sure they hit enter
                app.setSalesBar(goal);
            }
        }
    }
}
```

Save the InputPanel class.

Run the Main class. Enter a decimal number for one of the values and click **All Set**.


Sales Performance
File
Options
Enter the number of Sales People
4
Submit
Give values for each salesperson:
Sales Person 1
6.6
Input Error
OK

Input must be whole numbers.
Your decimal value 6.6 for Sales Person 1 will be truncated.
You may enter a different integer and click AllSet if truncation is unacceptable.

That information helps the user and the programmer. We can provide additional **Strings** of information for the user in the dialog boxes too. Or we can just fix things without notifying them at all. Choosing how to respond depends on the application and the significance of each piece of data.

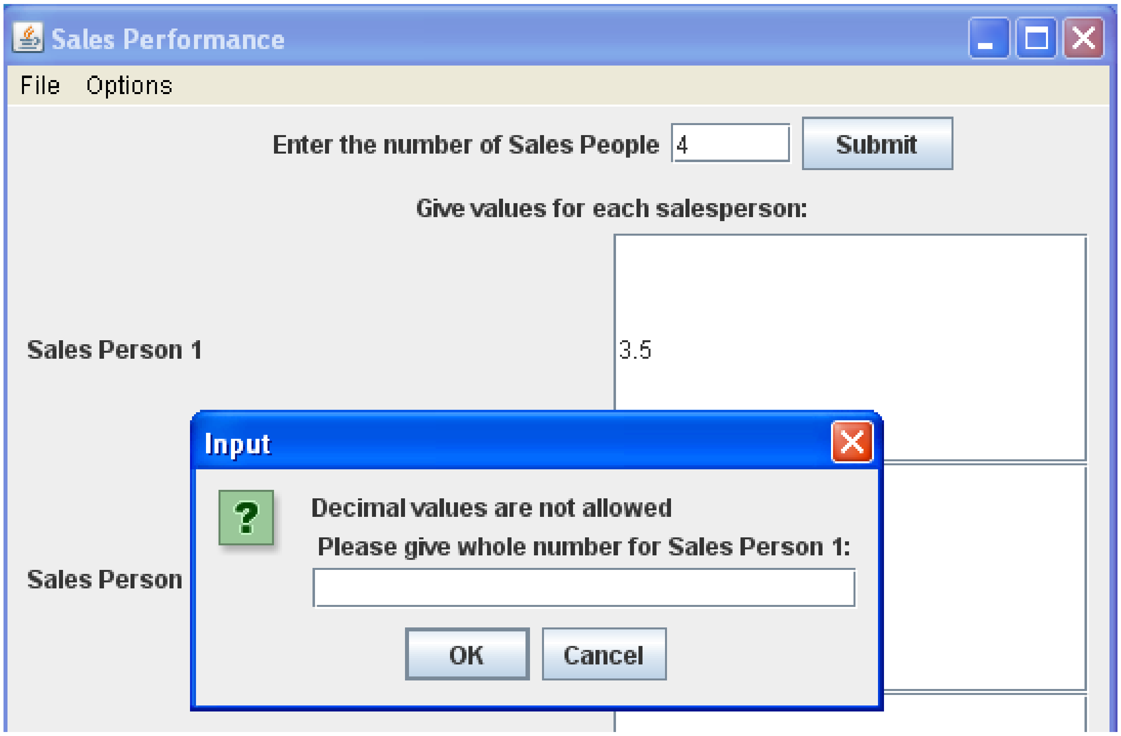
There are several other types of dialog box options; let's take a look at another one. Replace the **catch** clause inside the **for** loop as shown in **blue**:

CODE TO EDIT: InputPanel

```
package salesGUI;

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

public class InputPanel extends JPanel implements ActionListener {
    JPanel topPanel, middlePanel, bottomPanel, leftPanel, rightPanel;
    JLabel[] jlSales;
    JButton done;
    SalesApp app;
    JLabel prompt, doneLabel, jlSalesBar;
    JTextField[] jtfSales;
    JTextField jtfSalesBar;
    int numPeople;
    int [] sales;
    int goal;

    public InputPanel(SalesApp container, int numPeople, int gridX) {
        this.app = container;
        this.numPeople = numPeople;
        sales = new int[numPeople];

        this.setLayout(new BorderLayout());
        topPanel = new JPanel();
        topPanel.setLayout(new FlowLayout());
        middlePanel = new JPanel(new GridLayout(numPeople, gridX));
        bottomPanel = new JPanel();
        bottomPanel.setLayout(new FlowLayout());
        leftPanel = new JPanel();
        rightPanel = new JPanel();
        add("North", topPanel);
        add("Center", middlePanel);
        add("South", bottomPanel);
        add("East", rightPanel);
        add("West", leftPanel);

        jlSales = new JLabel[numPeople];
        jtfSales = new JTextField[numPeople];
        prompt = new JLabel("Give values for each salesperson:");
        topPanel.add(prompt);

        for (int x = 0; x < numPeople; x++) {
            jlSales[x] = new JLabel("Sales Person " + (x+1));
            jtfSales[x] = new JTextField("0", 8);
            middlePanel.add(jlSales[x]);
            middlePanel.add(jtfSales[x]);
        }
        jlSalesBar = new JLabel("Enter a value for the sales goal");
        bottomPanel.add(jlSalesBar);
        jtfSalesBar = new JTextField("0",8);
        //jtfSalesBar.addActionListener(new GoalButtonListener());
        bottomPanel.add(jtfSalesBar);
        doneLabel = new JLabel("Click when all are entered:");
        bottomPanel.add(doneLabel);
        done = new JButton("All Set");
        bottomPanel.add(done);
        done.addActionListener(this);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() instanceof JButton)
        {
            if ((JButton)event.getSource() == done)
            {
                for (int x = 0; x < numPeople; x++)
                {
                    try
                    {
                        sales[x] = Integer.parseInt(jtfSales[x].getText());
                    }
                    catch(NumberFormatException e)
                    {
                        String temp = JOptionPane.showInputDialog("Decimal value
s are not allowed.\n Please give a whole number for Sales Person " + (x+1) + ":
");
                        sales[x] = Integer.parseInt(temp);
                        jtfSales[x].setText(Integer.toString(sales[x]));
                    }
                }

                app.setSales(sales);
                goal = Integer.parseInt(jtfSalesBar.getText());
                app.setSalesBar(goal);
            }
        }
    }
}
```

Save the InputPanel class.

Run the Main class. Enter a decimal number for one of the values and click **All Set**:



Enter the number of Sales People 4

Give values for each salesperson:

Sales Person 1 3.5

Sales Person

Decimal values are not allowed

Please give whole number for Sales Person 1:

For more on dialog boxes, see the Oracle tutorial on How to Make Dialogs.

Types of Exceptions

The Java programming language uses **exceptions** to handle errors and other exceptional events. **Exceptions** are unusual conditions that a well-written application will anticipate and remedy. Java provides two main types of exceptions: checked and unchecked. **Checked exceptions** can be *checked* at compile time. All exceptions are checked exceptions, except for those that are instances of the **Error** and **RuntimeException** classes and their subclasses.

Checked Exceptions

If a method has a *checked* exception, Java informs the programmer using the method. The class that uses the method will not compile (or Eclipse will report errors) and the programmer will not be able to run the program until the exception in the code has been handled; the programmer is **forced** to handle that exception.

In addition, programmers can often anticipate problems that could occur in a method they have written. The author of the method is obliged to warn other programmers who may use it, that such problems are a possibility. A good programmer will handle those problems within the application. Programmers must consider other programmers, as well as users when writing methods and applications:

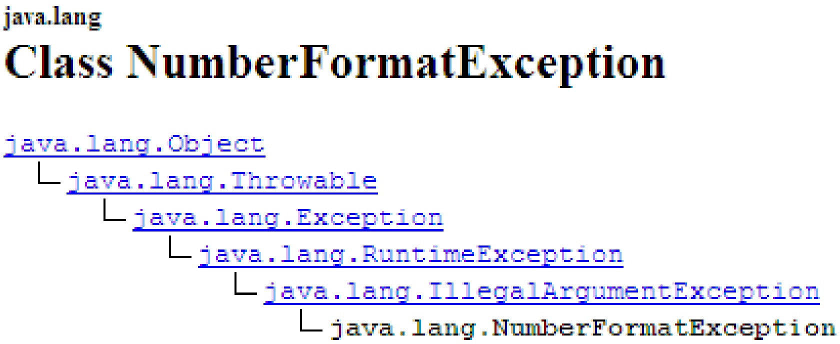
- A method's author needs to make sure that other *programmers* who use the method don't experience surprise failures.
- An application's author needs to make sure that the *users* of their application don't have surprise failures.

Of course, the author of a method can't always anticipate which environment a programmer will use, or the type of application a programmer may want to create. Because of such variables, the method author can't predict how each application might handle a problem. The best the method author can do is to inform users of the method that a problem *might* exist, and that using the method *might* throw an **Exception**. Then the method's author should include **throws** in the method definition.

Unchecked Exceptions

Errors, Runtime Exceptions, and their subclasses are **unchecked exceptions**. The code we wrote to retrieve user-entered sales values had the potential to present the problems associated with **unchecked exceptions**. Its specification in the API clearly stated that it throws a **NumberFormatException**. But we were still able to compile and run the code initially without a try/catch clause. Why?

API Go to **java.lang.NumberFormatException** in the API and look at its class hierarchy:



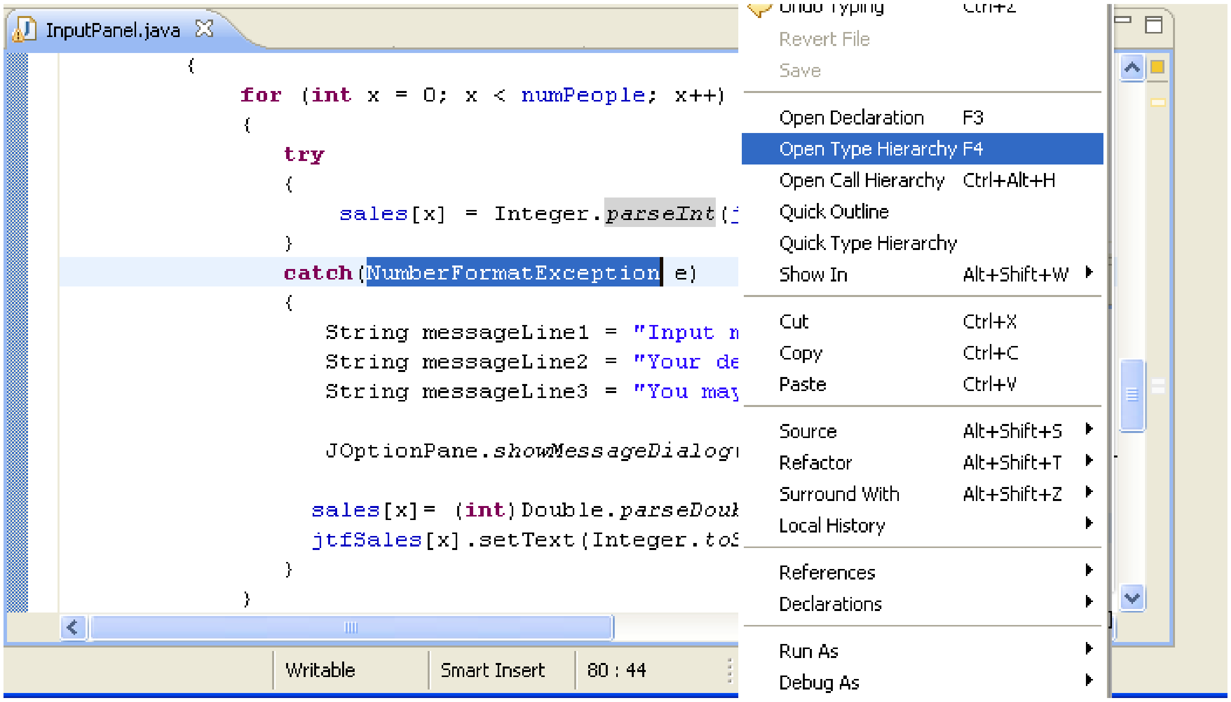
java.lang

Class NumberFormatException

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.lang.RuntimeException
              └java.lang.IllegalArgumentException
                  └java.lang.NumberFormatException
```

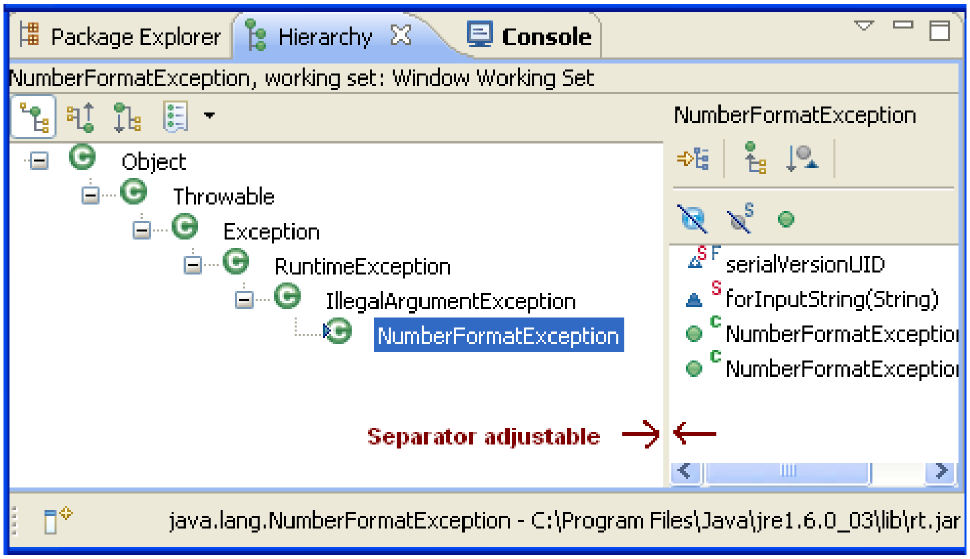
OR

Open **InputPanel.java** in the Editor. Go to the line that specifies the NumberFormatException in the catch clause. Highlight it. Right-click and choose **Open Type Hierarchy**:



```
{
    for (int x = 0; x < numPeople; x++)
    {
        try
        {
            sales[x] = Integer.parseInt(
        }
        catch(NumberFormatException e)
        {
            String messageLine1 = "Input n
            String messageLine2 = "Your de
            String messageLine3 = "You may

            JOptionPane.showMessageDialog(

          sales[x]= (int)Double.parseDoub
          jtfSales[x].setText(Integer.toS
        }
    }
```

A hierarchy window opens in the left panel (there are actually two panels there):



NumberFormatException is a subclass of **RuntimeException**. **All** exceptions are checked exceptions, **except** for those that are instances of the **Error** and **RuntimeException** classes, and their subclasses.

Sometimes exceptions arise when a program is *run*, depending on which variables are present at that particular time. These are called *Runtime Exceptions*.

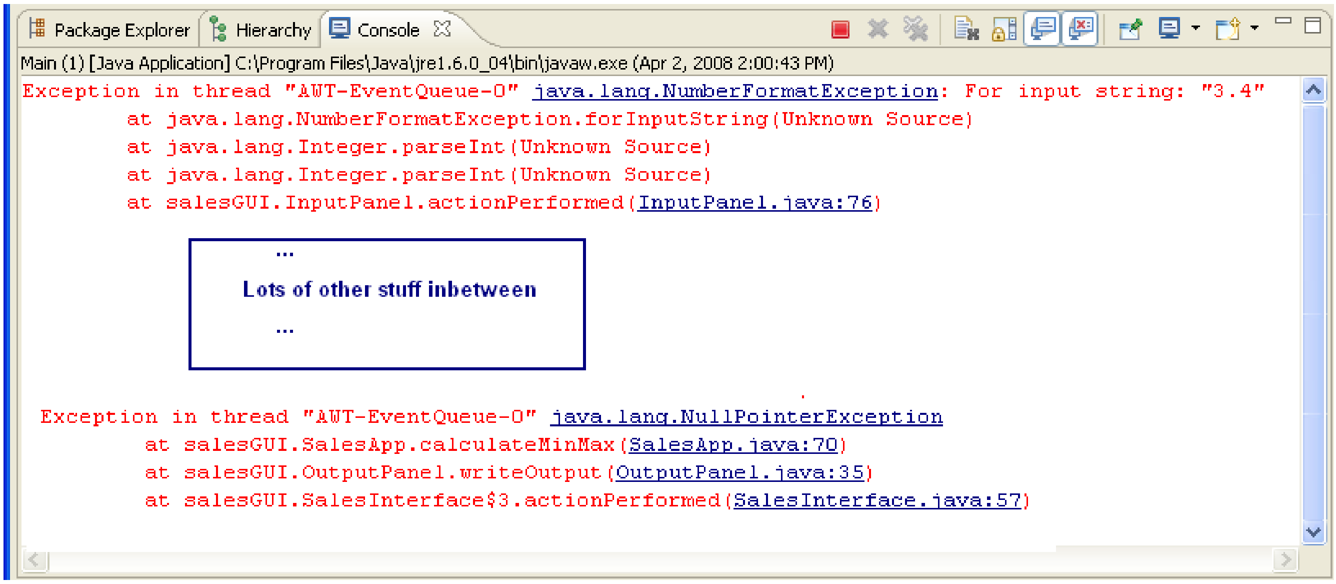
The exceptions that we have looked at in this lesson have been *unchecked (Runtime Exceptions)*, because the compiler cannot anticipate what a user will enter. So, even though the method **java.lang.Integer.parseInt(String s)** states that it might throw an exception, Java allowed the code to compile. As the Oracle Tutorial states:

Runtime exceptions represent problems that are the result of a programming problem, and as such, the API client code cannot reasonably be expected to recover from them or to handle them in any way. Such problems include arithmetic exceptions, such as dividing by zero; pointer exceptions, such as trying to access an object through a null reference; and indexing exceptions, such as attempting to access an array element through an index that is too large or too small.

Because such exceptions can happen anywhere in a program, and often runtime exceptions are not easy to spot, the compiler doesn't require programmers to catch runtime exceptions. But sooner or later, exceptions will make their presence known. The Java Virtual Machine is merciless and won't hesitate to broadcast the **red** details of our uncaught exceptions all over the console.

The Other Problem

When we ran our application at the beginning of this lesson, we saw two exceptions:



```
Main (1) [Java Application] C:\Program Files\Java\jre1.6.0_04\bin\javaw.exe (Apr 2, 2008 2:00:43 PM)
Exception in thread "AWT-EventQueue-0" java.lang.NumberFormatException: For input string: "3.4"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at salesGUI.InputPanel.actionPerformed(InputPanel.java:76)

                ...
            Lots of other stuff inbetween
                ...

 Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException
        at salesGUI.SalesApp.calculateMinMax(SalesApp.java:70)
        at salesGUI.OutputPanel.writeOutput(OutputPanel.java:35)
        at salesGUI.SalesInterface$3.actionPerformed(SalesInterface.java:57)
```

When we fixed the first exception (the **NumberFormatException**) with the **try/catch** clause, the second exception disappeared.

So, we're going to do what most sensible beginning programmers do: forget about it. But remember to expect the unexpected. That problem **will** pop up again.

Be prepared to see more exceptions in the coming lessons as we continue to investigate...

Unchecked Exceptions: Keeping Our Applications Running

About Exceptions

We know that all exceptions are checked exceptions, aside from those identified by **Error** and **RuntimeException** *and their subclasses*. If code includes methods with checked exceptions, that code won't compile until the exceptions are handled through **try/catch** clauses. A method that causes a checked exception has to specify that it *throws* the exception.

In the previous lesson, we were able to compile and run our application code, so we know our code didn't contain methods with checked exceptions. Does that mean our code is free from errors and exceptions? Sadly no. As we saw in the last lesson, our code contained one unchecked exception that had to be fixed. Unchecked exceptions usually occur because a user does something unexpected at runtime. In this lesson, we'll look at common subclasses of **RuntimeExceptions**.

Run-Time Exceptions

Subclasses of **RuntimeExceptions** appear in our code often, but sometimes they're hard to locate. These are the most commonly used subclasses of **RuntimeExceptions**:

- NullPointerException
- ArithmeticException
- Array Out of Bounds

NullPointerException

A null pointer exception occurs when your code tries to access an instance of an object that has not been properly instantiated. *Declaring* a variable to be of a certain type is **not** the same as *instantiating* it. If your variable is not of a primitive data type, it has been declared as a type of **Object** (remember that every object inherits from **Object** and is a subclass). If such an object has not been instantiated, then it doesn't point to anything in memory, so it's a *null pointer*.

Null pointer problems may occur when a user calls a method incorrectly. If an argument is null, the method might throw a NullPointerException, which is an unchecked exception. Let's look at such an exception in the first of five examples we'll use in this lesson:

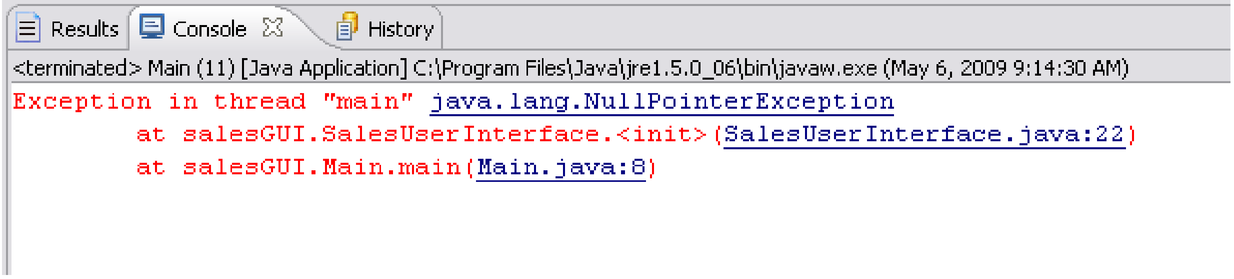
Example 1

In the java4_Lesson1 project, SalesGUI package, edit **Main.java** as shown in **blue** and **red**:

CODE TO TYPE: Main

```
package salesGUI;

public class Main {
    // declare a class variable that is set to null by default
    public static SalesApp newApp;

    public static void main(String[] args) {
        // comment the next line out so it looks like we forgot to instantiate i
t
        //SalesApp newApp = new SalesApp();
        SalesUserInterface appFrame = new SalesUserInterface(newApp);
    }
}
```

Save and run it. Look in the Console for the exception:



```
<terminated> Main (11) [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (May 6, 2009 9:14:30 AM)
Exception in thread "main" java.lang.NullPointerException
	at salesGUI.SalesUserInterface.<init>(SalesUserInterface.java:22)
	at salesGUI.Main.main(Main.java:8)
```

In this code, we passed a variable **newApp** that was **null** for **SalesApp**. It is not always that easy to find exceptions, particularly when we instantiate the objects in one place and access them in another.

Edit **Main.java** as shown (we're reverting to the previous version, so you can use the Undo key combination [**Ctrl+Z**] to undo the typing you did earlier):

CODE TO EDIT: Main

```
package salesGUI;

public class Main {
    // Remove this and the next line
    public static SalesApp newApp;

    public static void main(String[] args) {    // two classes to instantiate--the application and its GUI
        // Remove this line and the double-slash from the beginning of the next line
        // SalesApp newApp = new SalesApp();
        SalesUserInterface appFrame = new SalesUserInterface(newApp);  // tell the interface who its app is
    }
}
```

Save and run it again to make sure that all's well. Make sure to exit the running application afterward, so it's ready for the next test.

Example 2

Open the **InputPanel.java** class. In the constructor, comment out the line where we *instantiate* the **sales** array, as shown in **red**:

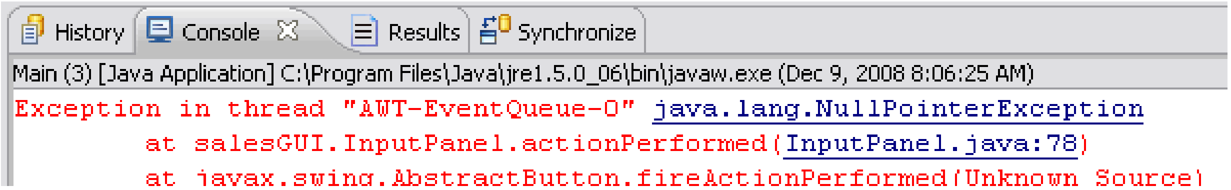
CODE TO EDIT: InputPanel

```
package salesGUI;

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

public class InputPanel extends JPanel implements ActionListener {
    JPanel topPanel, middlePanel, bottomPanel, leftPanel, rightPanel;
    JLabel[] jlSales;
    JButton done;
    SalesApp app;
    JLabel prompt, doneLabel, jlSalesBar;
    JTextField[] jtfSales;
    JTextField jtfSalesBar;
    int numPeople;
    int [] sales;
    int goal;

    public InputPanel(SalesApp container, int numPeople, int gridX){
        this.app = container;
        this.numPeople = numPeople;
        // sales = new int[numPeople];

        this.setLayout(new BorderLayout());
        topPanel = new JPanel();
        topPanel.setLayout(new FlowLayout());
        middlePanel = new JPanel(new GridLayout(numPeople, gridX));
        bottomPanel = new JPanel();
        bottomPanel.setLayout(new FlowLayout());
        leftPanel = new JPanel();
        rightPanel = new JPanel();
        add("North", topPanel);
        add("Center", middlePanel);
        add("South", bottomPanel);
        add("East", rightPanel);
        add("West", leftPanel);

        jlSales = new JLabel[numPeople];
        jtfSales = new JTextField[numPeople];
        prompt = new JLabel("Give values for each salesperson:");
        topPanel.add(prompt);

        for (int x = 0; x < numPeople; x++)
        {
            jlSales[x] = new JLabel("Sales Person " + (x+1));
            jtfSales[x] = new JTextField("0",8);
            middlePanel.add(jlSales[x]);
            middlePanel.add(jtfSales[x]);
        }
        jlSalesBar = new JLabel("Enter a value for the sales goal");
        bottomPanel.add(jlSalesBar);
        jtfSalesBar = new JTextField("0",8);
        bottomPanel.add(jtfSalesBar);
        doneLabel = new JLabel("Click when all are entered:");
        bottomPanel.add(doneLabel);
        done = new JButton("All Set");
        bottomPanel.add(done);
        done.addActionListener(this);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() instanceof JButton)
        {
            if ((JButton)event.getSource() == done)
            {
                for (int x = 0; x < numPeople; x++)
                {
                    try
                    {
                        sales[x] = Integer.parseInt(jtfSales[x].getText());  //
throws NumberFormatException
                    }
                    catch(NumberFormatException e)
                    {
                        String temp = JOptionPane.showInputDialog("Decimal value
s are not allowed.\n Please give a whole number for Sales Person " + (x+1) + ":
");
                        sales[x] = Integer.parseInt(temp);
                        jtfSales[x].setText(Integer.toString(sales[x]));
                    }
                }
                app.setSales(sales);
                goal = Integer.parseInt(jtfSalesBar.getText());
                app.setSalesBar(goal);
            }
        }
    }
}
```

Save it.

Run the **Main.java** class. Enter a value for the number of SalesPeople and click **Submit** to open the InputPanel. Enter a value for a particular Salesperson, then click **All Set**.

Scroll to the top of the exception trace in the Console.



```
Main (3) [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Dec 9, 2008 8:06:25 AM)
Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException
        at salesGUI.InputPanel.actionPerformed(InputPanel.java:78)
        at javax.swing.AbstractButton.fireActionPerformed(Unknown Source)
```

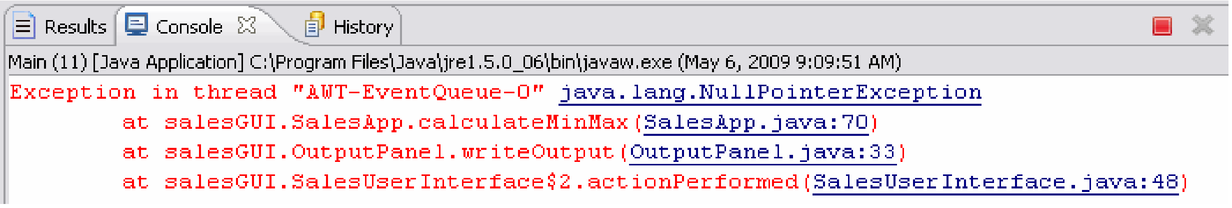
At or near line 78 in the InputPanel class is the line **sales[x] = Integer.parseInt(jtfSales[x].getText());** (your line numbers may vary slightly, depending on how you coded your dialog box.) The error message appears because we commented out the instantiation of the **sales [ ]** array, so it's not there to have elements added into it. New programmers often think that because they *declared* the array, it exists: **int [] sales;** (at or near line 16 in **InputPanel**). When you declare a variable as an instance variable and it is an Object, Java gives it the *default value* of null. So in the constructor, you need the line **sales = new int[numPeople];** to allow your variable a non-null value and size.

Close this application instance using **File | Exit**, uncomment (in other words, remove the // from) the line you commented out in **InputPanel.java**. Save it, and run it again from Main to make sure it works. Then, exit the application again, using **File | Exit**.

Example 3

Here's another potential snag in our current application:

Run the **Main.java** class and enter values for Sales People and for the Sales Goal, but *do not* click **All Set**. Select **Options | Results** in the menu. Take a look at the Console:



```
Main (11) [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (May 6, 2009 9:09:51 AM)
Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException
        at salesGUI.SalesApp.calculateMinMax(SalesApp.java:70)
        at salesGUI.OutputPanel.writeOutput(OutputPanel.java:33)
        at salesGUI.SalesUserInterface$2.actionPerformed(SalesUserInterface.java:48)
```

Your line numbers may be slightly different. Does this look familiar? It's the exception we didn't fix in the previous lessons.

We'll trace this error from the bottom up:

OBSERVE

```
at salesGUI.SalesUserInterface$3.actionPerformed(SalesUserInterface.java:48)

at salesGUI.OutputPanel.writeOutput(OutputPanel.java:33)

at salesGUI.SalesApp.calculateMinMax( SalesApp.java:70)
```

SalesUserInterface.java:48 is **results.writeOutput();}});**. **results** is an instance of **OutputPanel**. We are calling its method, **writeOutput()**.

OutputPanel.java:33 is the first line in that method. There is a call to **app.calculateMinMax();**. **app** is an instance of **SalesApp**; we are calling its method **calculateMinMax()**.

SalesApp.java:70 is the first line in that method. We see **int minimum = sales[0];**--so, why the null pointer exception? Because we don't have a **sales[0]**. The user hasn't clicked **SetAll**, so the **sales []** array never received its values, and so **sales[0]** doesn't exist.

There are various ways to fix these problems. We'll illustrate just one. All of the menu items were made in the **SalesUserInterface** class, so let's look there to find out how to fix it.

Open the **SalesUserInterface.java** class. See the constructor, where the Results MenuItem is added at **m1.add(t = new MenuItem("Results"));**. Check out the **ActionListener** and its requirements.

The **MenuItem** requires the array set in order to perform the methods described in its **ActionListener**. Here are some possible remedies:

- Set a flag to indicate whether the array has been set and if not, set it.
- Ask the user to click the **AllSet** button.
- Set everything to zeros so we start with a known quantity.
- Make sure that it is set by doing it ourselves again.

For this example, let's set all of the values to whatever is currently in the **JTextFields**. To do that we'll need to check the inputs again and then set the array. Also, we'll need another method that does almost the same thing as **actionPerformed()** in **InputPanel**. To promote modularity of code, edit **InputPanel**. There are two major changes: **actionPerformed()** is edited and much of its contents go into a new method named **setAllInputs()**:

CODE TO EDIT: InputPanel

```
package salesGUI;

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

public class InputPanel extends JPanel implements ActionListener {
    JPanel topPanel, middlePanel, bottomPanel, leftPanel, rightPanel;
    JLabel[] jlSales;
    JButton done;
    SalesApp app;
    JLabel prompt, doneLabel, jlSalesBar;
    JTextField[] jtfSales;
    JTextField jtfSalesBar;
    int numPeople;
    int [] sales;
    int goal;

    public InputPanel(SalesApp container, int numPeople, int gridX) {
        this.app = container;
        this.numPeople = numPeople;
        sales = new int[numPeople];

        this.setLayout(new BorderLayout());
        topPanel = new JPanel();
        topPanel.setLayout(new FlowLayout());
        middlePanel = new JPanel(new GridLayout(numPeople, gridX));
        bottomPanel = new JPanel();
        bottomPanel.setLayout(new FlowLayout());
        leftPanel = new JPanel();
        rightPanel = new JPanel();
        add("North", topPanel);
        add("Center", middlePanel);
        add("South", bottomPanel);
        add("East", rightPanel);
        add("West", leftPanel);

        jlSales = new JLabel[numPeople];
        jtfSales = new JTextField[numPeople];
        prompt = new JLabel("Give values for each salesperson:");
        topPanel.add(prompt);

        for (int x = 0; x < numPeople; x++)
        {
            jlSales[x] = new JLabel("Sales Person " + (x+1));
            jtfSales[x] = new JTextField("0",8);
            middlePanel.add(jlSales[x]);
            middlePanel.add(jtfSales[x]);
        }
        jlSalesBar = new JLabel("Enter a value for the sales goal");
        bottomPanel.add(jlSalesBar);
        jtfSalesBar = new JTextField("0",8);
        bottomPanel.add(jtfSalesBar);
        doneLabel = new JLabel("Click when all are entered:");
        bottomPanel.add(doneLabel);
        done = new JButton("All Set");
        bottomPanel.add(done);
        done.addActionListener(this);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() instanceof JButton)
        {
            if ((JButton)event.getSource() == done)
            {
                setAllInputs();    // all of the code that was here is
```

```
                                   //now in the method named setAllInputs
            }
        }
    }

    public void setAllInputs(){
        for (int x = 0; x < numPeople; x++)
        {
            try
            {
                sales[x] = Integer.parseInt(jtfSales[x].getText());
            }
            catch(NumberFormatException e)
            {
                String messageLine1 = "Input must be whole numbers.\n ";
                String messageLine2 = "Your decimal value " + jtfSales[x].getTex
t() + " for Sales Person " + (x+1) +" will be truncated.\n ";
                String messageLine3 = "You may enter a different integer and cli
ck AllSet if truncation is unacceptable.";

                JOptionPane.showMessageDialog(this, messageLine1+messageLine2+me
ssageLine3,"Input Error", JOptionPane.ERROR_MESSAGE);

                sales[x]= (int)Double.parseDouble(jtfSales[x].getText());
                jtfSales[x].setText(Integer.toString(sales[x]));
            }
        }

        app.setSales(sales);
        goal = Integer.parseInt(jtfSalesBar.getText());  // so don't have to be
sure they hit enter
        app.setSalesBar(goal);
    }
}
```

Save it and run **Main.java**. Once you've confirmed that it still works correctly, fix the menu choices. Edit **SalesUserInterface** as shown in **blue**:

CODE TO EDIT: SalesUserInterface

```
package salesGUI;

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

public class SalesUserInterface extends JFrame {
    SalesApp app;
    JMenuBar mb;
    JMenu m, m1;
    JMenuItem q, r, s, t;
    InputPanel inputPanel;
    JLabel peopleLabel;
    JTextField peopleField;
    JButton jbNumPeople, done;
    int numPeople;
    OutputPanel results;
    boolean processed = false;

    public SalesUserInterface(SalesApp myApp) {
        app = myApp;
        app.setMyUserInterface(this);
        setLayout(new BorderLayout());
        setPreferredSize(new Dimension(600, 600));
        mb = new JMenuBar();
        setJMenuBar(mb);
        m = new JMenu("File");
        m1 = new JMenu ("Options");
        mb.add(m);
        mb.add(m1);
        m.add(q = new JMenuItem("Exit"));
        q.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });

        m1.add(t= new MenuItem("Results"));
        t.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e){
                inputPanel.setAllInputs();  // added method call to make sure al
l is set
                if (processed)
                {
                    remove(results);
                }
                results = new OutputPanel(app);
                add("South", results);
                processed = true;
                results.writeOutput();
        }});

        InitPanel specifyNumber = new InitPanel();     // a panel to set up how
many salespeople
        add("North", specifyNumber);
        pack();                                        // put it all together
        setVisible(true);                              // make it show up
    }

    private class InitPanel extends JPanel {
        public InitPanel() {
            peopleLabel = new JLabel("Enter the number of sales people");
            add(peopleLabel);
            peopleField = new JTextField(5);
            add(peopleField);
            jbNumPeople = new JButton("Submit");
            add(jbNumPeople);
            jbNumPeople.addActionListener(new NumSalesPeopleListener());
        }
    }

    private class NumSalesPeopleListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            if (inputPanel != null)
            {
                remove(inputPanel);
                app = new SalesApp();
            }
            numPeople = Integer.parseInt(peopleField.getText());
            inputPanel = new InputPanel(app, numPeople, 2);
            add("Center", inputPanel);
            SalesUserInterface.this.validate();
        }
    }
}
```

Save it and Run **Main.java**. Enter the values, but do not click **All Set**. Select **Options | Results** from the menu.

Example 4

Be sure your constructors do *not* have a return type. If a method has a return type, then it is not a constructor. Constructors do not have return types; by default they return an instance of themselves.

In some cases, you may *think* you've called a constructor to create an instance of something, but you really haven't. In this next example, our code won't give us a **NullPointerException**, because it never creates an instance.

Edit **SalesUserInterface** as shown in **blue**:

CODE TO EDIT: SalesUserInterface

```
package salesGUI;

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

public class SalesUserInterface extends JFrame {
    SalesApp app;
    JMenuBar mb;
    JMenu m, m1;
    JMenuItem q, r, s, t;
    InputPanel inputPanel;
    JLabel peopleLabel;
    JTextField peopleField;
    JButton jbNumPeople, done;
    int numPeople;
    OutputPanel results;
    boolean processed = false;

    public SalesUserInterface(SalesApp myApp) {
        System.out.println("Did I get made?");
        app = new SalesApp();                          // who am I an interface fo
r?
        app.setMyUserInterface(this);
        setLayout(new BorderLayout());
        setPreferredSize(new Dimension(600, 600));
        mb = new JMenuBar();
        setJMenuBar(mb);
        m = new JMenu("File");
        m1 = new JMenu ("Options");
        mb.add(m);
        mb.add(m1);
        m.add(q = new JMenuItem("Exit"));
        q.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });

        m1.add(t=new JMenuItem("Results"));
        t.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                inputPanel.setAllInputs();  // added method call to make sure al
l is set
                if (processed)
                {
                    remove(results);
                }
                results = new OutputPanel(app);
                add("South", results);
                processed = true;
                results.writeOutput();
            }
        });

        InitPanel specifyNumber = new InitPanel();
        add("North", specifyNumber);
        pack();
        setVisible(true);
    }

    private class InitPanel extends JPanel {
        public InitPanel() {
            peopleLabel = new JLabel("Enter the number of sales people");
            add(peopleLabel);
            peopleField = new JTextField(5);
            add(peopleField);
            jbNumPeople = new JButton("Submit");
            add(jbNumPeople);
            jbNumPeople.addActionListener(new NumSalesPeopleListener());
        }
    }

    private class NumSalesPeopleListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            if (inputPanel != null)
            {
                remove(inputPanel);
                app = new SalesApp();
            }
            numPeople = Integer.parseInt(peopleField.getText());
            inputPanel = new InputPanel(app, numPeople, 2);
            add("Center", inputPanel);
            SalesUserInterface.this.validate();
        }
    }
}
```

Edit **Main** as shown in **blue**:

CODE TO EDIT: Main

```
package salesGUI;

public class Main {
    public static void main(String[] args) {
        SalesApp newApp = new SalesApp();
        SalesUserInterface appFrame = new SalesUserInterface();
        // so we can see if things are set as expected:
        System.out.println("I think I made it and am back");
        appFrame.app.setMyUserInterface(appFrame);
    }
}
```

Save and run the application from **Main**. Check the Console to make sure that both **println** comments appear. Make sure that the rest of the application works as expected. Now, give the **SalesUserInterface** class's *constructor* (found around line 21) a **void** return type. Add **void** to the **SalesUserInterface()** constructor as shown:

CODE TO EDIT: SalesUserInterface

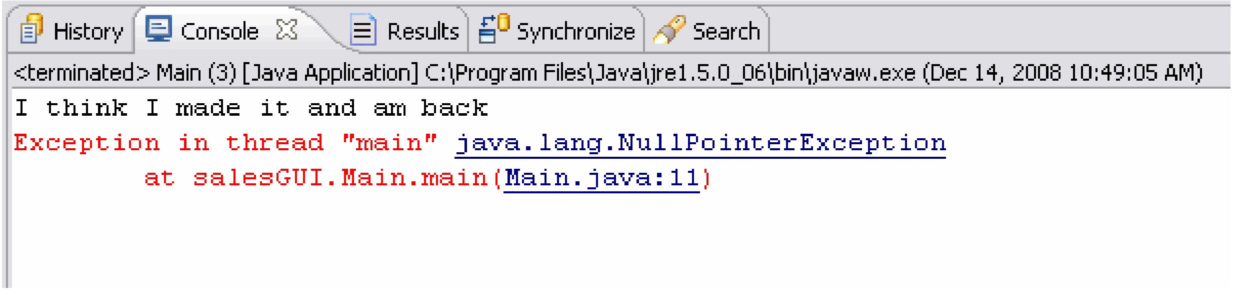
```
package salesGUI;

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

public class SalesUserInterface extends JFrame {
    SalesApp app;
    JMenuBar mb;
    JMenu m, m1;
    JMenuItem q, r, s, t;
    InputPanel inputPanel;
    JLabel peopleLabel;
    JTextField peopleField;
    JButton jbNumPeople, done;
    int numPeople;
    OutputPanel results;
    boolean processed = false;

    public void SalesUserInterface(SalesApp myApp){
        System.out.println("Did I get made?");
        app = new SalesApp();
        app.setMyUserInterface(this);
        setLayout(new BorderLayout());
        setPreferredSize(new Dimension(600, 600));
        mb = new JMenuBar();
        setJMenuBar(mb);
        m = new JMenu("File");
        m1 = new JMenu ("Options");
        mb.add(m);
        mb.add(m1);
        m.add(q = new JMenuItem("Exit"));
        q.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });

        m1.add(t=new JMenuItem("Results"));
        t.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                inputPanel.setAllInputs();  // added method call to make sure al
l is set
                if (processed)
                {
                    remove(results);
                }
                results = new OutputPanel(app);
                add("South", results);
                processed = true;
                results.writeOutput();
            }
        });

        InitPanel specifyNumber = new InitPanel();
        add("North", specifyNumber);
        pack();
        setVisible(true);
    }

    private class InitPanel extends JPanel {
        public InitPanel() {
            peopleLabel = new JLabel("Enter the number of sales people");
            add(peopleLabel);
            peopleField = new JTextField(5);
            add(peopleField);
            jbNumPeople = new JButton("Submit");
            add(jbNumPeople);
            jbNumPeople.addActionListener(new NumSalesPeopleListener());
        }
    }

    private class NumSalesPeopleListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            if (inputPanel != null)
            {
                remove(inputPanel);
                app = new SalesApp();
            }
            numPeople = Integer.parseInt(peopleField.getText());
            inputPanel = new InputPanel(app, numPeople, 2);
            add("Center", inputPanel);
            SalesUserInterface.this.validate();
        }
    }
}
```

Make this change to the **Main** as well:

CODE TO EDIT: Main

```
package salesGUI;

public class Main {

    public static void main(String[] args) {
        SalesApp newApp = new SalesApp();
        // Remove the passed newApp parameter
        SalesUserInterface appFrame = new SalesUserInterface();
        System.out.println("I think I made it and am back");
        appFrame.app.setMyUserInterface(appFrame);
    }
}
```

Save both classes, and run the **Main** class. Nothing opens and we see this in the console:



```
<terminated> Main (3) [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Dec 14, 2008 10:49:05 AM)
I think I made it and am back
Exception in thread "main" java.lang.NullPointerException
	at salesGUI.Main.main(Main.java:11)
```

We did not get the **System.out.println Did I get made?** from our **SalesUserInterface** constructor. And we didn't get a NullPointerException until the line *after* the instantiation in Main. The line of code intended to make an instance for **SalesUserInterface** ran, but nothing happened. Why?

Because **void** was given as a return type, the **SalesUserInterface** class didn't really have a constructor, so Java just let the class inherit the constructor from its **super**. The line **SalesUserInterface appFrame = new SalesUserInterface();** ran as expected, but its Constructor was only run from **JFrame**. As a result of inheritance, we did not get a NullPointerException--yet. We called access to the application in the **Main**, but it *should* have been called in the Constructor. Since Java never made the instance in the proper Constructor, most of the variables we thought were set weren't.

Note This is a difficult error to find, so make sure you never give a constructor declaration a return type.

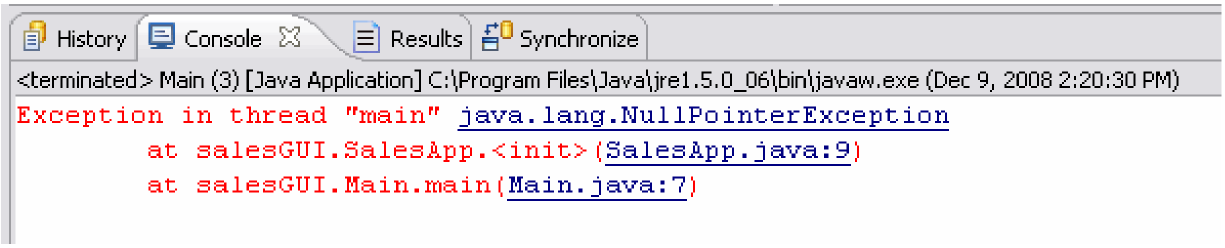
Example 5

Edit **SalesApp** as shown in **blue** and **red**:

CODE TO EDIT: SalesApp

```
package salesGUI;

public class SalesApp {
    private int [] sales;
    private int salesBar;
    private int totalSales;
    // Comment out the next line and add the following line.
    // private double average;
    private double average = totalSales/sales.length;
    private int minIndex = 0;
    private int maxIndex = 0;
    SalesUserInterface myUserInterface;

    public void setMyUserInterface(SalesUserInterface myGUI){
        myUserInterface = myGUI;
    }

    public void setSales(int[] sales) {
        this.sales = sales;
        for (int i = 0; i < sales.length; i++)
            System.out.println("sales [i] = " + sales[i]);
        setTotalSales();
    }

    public void setTotalSales() {
        totalSales = 0;
        for (int x = 0; x < sales.length; x++)
            totalSales += sales[x];
        setAverage();
    }

    public void setAverage() {
        if (sales.length != 0)
            average = (double)(totalSales / sales.length);
        System.out.println("totalSales is " + totalSales + " and sales.length is
 "
                + sales.length + ", making average "
                + ((double) totalSales / sales.length));
    }

    public void setSalesBar(int goal) {
        salesBar = goal;
    }

    public int[] getSales() {
        return sales;
    }

    public double getAverage() {
        if (sales.length != 0)
            return ((double) totalSales / sales.length);
        else
            return average;
    }

    public int getBar() {
        return salesBar;
    }

    public int getTotalSales() {
        return totalSales;
    }

    public int getMin() {
        return minIndex;
    }
    public int getMax() {
        return maxIndex;
    }

    public void calculateMinMax() {
        int minimum = sales[0];
        int maximum = sales[0];
        for (int x = 0; x < sales.length; x++) {
            if (sales[x] > maximum) {
                maximum = sales[x];
                maxIndex = x;
            }
            else if (sales[x] < minimum) {
                minimum = sales[x];
                minIndex = x;
            }
        }
        System.out.println("Maximum value is at index " + maxIndex
                + " (Salesperson " + (maxIndex + 1) + ") with value " + maximum)
;
        System.out.println("Minimum value is at index " + minIndex
                + " (Salesperson " + (minIndex + 1) + ") with value " + minimum)
;
        setAverage();
    }

    public int[] determineTopSalesPeople() {
        System.out.println("I'm here and salesBar is " + salesBar);
        int[] performance = new int [sales.length];
        for (int x = 0; x < sales.length; x++) {
            if (sales[x] > salesBar) {
                performance[x] = 1;
            }
            else if (sales[x] == salesBar) {
                  performance[x] = 0;
            }
            else {
                performance[x] = -1;
            }
        }
        return performance;
    }
}
```

Save it and run **Main**.



```
<terminated> Main (3) [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Dec 9, 2008 2:20:30 PM)
Exception in thread "main" java.lang.NullPointerException
        at salesGUI.SalesApp.<init>(SalesApp.java:9)
        at salesGUI.Main.main(Main.java:7)
```

Can you see why there's a null pointer? You don't have a **sales[]** array instantiated yet.

Division By Zero

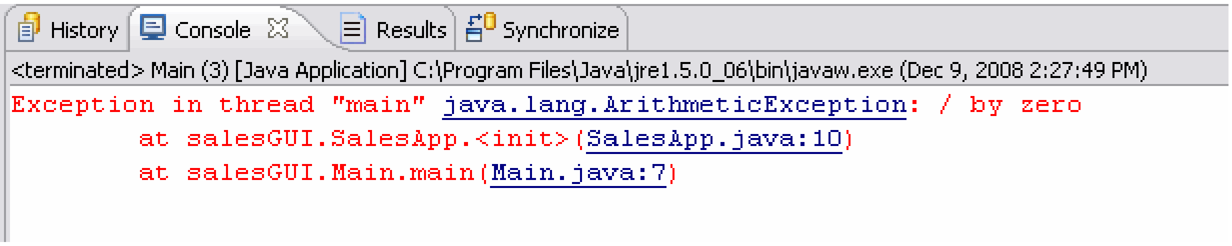
We'll keep working with the last example to explore another subclass (**java.lang.ArithmeticException**) of **RuntimeException**.

Edit **SalesApp** as shown in **blue** and **red**:

CODE TO EDIT: SalesApp

```
package salesGUI;

public class SalesApp {
    private int [] sales;
    private int salesBar;
    private int totalSales;
    // private double average;
    // Comment out or remove the next line:
    // private double average = totalSales/sales.length;
    private int numSalesPeople;
    private double average = totalSales/numSalesPeople;
    private int minIndex = 0;
    private int maxIndex = 0;
    SalesUserInterface myUserInterface;

    public void setMyUserInterface(SalesUserInterface myGUI){
        myUserInterface = myGUI;
    }

    public void setSales(int[] sales) {
        this.sales = sales;
        for (int i = 0; i < sales.length; i++)
            System.out.println("sales [i] = " + sales[i]);
        setTotalSales();
    }

    public void setTotalSales() {
        totalSales = 0;
        for (int x = 0; x < sales.length; x++)
            totalSales += sales[x];
        setAverage();
    }

    public void setAverage() {
        if (sales.length != 0)
            average = (double)(totalSales / sales.length);
        System.out.println("totalSales is " + totalSales + " and sales.length is
 "
                + sales.length + ", making average "
                + ((double) totalSales / sales.length));
    }

    public void setSalesBar(int goal) {
        salesBar = goal;
    }

    public int[] getSales() {
        return sales;
    }

    public double getAverage() {
        if (sales.length != 0)
            return ((double) totalSales / sales.length);
        else
            return average;
    }

    public int getBar() {
        return salesBar;
    }

    public int getTotalSales() {
        return totalSales;
    }

    public int getMin() {
        return minIndex;
    }

    public int getMax() {
        return maxIndex;
    }

    public void calculateMinMax() {
        int minimum = sales[0];
        int maximum = sales[0];
        for (int x = 0; x < sales.length; x++) {
            if (sales[x] > maximum) {
                maximum = sales[x];
                maxIndex = x;
            }
            else if (sales[x] < minimum) {
                minimum = sales[x];
                minIndex = x;
            }
        }
        System.out.println("Maximum value is at index " + maxIndex
                + " (Salesperson " + (maxIndex + 1) + ") with value " + maximum)
;
        System.out.println("Minimum value is at index " + minIndex
                + " (Salesperson " + (minIndex + 1) + ") with value " + minimum)
;
        setAverage();
    }

    public int[] determineTopSalesPeople() {
        System.out.println("I'm here and salesBar is " + salesBar);
        int[] performance = new int [sales.length];
        for (int x = 0; x < sales.length; x++) {
            if (sales[x] > salesBar) {
                performance[x] = 1;
            }
            else if (sales[x] == salesBar) {
                  performance[x] = 0;
            }
            else {
                performance[x] = -1;
            }
        }
        return performance;
    }
}
```

Save it and run **Main**.



```
<terminated> Main (3) [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Dec 9, 2008 2:27:49 PM)
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at salesGUI.SalesApp.<init>(SalesApp.java:10)
        at salesGUI.Main.main(Main.java:7)
```

It's difficult for the compiler to catch these kinds of exceptions before runtime. When the compiler scans the code for proper syntax, it doesn't know in advance whether a value for **numSalesPeople** has been set. The compiler won't know whether the value is 0 at the time of compilation.

Change **SalesApp** back, as shown in **blue**:

CODE TO EDIT: SalesApp

```
package salesGUI;

public class SalesApp {
    private int[] sales;
    private int salesBar;
    private int totalSales;
    private double average;
    private int minIndex=0;
    private int maxIndex=0;
    SalesUserInterface myUserInterface;

    public void setMyUserInterface(SalesUserInterface myGUI){
        myUserInterface = myGUI;
    }

    public void setSales(int[] sales) {
        this.sales = sales;
        for (int i = 0; i < sales.length; i++)
            System.out.println("sales [i] = " + sales[i]);
        setTotalSales();
    }

    public void setTotalSales() {
        totalSales = 0;
        for (int x = 0; x < sales.length; x++)
            totalSales += sales[x];
        setAverage();
    }

    public void setAverage() {
        if (sales.length != 0)
            average = (double)(totalSales / sales.length);
        System.out.println("totalSales is " + totalSales + " and sales.length is
"
                + sales.length + ", making average "
                + ((double) totalSales / sales.length));
    }

    public void setSalesBar(int goal) {
        salesBar = goal;
    }

    public int[] getSales() {
        return sales;
    }

    public double getAverage() {
        if (sales.length != 0)
            return ((double) totalSales / sales.length);
        else
            return average;
    }

    public int getBar() {
        return salesBar;
    }

    public int getTotalSales() {
        return totalSales;
    }

    public int getMin() {
        return minIndex;
    }

    public int getMax() {
        return maxIndex;
    }

    public void calculateMinMax() {
        int minimum = sales[0];
        int maximum = sales[0];
        for (int x = 0; x < sales.length; x++) {
            if (sales[x] > maximum) {
                maximum = sales[x];
                maxIndex = x;
            }
            else if (sales[x] < minimum) {
                minimum = sales[x];
                minIndex = x;
            }
        }
        System.out.println("Maximum value is at index " + maxIndex
                + " (Salesperson " + (maxIndex + 1) + ") with value " + maximum)
;
        System.out.println("Minimum value is at index " + minIndex
                + " (Salesperson " + (minIndex + 1) + ") with value " + minimum)
;
        setAverage();
    }

    public int[] determineTopSalesPeople() {
        System.out.println("I'm here and salesBar is " + salesBar);
        int[] performance = new int [sales.length];
        for (int x = 0; x < sales.length; x++) {
            if (sales[x] > salesBar) {
                performance[x] = 1;
            }
            else if (sales[x] == salesBar) {
                  performance[x] = 0;
            }
            else {
                performance[x] = -1;
            }
        }
        return performance;
    }
}
```

Change **Main** back, as shown in **blue**:

Code to Edit: Main

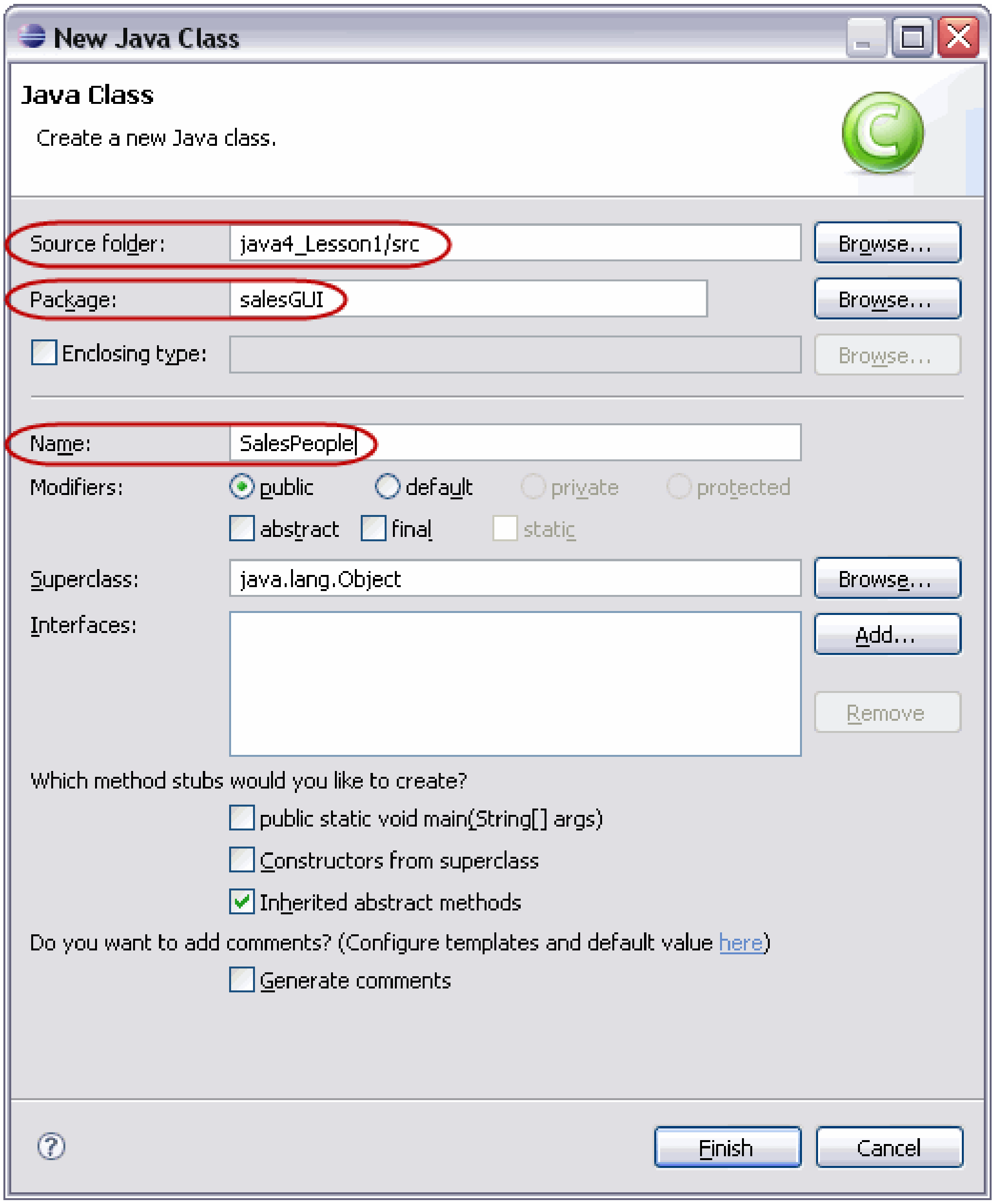
```
package salesGUI;

public class Main {
    public static void main(String[] args) {

        SalesApp newApp = new SalesApp();
        SalesUserInterface appFrame = new SalesUserInterface(newApp);
        System.out.println("I think I made it and am back");
        appFrame.app.setMyUserInterface(appFrame);
    }
}
```

Save and run it to make sure it works as before.

Array Out of Bounds

Let's check out another subclass (**java.lang.ArrayIndexOutOfBoundsException**) of **RuntimeException** that can cause runtime exceptions.

You can go out of bounds pretty easily using **for** loops. Take a look. Create a new class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson1/src

Package: salesGUI

□ Enclosing type:

Name: SalesPeople

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

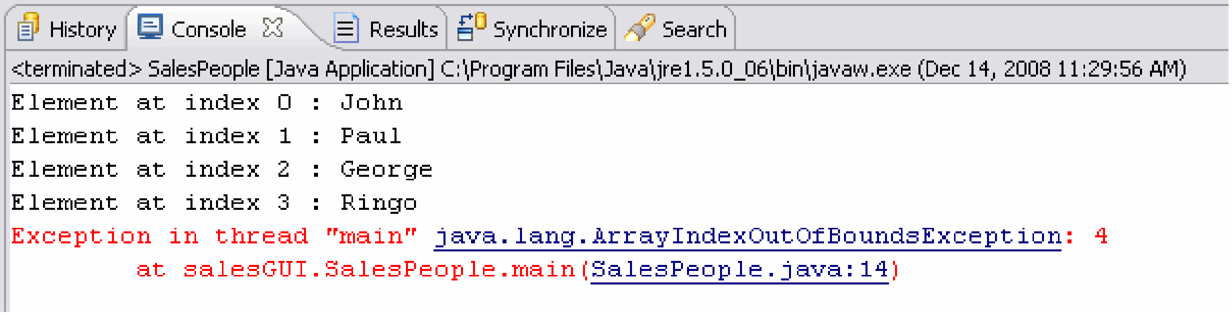
□ Generate comments

Type **SalesPeople** as shown in **blue**:

CODE TO TYPE: SalesPeople

```
package salesGUI;

public class SalesPeople {
    public static void main(String[] args) {
        String[] salesPeople;
        salesPeople = new String[4];

        salesPeople[0] = "John";
        salesPeople[1] = "Paul";
        salesPeople[2] = "George";
        salesPeople[3] = "Ringo";

        for (int i=0;  i <= salesPeople.length; i++)
            System.out.println("Element at index " + i + " : " + salesPeople[i])
;
        System.out.println("Size of the salesPeople array is " + salesPeople.len
gth);
    }
}
```

Save and run it.



```
<terminated> SalesPeople [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Dec 14, 2008 11:29:56 AM)
Element at index 0 : John
Element at index 1 : Paul
Element at index 2 : George
Element at index 3 : Ringo
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
	at salesGUI.SalesPeople.main(SalesPeople.java:14)
```

The attribute **length** of an array is equal to the number of elements in the array, but the *indices start at 0*, so the last index is **length - 1**. If we try to loop to the value of **sales[length]**, we will get a **java.lang.ArrayIndexOutOfBoundsException**.

Array Out of Bounds Example 2

Look at **Main.java** in our java4_Lesson1 project under the **sales1** package:

OBSERVE: Main

```
package sales1;

public class Main {

    public static void main(String[] args){
        if (args.length > 0)
        {
            int argIn = Integer.parseInt(args[0]);              // user inputs ar
gument at command line so use it
            SalesReport mySalesInfo = new SalesReport(argIn);   // pass input as
parameter to constructor
            mySalesInfo.testMe();                               // start the appl
ication
        }
        else
        {                                                       // no input from
user so ask in constructor
            SalesReport mySalesInfo = new SalesReport();        // instantiate th
e class with constructor with no parameters - will prompt for input
            mySalesInfo.testMe();                               // start the appl
ication
        }
    }
}
```

Because we have included **if (args.length > 0)**, the user can enter arguments either at the command line or via a GUI prompt. In an earlier version of this **Main.java** class, we tried to determine whether the user had entered an argument at the command line by using the conditional statement: **if (args[0] != null)**. This conditional statement actually looks for **args[0]**, but might not find an **args** array at all. In that case, **args[0]** would already be out of bounds. Let's try it again. Edit the conditional statement as shown in **blue**:

CODE TO EDIT: Main

```
package sales1;

public class Main {

    public static void main(String[] args){
        if (args[0] != null)
        {
            int argIn = Integer.parseInt(args[0]);
            SalesReport mySalesInfo = new SalesReport(argIn);
            mySalesInfo.testMe();
        }
        else
        {
            SalesReport mySalesInfo = new SalesReport();
            mySalesInfo.testMe();
        }
    }
}
```

Save and run it. You'll see a message in the Console: **Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0**.

It is generally time-consuming and inefficient for Java to use **try** and **catch** blocks when an appropriate **if** statement works well enough. But let's try it anyway just to illustrate the idea of catching an **ArrayIndexOutOfBoundsException**. Edit the **Main** in sales1 as shown in **blue** below:

CODE TO EDIT: Main

```
package sales1;

public class Main {

    public static void main(String[] args){

        try
        {
            int argIn = Integer.parseInt(args[0]);             // if no args[0],
 will throw ArrayIndexOutOfBoundsException
            SalesReport mySalesInfo = new SalesReport(argIn);
            mySalesInfo.testMe();
        }
        catch (ArrayIndexOutOfBoundsException exception)
        {
            SalesReport mySalesInfo = new SalesReport();
            mySalesInfo.testMe();
        }
    }
}
```

Save and run it.

Tip Because try/catch clauses are more labor intensive (for Java and for you) than conditional statements, exception handling should be just that: an *exception* to the norm.

Errors

Errors are conditions that happen *outside* of the application; for example, **OutOfMemoryError**. They are usually the result of a major programming mistake. For example, a recursive program (a program with a method that calls itself) might not have a stop statement, which means an infinite loop would be created. This could then generate a **StackOverflowError**.

Try this code:

CODE TO TYPE: InfiniteLoop

```
public class InfiniteLoop {

    public int tryMe(int x){
        // System.out.println(x); // Run first without this line--then uncomment just t
o see how many times it ran before having the overflow
        return tryMe(x+1);
    }

    public static void main(String [] args){
        InfiniteLoop silly = new InfiniteLoop();
        silly.tryMe(1);
    }
}
```

Save and run it. This is what you *don't* want your customers to see!

Unfortunately, **Error**s can also occur when we do distributed computing--that is, when we go outside of the environment of our own machine. When a dynamic linking failure or other hard failure (that is, a failure that needs to be fixed by a programmer) occurs in the Java virtual machine, the virtual machine throws an Error. According to the Oracle Tutorial's section on **Error**s in The Catch or Specify Requirement:

"The second kind of exception is the *error*. These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. For example, suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction. The unsuccessful read will throw java.io.IOError. An application might choose to catch this exception, in order to notify the user of the problem — but it also might make sense for the program to print a stack trace and exit.

Errors *are not subject* to the Catch or Specify Requirement. Errors are those exceptions indicated by **Error** and its subclasses."

We've already seen that the API is a valuable source of Interface, Class, and Exception information; now we'll see how it helps us tackle **Error**s.

API Go to the **java.lang** package in the API. Scroll down to the **Error Summary**. Read about a few of them. Both **Exception**s and **Error**s inherit from the class **Throwable**...interesting.

Simple programs typically do not catch or throw Errors, but checked exceptions *are* subject to the **Catch or Specify Requirement**. We'll discuss that requirement in detail in the next lesson.

Programming Responsibly

In this lesson, we've seen examples of subclasses of runtime exceptions and how to prevent them by careful coding. Since programmers share code with one another so often, that's pretty important.

Although Java requires that methods catch or specify checked exceptions, methods that we write do not have to catch or specify unchecked exceptions (such as runtime exceptions). And because catching or specifying an exception requires more work, programmers are occasionally tempted to write code that throws only runtime exceptions and therefore doesn't have to catch or specify them. This is *exception abuse*, and is not recommended. For more information, see the Oracle Tutorial Unchecked Exceptions - The Controversy.

The more you check your code to prevent unchecked exceptions from occurring at runtime, the better for all concerned. Don't make Duke angry.

Checked Exceptions: Catching Problems

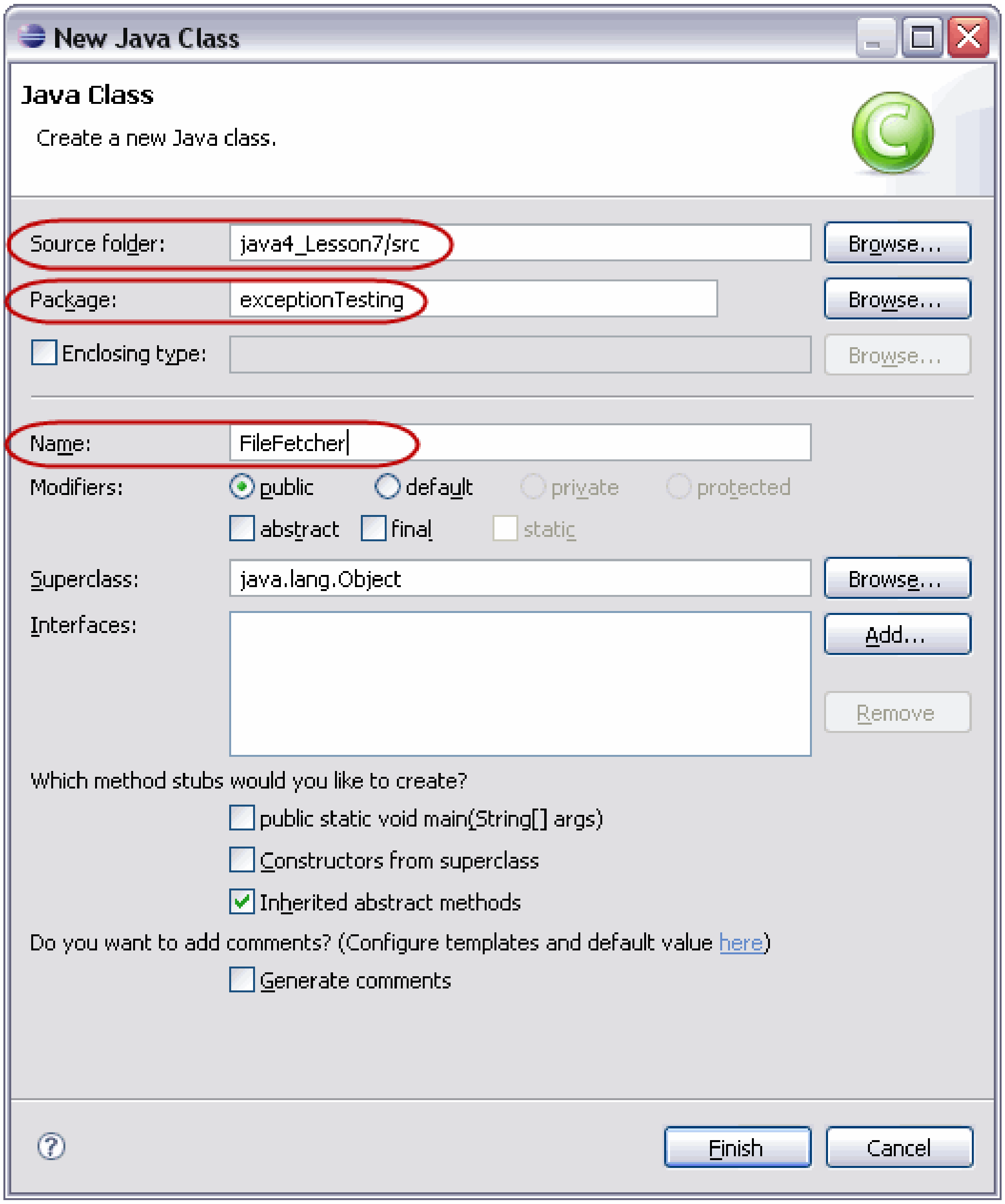
Degrading Gracefully

In this lesson, we'll focus on *checked exceptions*. A well-written application will anticipate and provide mechanisms for recovery from these exceptional conditions. We want you to understand exactly what's happening when an **Exception** is thrown. We'll illustrate that process so you'll be confident *using*, and ultimately defining your own checked exceptions.

I/O Exceptions

Other than **RuntimeException** and its subclasses, the most common exceptions occur when attempting to access files that are either nonexistent or inaccessible. The next course will discuss Java's classes for input and output (I/O) in more explciit detail, but for now, we'll use an less complicated example that reads a file to demonstrate the use of checked exceptions in the **java.io** package.

Create a new **java4_Lesson7** project. If you're given the option to **Open Associated Perspective**, click **No**. In this new project, create a **FileFetcher** class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson7/src

Package: exceptionTesting

□ Enclosing type:

Name: FileFetcher

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

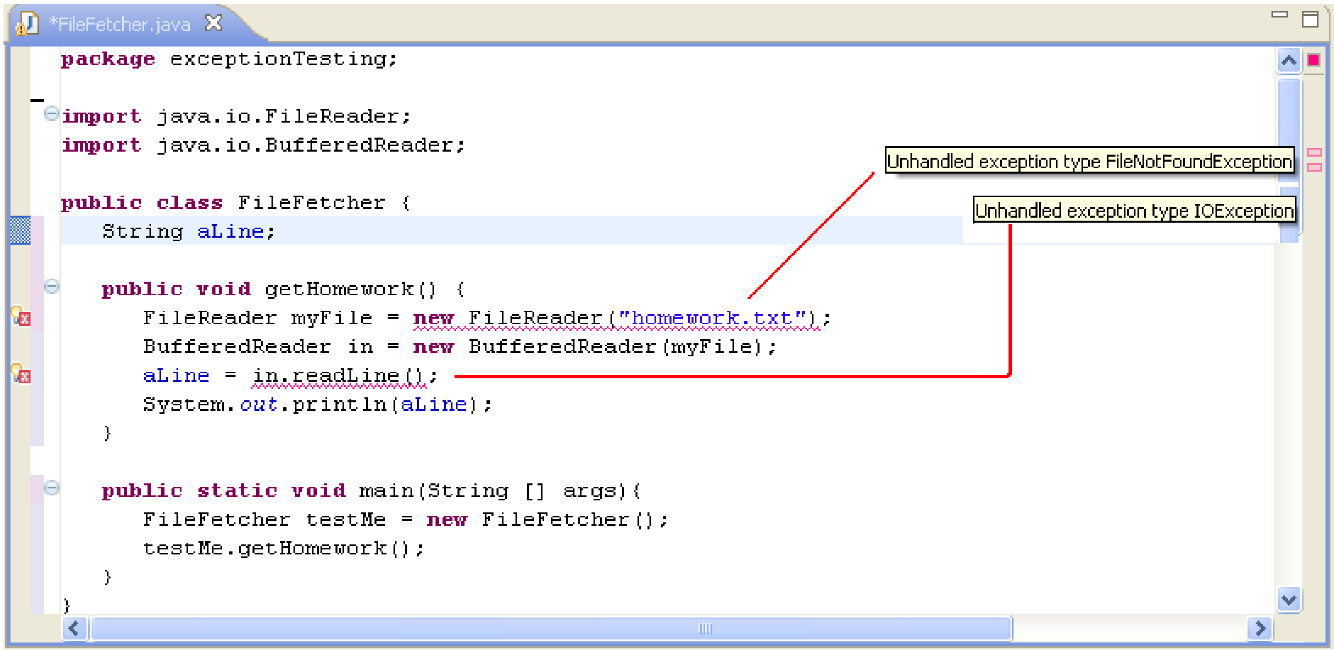
□ Generate comments

Type **FileFetcher** as shown in **blue**:

CODE TO TYPE: FileFetcher

```
package exceptionTesting;

import java.io.FileReader;
import java.io.BufferedReader;

public class FileFetcher {
    String aLine="";                    // we will look at the file one line at a time

    public void getHomework() {
        FileReader myFile = new FileReader("homework.txt");  // create a Reader for a f
ile--we will define this file next
        BufferedReader in = new BufferedReader(myFile);      // wrap the FileReader in
a class that lets us manipulate it
        aLine = in.readLine();                               // read in a line of the f
ile
        System.out.println(aLine);                           // print it to the Console
    }

    public static void main(String [] args){
        FileFetcher testMe = new FileFetcher();
        testMe.getHomework();
    }
}
```

We see two sources of errors:



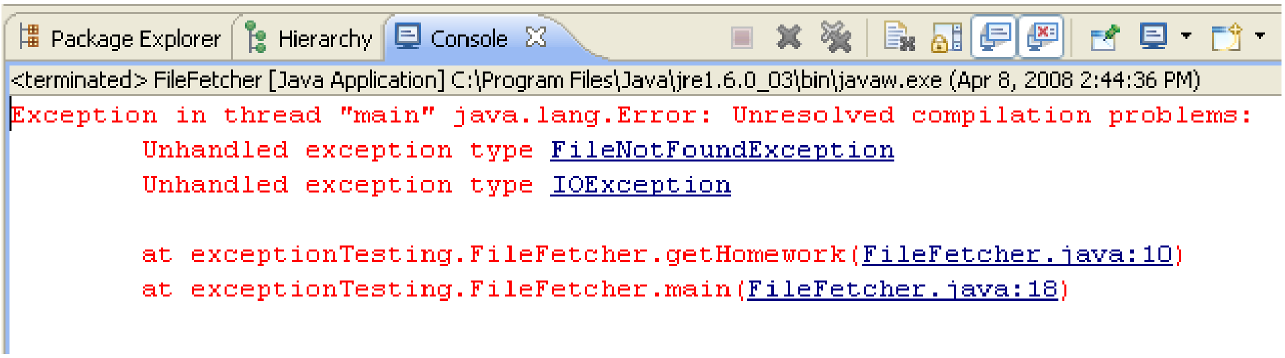
```
package exceptionTesting;

import java.io.FileReader;
import java.io.BufferedReader;

public class FileFetcher {
    String aLine;

    public void getHomework() {
        FileReader myFile = new FileReader("homework.txt");
        BufferedReader in = new BufferedReader(myFile);
        aLine = in.readLine();
        System.out.println(aLine);
    }

    public static void main(String [] args){
        FileFetcher testMe = new FileFetcher();
        testMe.getHomework();
    }
}
```

Unhandled exception type FileNotFoundException

Unhandled exception type IOException

Save and run it anyway. Even though Java complains about Errors, click **Proceed**.

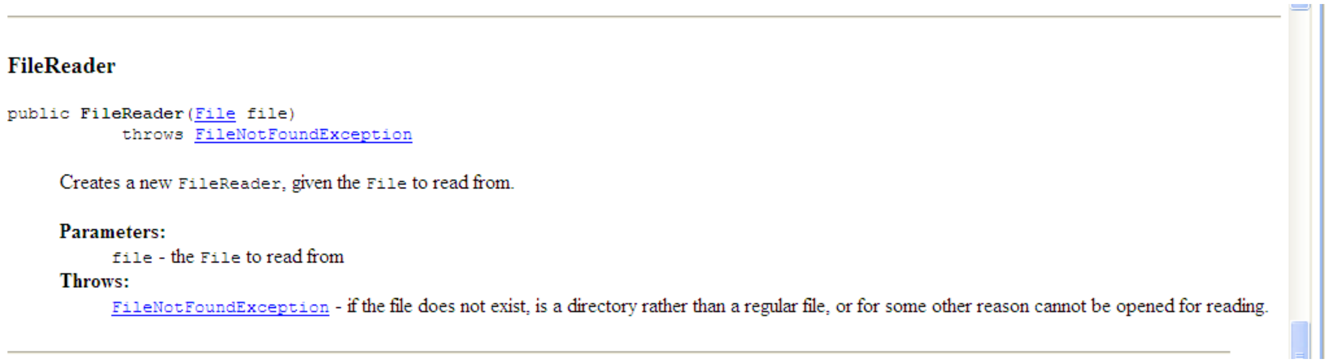
We have **Unresolved compilation problems**. (Also, since our program could not compile, it threw a **java.lang.Error**):



```
<terminated> FileFetcher [Java Application] C:\Program Files\Java\jre1.6.0_03\bin\javaw.exe (Apr 8, 2008 2:44:36 PM)
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
	Unhandled exception type FileNotFoundException
	Unhandled exception type IOException

	at exceptionTesting.FileFetcher.getHomework(FileFetcher.java:10)
	at exceptionTesting.FileFetcher.main(FileFetcher.java:18)
```

On the first line of code with an error, we invoke the **FileReader** constructor, so let's go to that part of the API to find out more.

API Go to the **java.io** package. Scroll down to the **FileReader** class in the **Class Summary**. Look at its constructor **FileReader(File file)**:



FileReader

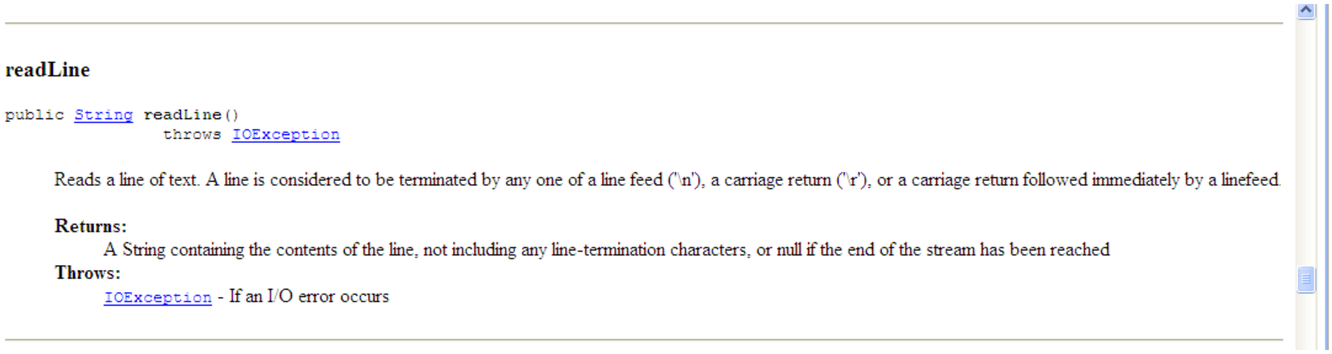
```
public FileReader(File file)
           throws FileNotFoundException
```

Creates a new `FileReader`, given the `File` to read from.

Parameters:
`file` - the `File` to read from
Throws:
`FileNotFoundException` - if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

In the Editor, on the next line with an error, we are calling the **in.readLine()** method. **in** is an instance of **BufferedReader**, so let's look at its **readLine()** method.

API Go back to the **java.io** package. Scroll down to the **BufferedReader** class in the **Class Summary**. Look at its **readLine()** method. Sure enough, there's our exception:



readLine

```
public String readLine()
                throws IOException
```

Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

Returns:
A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached
Throws:
`IOException` - If an I/O error occurs

Exception Types

We have seen various types of exceptions. Not every **catch** clause will work for every exception that's thrown.

Note An exception handler is considered appropriate if the type of the exception object thrown matches the type that it can handle.

Using Try and Catch

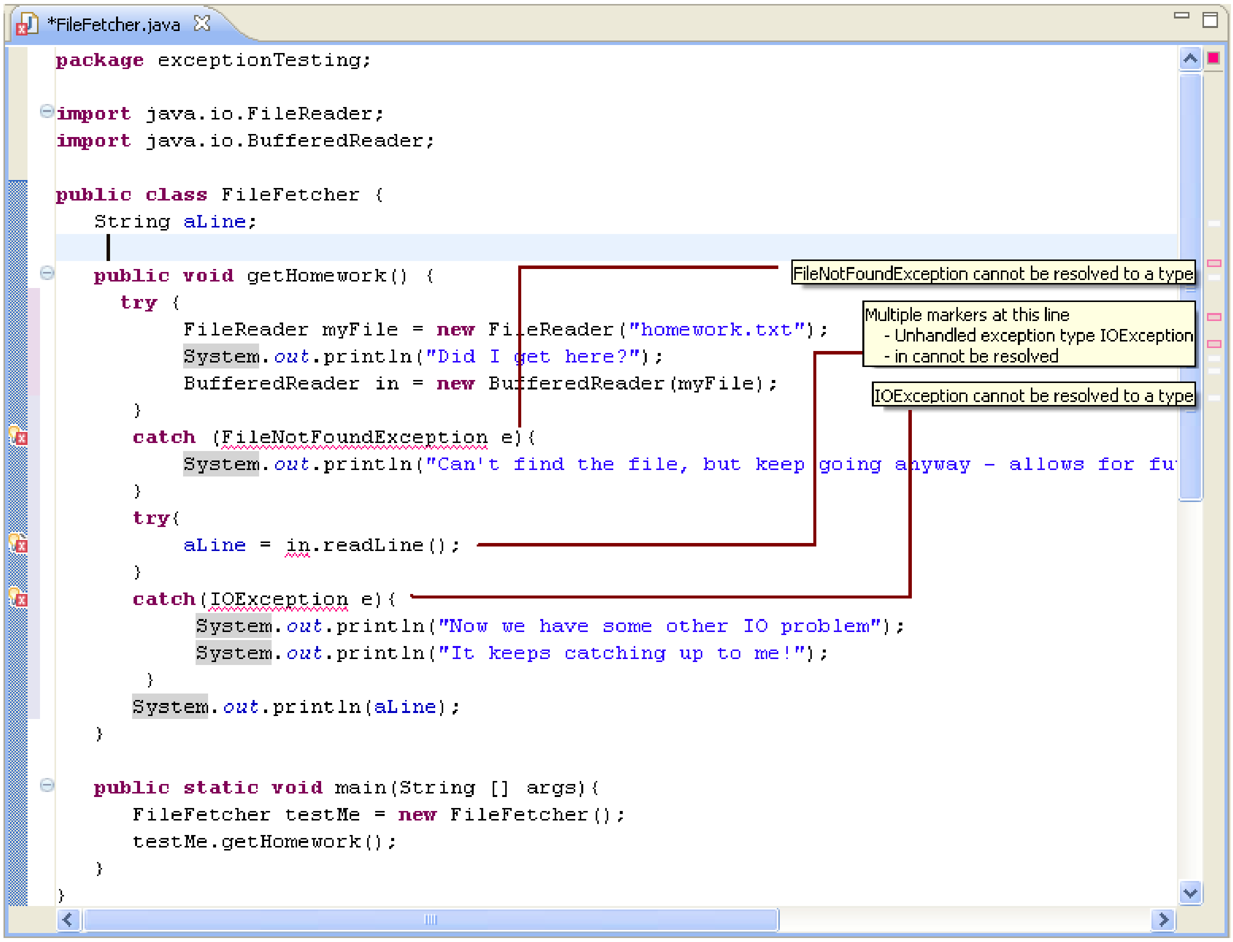
Since these are *checked exceptions*, Java will not let us compile the code to run it until we handle them. In this example, we aren't really handling the situation (in that we are not addressing the "big picture" by, for example, finding replacement files), but we are *handling* the individual exceptions the code throws, by providing **try/catch** clauses for them. In a *real* application, you would do something more in these **catch** blocks to recover from the exception in a more meaningful way.

Edit **FileFetcher** as shown in **blue** below:

CODE TO EDIT: FileFetcher

```
package exceptionTesting;

import java.io.FileReader;
import java.io.BufferedReader;

public class FileFetcher {
    String aLine="";        // we will look at the file one line at a time

    public void getHomework() {
        try
        {
            FileReader myFile = new FileReader("homework.txt");  // create a Rea
der for a file - we will define this file next
            System.out.println("I did get here");
            in = new BufferedReader(myFile);                     // wrap the Fil
eReader in a class that lets us manipulate it
        }
        catch (FileNotFoundException e)
        {
            System.out.println("Can't find the file, but keep going anyway--allo
ws for future problems!");
        }
        try
        {
            aLine = in.readLine();                               // read a line
of the file
        }
        catch(IOException e){
            System.out.println("Now we have some other problem!");
        }
        System.out.println(aLine);                               // print it to
the Console
    }

    public static void main(String [] args){
        FileFetcher testMe = new FileFetcher();
        testMe.getHomework();
    }
}
```

That didn't help much:



```
package exceptionTesting;

import java.io.FileReader;
import java.io.BufferedReader;

public class FileFetcher {
   String aLine;

   public void getHomework() {
     try {
          FileReader myFile = new FileReader("homework.txt");
          System.out.println("Did I get here?");
          BufferedReader in = new BufferedReader(myFile);
      }
      catch (FileNotFoundException e){
          System.out.println("Can't find the file, but keep going anyway - allows for fu
      }
      try{
          aLine = in.readLine();
      }
      catch(IOException e){
           System.out.println("Now we have some other IO problem");
           System.out.println("It keeps catching up to me!");
       }
      System.out.println(aLine);
   }

   public static void main(String [] args){
      FileFetcher testMe = new FileFetcher();
      testMe.getHomework();
   }
}
```

FileNotFoundException cannot be resolved to a type

Multiple markers at this line
- Unhandled exception type IOException
- in cannot be resolved

IOException cannot be resolved to a type

We can fix these. The unresolved issues can be resolved by importing appropriate classes (**java.io.FileNotFoundException** and **java.io.IOException**). The variable **in** cannot be resolved due to a **scope** issue. It is declared in one block of code, a **try** clause, and then used in another **try** clause.

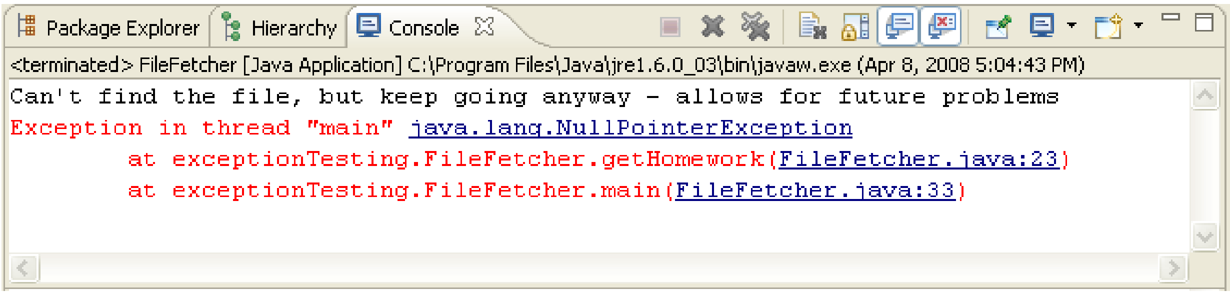
Edit **FileFetcher**. Add the imports and declare the instance of **FileReader** and **BufferedReader** so they can be seen by all methods:

CODE TO EDIT: FileFetcher

```
package exceptionTesting;

import java.io.FileReader;         // could also import java.io.*; to get all of
these
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;

public class FileFetcher {
    String aLine="";
    FileReader myFile;      // declare FileReader and BufferedReader as instance
 variables
    BufferedReader in;


    public void getHomework() {
        try
        {
            myFile = new FileReader("homework.txt");  // Do NOT declare here too
 or scope stays within try block
            System.out.println("I did get here");
            in = new BufferedReader(myFile);         // Do NOT declare here too
or scope stays within try block
        }
        catch (FileNotFoundException e)
        {
            System.out.println("Can't find the file, but keep going anyway - all
ows for future problems!");
        }
        try
        {
            aLine = in.readLine();                          // read a line of th
e file
        }
        catch(IOException e){
            System.out.println("Now we have some other problem!");
        }
        System.out.println(aLine);                     // print it to the Console
    }

    public static void main(String [] args){
        FileFetcher testMe = new FileFetcher();
        testMe.getHomework();
    }
}
```

All of the errors are gone now.

Save and run it.

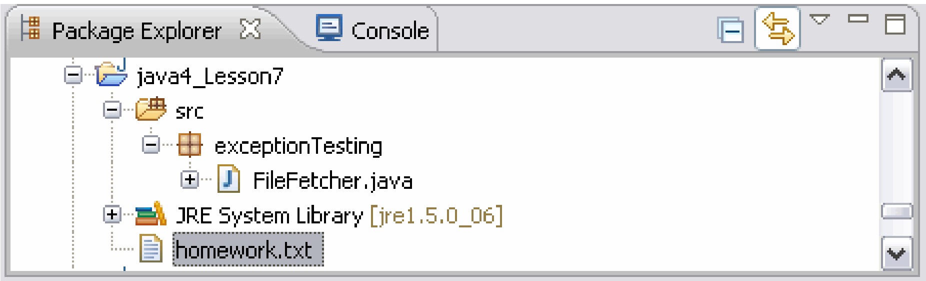


<terminated> FileFetcher [Java Application] C:\Program Files\Java\jre1.6.0_03\bin\javaw.exe (Apr 8, 2008 5:04:43 PM)

```
Can't find the file, but keep going anyway - allows for future problems
Exception in thread "main" java.lang.NullPointerException
        at exceptionTesting.FileFetcher.getHomework(FileFetcher.java:23)
        at exceptionTesting.FileFetcher.main(FileFetcher.java:33)
```

Of course, we can't find a file--we haven't made one yet. Let's make one now. This time we're making a new **File**, not a Java Class. Right-click on the **java4_Lesson7** Project folder. Select **New | File**. Enter the name **homework.txt** and click **Finish**.

Now the file structure for your java4_Lesson7 looks like this:

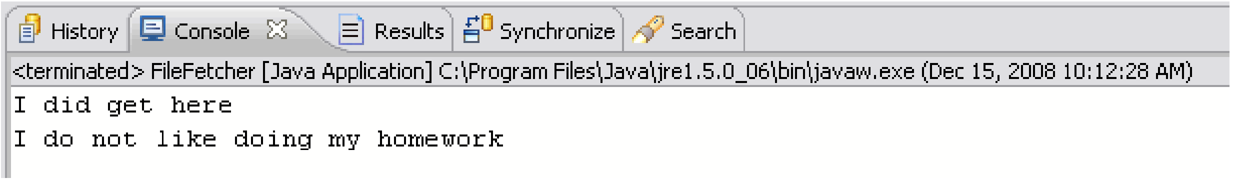


Type **homework.txt** as shown:

CODE TO TYPE: homework.txt

```
I do not like doing my homework
so I will try to make my parents do it.

I will tell them that I am sick and see
if I can get away with it.
```

Save it and run the **FileFetcher** class. Your code has this in the Console now:



<terminated> FileFetcher [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Dec 15, 2008 10:12:28 AM)

```
I did get here
I do not like doing my homework
```

Forwarding the Exception

We've already used Java methods that throw and handle exceptions . Now we'll forward the exceptions for other methods to handle.

To see the entire text file in your console output, edit **FileFetcher** as shown:

CODE TO EDIT: FileFetcher

```
package exceptionTesting;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.FileReader;
import java.io.BufferedReader;

public class FileFetcher {
    FileReader myFile;
    BufferedReader in;
    String aLine ="";

    public void getHomework() {
        try
        {
            myFile = new FileReader("homework.txt");  // create a Reader for a file
            System.out.println("I did get here");
            in = new BufferedReader(myFile);        // wrap the FileReader in a class that lets us manipulate it
        }
        catch (FileNotFoundException e){
            System.out.println("Can't find the file, but keep going anyway--allows for future problems!");
        }

        while (aLine != null){
            try {
                aLine = in.readLine();
            }
            catch(IOException e){
                System.out.println("Now we have some other I/O problem");
            }
            if (aLine !=null) System.out.println(aLine);  // we had another readLine after the check for null
        }
        // later, we will do something more here
    }

    public static void main(String [] args){
        FileFetcher testMe = new FileFetcher();
        testMe.getHomework();
    }
}
```

Save and run it.

Determining What To Do With Exceptions

Throwing Exceptions

Java methods may throw exceptions at *runtime*. Sometimes these errors can be avoided through well-written code, but others are unavoidable because we can't always anticipate what a user will do until runtime. Similarly, in handling *checked* exceptions, a method may know that it has an exceptional condition, but the not know how to handle it. The method would need to **throw** the exception, so the user of the method could specify how to handle the problem given the current environment.

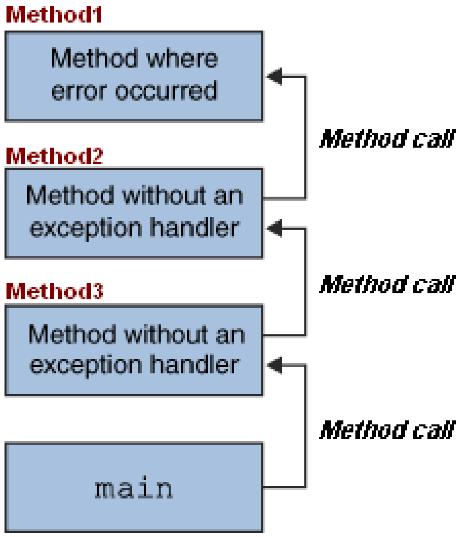
All code in an object-oriented program (like Java) is performed through the use of methods, and methods that call other methods. So exceptions will always be thrown in (or more accurately, *by*) methods *to* some other method that invoked it.

In fact, this is precisely the reason it is throwing the exception. If a method (say **method1()**) can handle an exceptional condition, it should. Only when a method doesn't *know* what to do with an exception, does it need to throw it to the method (say **method2()**) that is *using* **method1()** with the potentially exceptional condition.

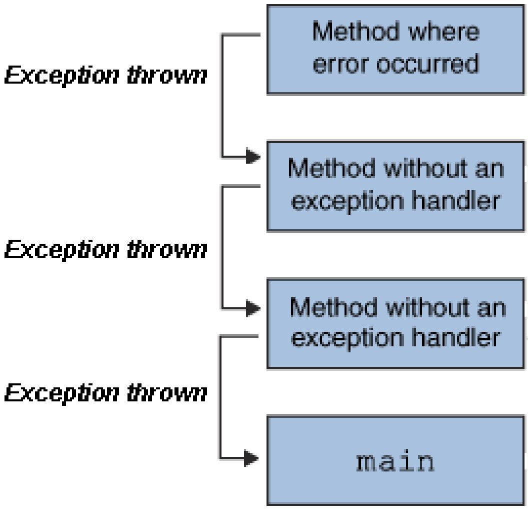
Keep in mind that if **method1()** is throwing, then **method2()** needs to **catch**--assuming **method2** is in an environment where it knows what to do.

If **method2()** is *not* in such an environment, then it needs to specify that *it* will throw (forward) the exception as well. Then the method that called *it* (say **method3()**) would need to **catch** it.

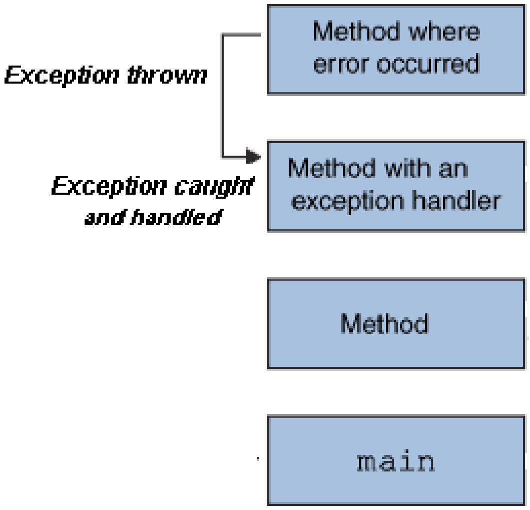
This is the **call stack**:


Method1
Method where error occurred
Method2
Method without an exception handler
Method3
Method without an exception handler
main
Method call
Method call
Method call

The throws could be passed back down the stack like this:


Method where error occurred
Method without an exception handler
Method without an exception handler
main
Exception thrown
Exception thrown
Exception thrown

...and still be caught:


Exception thrown
Exception caught and handled
Method where error occurred
Method with an exception handler
Method
main

The Java runtime system searches the call stack for a method that contains a block of code that can handle the exception.

Example

We will let our **FileFetcher** class represent a student who is supposed to be writing a file named *homework.txt*. Our student wants to convince his parents that he has too much to do and cannot do his homework this time around; he is not going to "handle" the homework situation. He is **not** going to handle the exceptions for retrieving his homework file in the **FileFetcher** method of **getHomework()**. Here are the conditions that represent Method2 for us:

1. The Method(s) where the error occurred throw the exceptions. Constructor **FileReader("homework.txt")** and **readLine()** will be Method1(a) and Method1(b), respectively.
2. The **FileFetcher** method of **getHomework()** is Method2: Method without an exception handler.

So, after all that work we did to handle the problem within the method itself, now our student says he is going to forward the exceptions. We'll see.

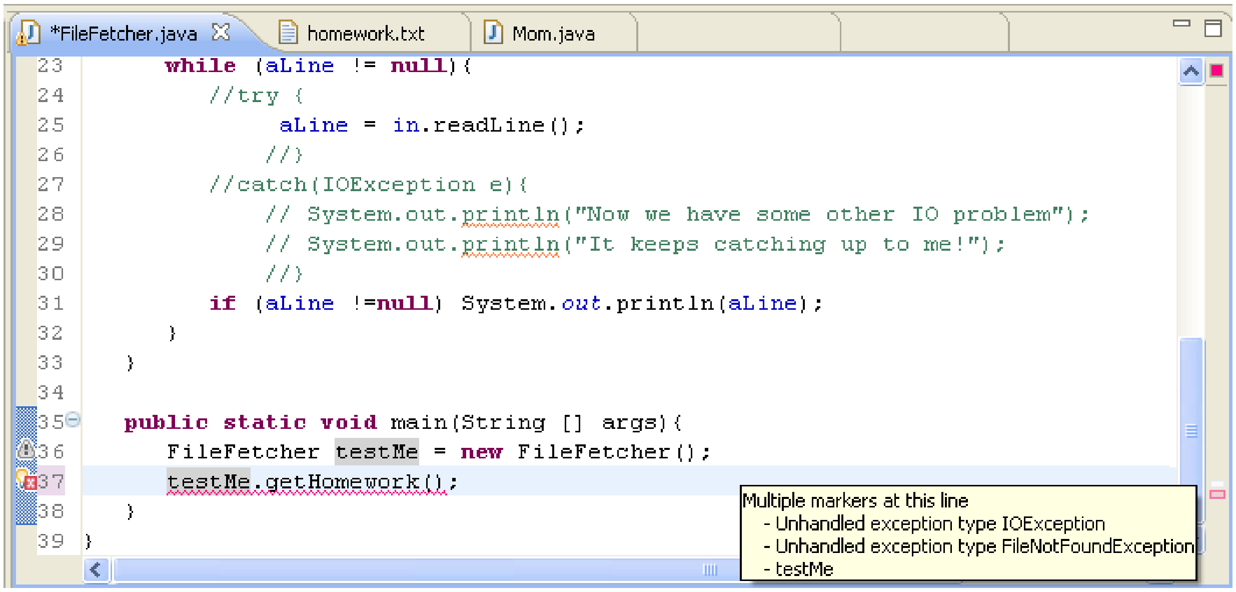
Edit **FileFetcher** as shown in **blue** and **red**:

CODE TO EDIT: FileFetcher

```
package exceptionTesting;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.FileReader;
import java.io.BufferedReader;

public class FileFetcher {
    FileReader myFile;
    BufferedReader in;
    String aLine ="";

    public void getHomework() throws FileNotFoundException, IOException {

        // try {
            myFile = new FileReader("homework.txt");
            System.out.println("I did get here");
            in = new BufferedReader(myFile);
        // }
        // catch (FileNotFoundException e){
        //     System.out.println("Can't find the file, but keep going anyway -
allows for future problems");
        // }
        while (aLine != null){
            // try {
                aLine = in.readLine();
            // }
            // catch(IOException e){
                // System.out.println("Now we have some other I/O problem");
            // }
            if (aLine !=null) System.out.println(aLine);             // we had a
nother readLine after the check for null
        }
    }

    public static void main(String [] args){
        FileFetcher testMe = new FileFetcher();
        testMe.getHomework();
    }
}
```

The only active pieces of code in the **getHomework()** method now are the FileReader instance, the **println** statements, and the while loop for reading and printing lines from the homework file.

We might as well comment out the **main()** method too. As you can see by the error message, if the **getHomework()** method doesn't handle the exceptions, then the instance **testMe** cannot call the method, unless *it* handles them.



```
        while (aLine != null){
           //try {
                aLine = in.readLine();
              //}
           //catch(IOException e){
              // System.out.println("Now we have some other IO problem");
              // System.out.println("It keeps catching up to me!");
              //}
           if (aLine !=null) System.out.println(aLine);
        }
    }

    public static void main(String [] args){
        FileFetcher testMe = new FileFetcher();
        testMe.getHomework();
    }
}
```

Multiple markers at this line
- Unhandled exception type IOException
- Unhandled exception type FileNotFoundException
- testMe

It looks like our irresponsible student isn't going to handle anything. Instead he's letting his Mom do it. Comment out the entire **main** method here so there are no errors:

CODE TO EDIT: FileFetcher

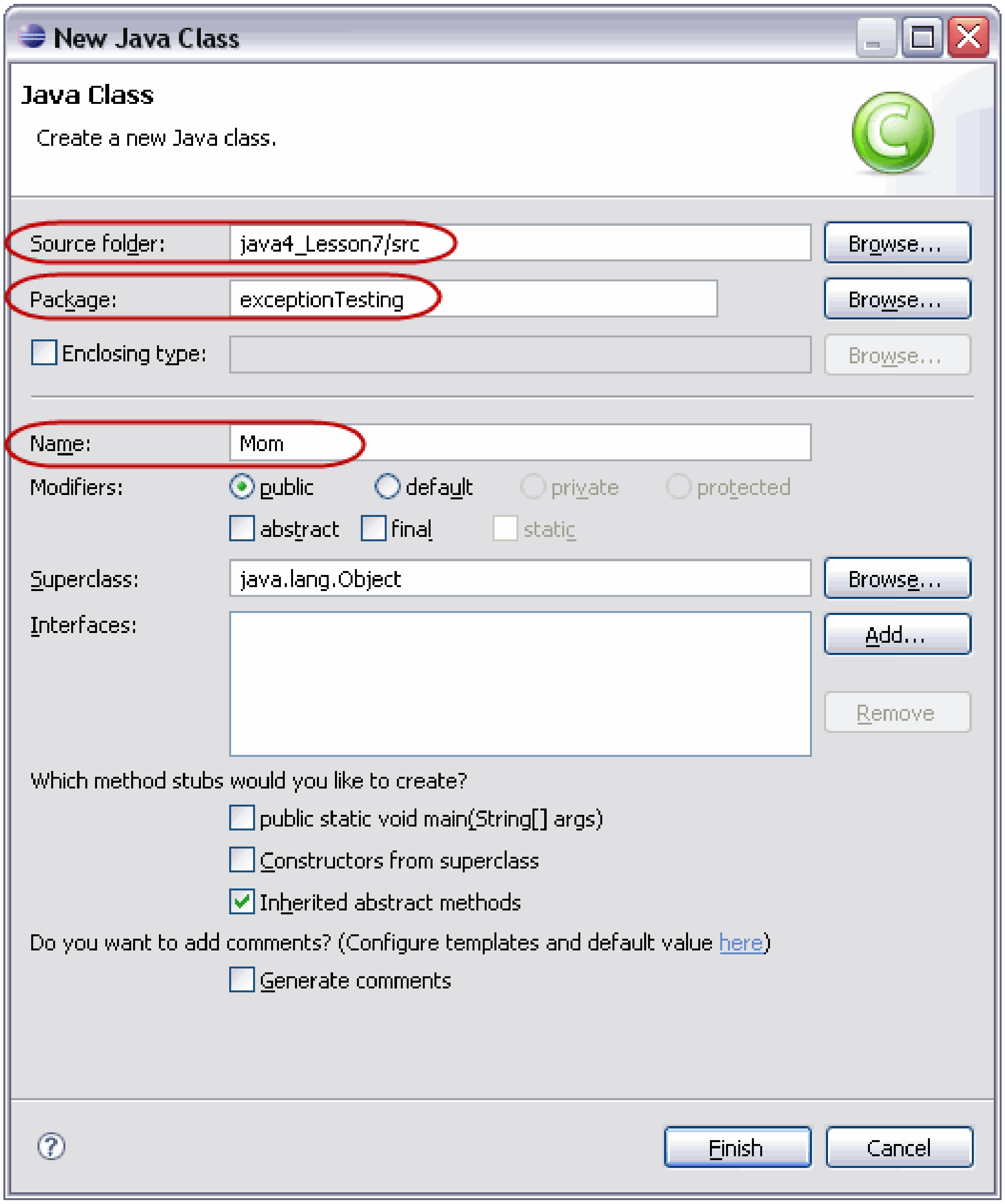
```
package exceptionTesting;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.FileReader;
import java.io.BufferedReader;

public class FileFetcher {
    FileReader myFile;
    BufferedReader in;
    String aLine ="";

    public void getHomework() throws FileNotFoundException, IOException {
        // try {
            myFile = new FileReader("homework.txt");
            System.out.println("I did get here");
            in = new BufferedReader(myFile);
        // }
        // catch (FileNotFoundException e){
        //     System.out.println("Can't find the file, but keep going anyway -
allows for future problems");
        // }
        while (aLine != null){
        //     try {
                aLine = in.readLine();
        //     }
        //     catch(IOException e){
        //         System.out.println("Now we have some other I/O problem");
        //     }
            if (aLine !=null) System.out.println(aLine);             // we had a
nother readLine after the check for null
        }
    }

    //public static void main(String [] args){
    //    FileFetcher testMe = new FileFetcher();
    //    testMe.getHomework();
    //}
}
```

If a method in a class does not catch the exceptions from method calls within itself, then that method needs to throw those uncaught exceptions or it will not compile. Throwing exceptions rather than handling them is *not* the preferred way to write methods. Only do it when it's *absolutely necessary*. If your method can handle the exceptions from the methods that it uses, it should. Throwing methods should happen only when the application cannot deal with the exception in its current method environment.

Passing Exceptions to Other Methods

Now, let's suppose that our student's Mom says she will help, but only with a single exception (she chooses to handle **FileNotFoundException**), and that Dad has to help with the other.

In the java4_Lesson7 project, create a new **Mom** class as shown:



Source folder: java4_Lesson7/src

Package: exceptionTesting

□ Enclosing type:

Name: Mom

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type the **Mom** class, as shown in **blue** below:

CODE TO TYPE: Mom

```
package exceptionTesting;

import java.io.FileNotFoundException;
import java.io.IOException;

public class Mom {

    public void getToDoHomework() throws IOException {
        FileFetcher testMe = new FileFetcher();
        try{
            testMe.getHomework();
        }
        catch(FileNotFoundException e){
            System.out.println("Mom caught the File Not Found Exception.");
        }
    }

    public static void main(String [] args) throws IOException {
        // Note: This is VERY BAD programming. Do not throw exceptions in main m
ethods.
        Mom parent1 = new Mom();
        parent1.getToDoHomework();
    }
}
```

Save and run it. You might get a warning that there are still errors--click **Proceed** anyway.

We are throwing an exception in this **main()** method to demonstrate that it's a terrible thing to do, because no one can catch from **main()**. That's the reason Eclipse warned that you still had errors. However, the code does run. No exception was thrown, so there was nothing for the file name to catch.

Go back to the **FileFetcher** class and change the file name as shown:

CODE TO EDIT: FileFetcher

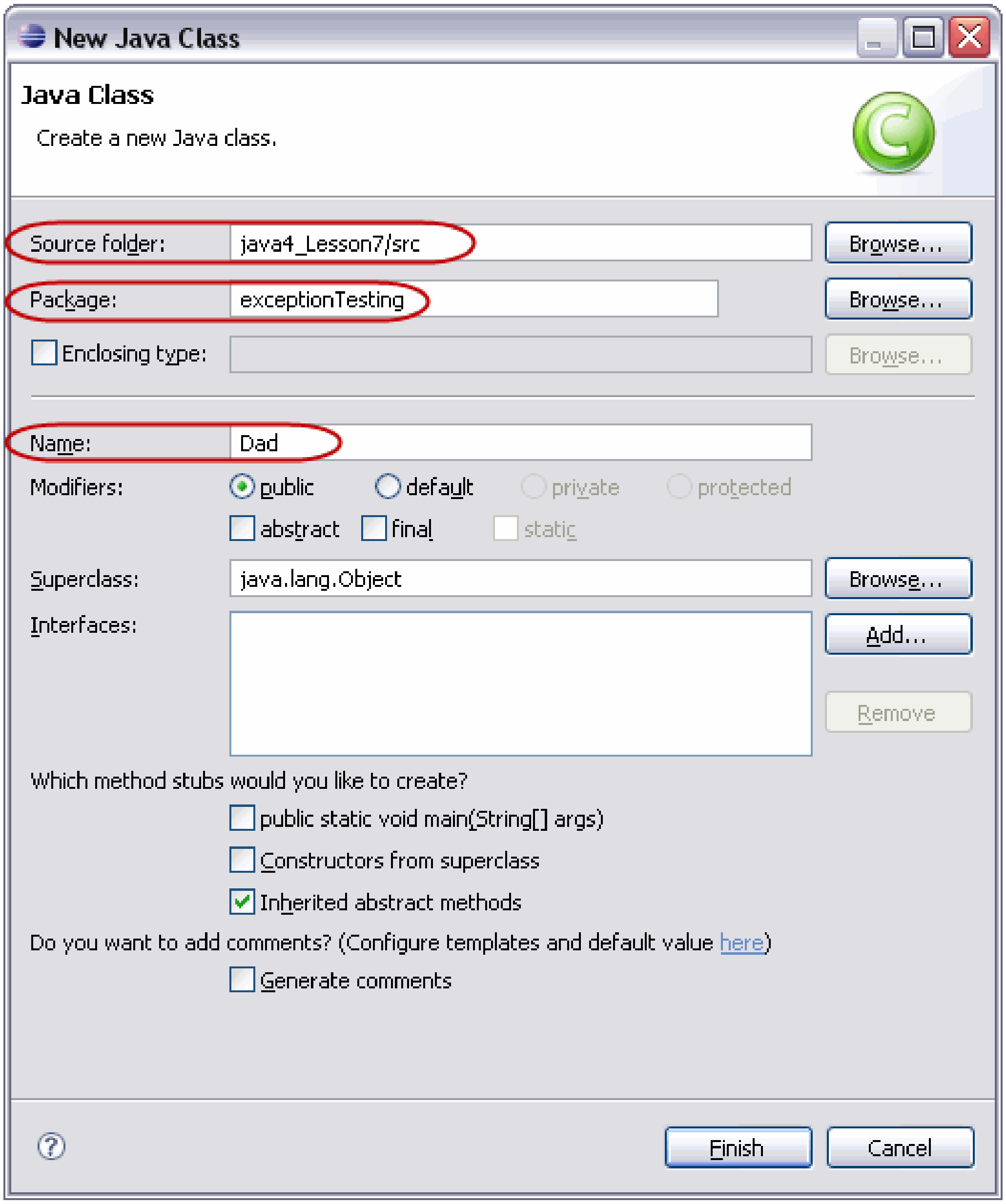
```
package exceptionTesting;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.FileReader;
import java.io.BufferedReader;

public class FileFetcher {
    FileReader myFile;
    BufferedReader in;
    String aLine ="";

    public void getHomework() throws FileNotFoundException, IOException {
    //try {
        myFile = new FileReader("homework2.txt");
        System.out.println("I did get here");
        in = new BufferedReader(myFile);
    //}
    //catch (FileNotFoundException e){
    //    System.out.println("Can't find the file, but keep going anyway - allow
s for future problems");
    //}
    while (aLine != null){
    //    try {
            aLine = in.readLine();
    //    }
    //    catch(IOException e){
    //        System.out.println("Now we have some other IO problem");
    //    }
        if (aLine !=null) System.out.println(aLine);   // we had another readLin
e after the check for null
        }
    }

    //public static void main(String [] args){
    //    FileFetcher testMe = new FileFetcher();
    //    testMe.getHomework();
    //}
}
```

Save it. You haven't created a homework2.txt file, so you probably have a little problem.

Open the **Mom** class and run it. We can see that **Mom** caught the **FileNotFoundException**; the Console shows the **println** from the **catch** clause block. Now in **FileFetcher**, change the filename back to **homework.txt**.

Catching Exceptions from Other Methods

In java4_Lesson7, create a new **Dad** class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson7/src

Package: exceptionTesting

□ Enclosing type:

Name: Dad

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type the **Dad** class as shown in **blue** below:

CODE TO TYPE: Dad

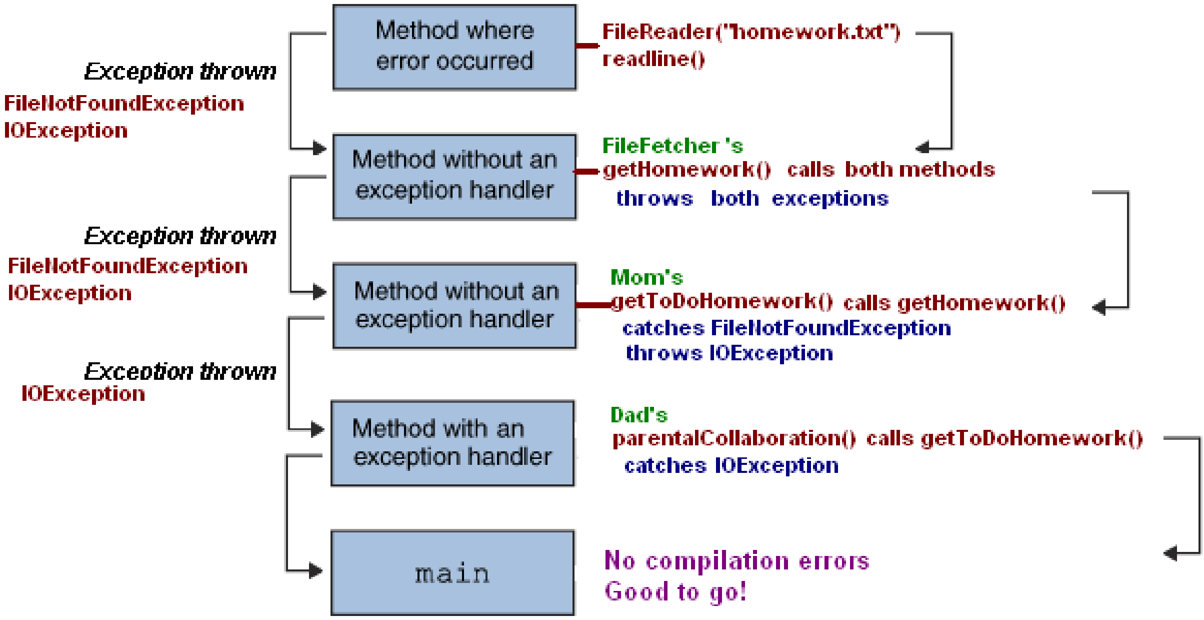
```
package exceptionTesting;

import java.io.IOException;

public class Dad {

    public void parentalCollaboration() {
        Mom spouse = new Mom();
        try{
            spouse.getToDoHomework();
        }
        catch(IOException e){
            System.out.println("Dad caught the I/O Exception.");
        }
    }

    public static void main(String [] args)  {
        Dad parent2 = new Dad();
        parent2.parentalCollaboration();
    }
}
```

Save and run it. You do *not* get any errors, because now you've caught all Exceptions.

One More Time

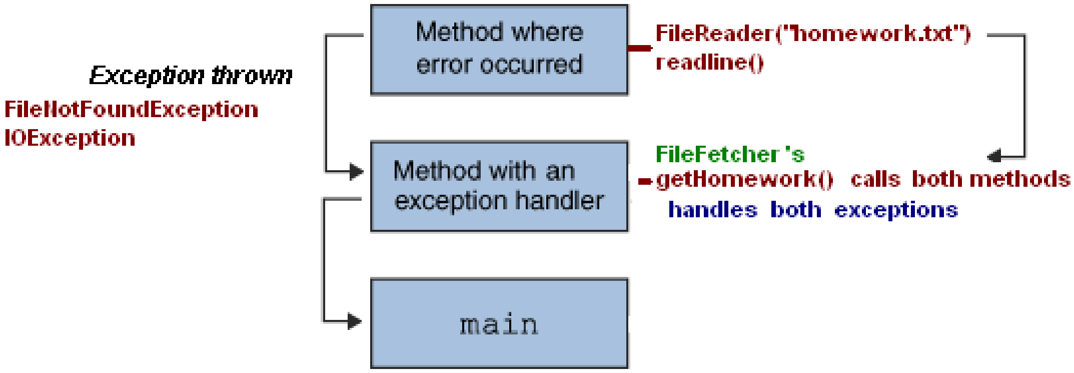
The last example demonstrates the way you can either handle exceptions within your code with try/catch clauses, or have your method throw the exception and let the methods' users handle them.

And in our example that passed an exception down the call stack, we saw this:



Method where error occurred
FileReader("homework.txt")
readline()
Exception thrown
FileNotFoundException
IOException
Method without an exception handler
FileFetcher 's
getHomework() calls both methods
throws both exceptions
Exception thrown
FileNotFoundException
IOException
Method without an exception handler
Mom's
getToDoHomework() calls getHomework()
catches FileNotFoundException
throws IOException
Exception thrown
IOException
Method with an exception handler
Dad's
parentalCollaboration() calls getToDoHomework()
catches IOException
main
No compilation errors
Good to go!

Compare that to our method with a handler:


Exception thrown
FileNotFoundException
IOException
Method where error occurred
FileReader("homework.txt")
readline()
Method with an exception handler
FileFetcher 's
getHomework() calls both methods
handles both exceptions
main

Exception Hierarchy

Let's have Mom handle a different exception. Edit the **Mom** class as shown in **blue**:

CODE TO EDIT: Mom

```
package exceptionTesting;

import java.io.FileNotFoundException;
import java.io.IOException;

public class Mom {

    public void getToDoHomework() throws FileNotFoundException {
        FileFetcher testMe = new FileFetcher();
        try{
            testMe.getHomework();
        }
        catch(IOException e){
            System.out.println("Mom caught the I/O Exception.");
        }
    }

    public static void main(String [] args) throws FileNotFoundException {
        Mom parent1 = new Mom();
        parent1.getToDoHomework();
    }
}
```

And edit the **Dad** class as shown in **blue** below:

CODE TO EDIT: Dad

```
package exceptionTesting;

import java.io.FileNotFoundException;

public class Dad {

   public void parentalCollaboration() {
      Mom spouse = new Mom();
      try{
          spouse.getToDoHomework();
      }
      catch(FileNotFoundException e){
          System.out.println("Dad caught the File Not Found Exception.");
      }
   }

   public static void main(String [] args)  {
      Dad parent2 = new Dad();
      parent2.parentalCollaboration();
   }
}
```

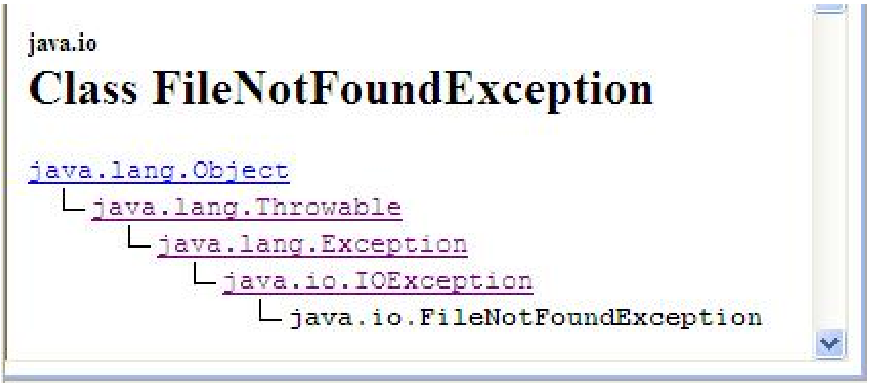
Go to **FileFetcher** and change the filename to **homework2.txt**.

Save all three classes: **FileFetcher**, **Mom**, and **Dad**.

Run from the **Dad** class.

Even though this was a **FileNotFoundException**, **Mom** caught it--we see the output from her catch in the Console. Why? Because she was supposed to catch the **IOExceptions** and Dad was supposed to catch the **FileNotFoundExceptions**.

Go to the **java.io** package. Scroll down to **FileNotFoundException** in the **Exception Summary** and take a look at its hierarchy:



java.io

Class FileNotFoundException

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.io.IOException
              └java.io.FileNotFoundException
```

Exceptions also pay attention to inheritance. **Mom** said she would catch **IOExceptions**, and **FileNotFoundException inherits** from **IOExceptions**, so it actually *is* an **IOException**. If you want to make sure to catch the right exceptions, always catch the more specific one first. Exceptions are often the results of I/O problems.

Using Finally in a Try/Catch Block

A given method can throw more than one exception. In fact, the **getHomework()** method in the **FileFetcher** class threw two exceptions. If you had wanted the **Mom** class to catch and handle both specifically, the method **getToDoHomework()** might have been written like this:

Multiple Catches

```
    public void getToDoHomework(){
        FileFetcher testMe = new FileFetcher();
        try { //Begin "try" block
            testMe.getHomework();
        }
        catch(FileNotFoundException e){
            System.out.println("Mom caught the File Not Found Exception.");
        }
        catch(IOException e) {
            System.out.println("Mom caught the I/O Exception.");
        }
        finally
        {
            System.out.println("After you finish reading the file, you need to c
lose the file streams.");
            try{
                if (testMe.in != null) testMe.in.close();
                    if (testMe.myFile != null) testMe.myFile.close();
            }
            catch(IOException e){}
        } // End "try" block
    }
```

The **Mom** class also added a **finally** clause, to remind **Dad** that he should close his files and input streams when he finishes reading them. The **finally** clause *always* executes when the **try** block exits. This guarantees that the **finally** block is executed even if an unexpected exception occurs. We want to allow a programmer to "clean up" after exceptions have occurred because a call from a **try** clause may have jumped out of code prematurely. The **finally** block is a key tool for preventing resource leaks.

Additional Information

There is plenty more to see in the Java Tutorial Lesson on Exceptions. Check it out. See you in the next lesson...

Threads: Introduction

Multi-Tasking

In the next few lessons, we'll explore the ways Java allows us to coordinate multiple tasks. Programmers need be able to determine which operations are executed and in what order. The Java programming language allows us to write various individual programs that, while running seperately and concurrently, appear as a single, unified and seamless operation to the user. This is accomplished by providing mechanisms for synchronizing the concurrent activity of *threads*.

Threads

A thread is a single sequential flow of control within a program. The API says, "A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently."

Before we discuss the theory of Threads, we'll look at the tools Java provides for us to weave them into our code. We will look at Threads themselves more explicitly in the next lesson.

API Go to the **java.lang** package and scroll down to the **Thread** class and read its introduction. A thread has these qualities:

- It **implements** the **Runable** interface.
- It inherits from **Object**.
- It has a **Nested Class** of **Thread.State**.

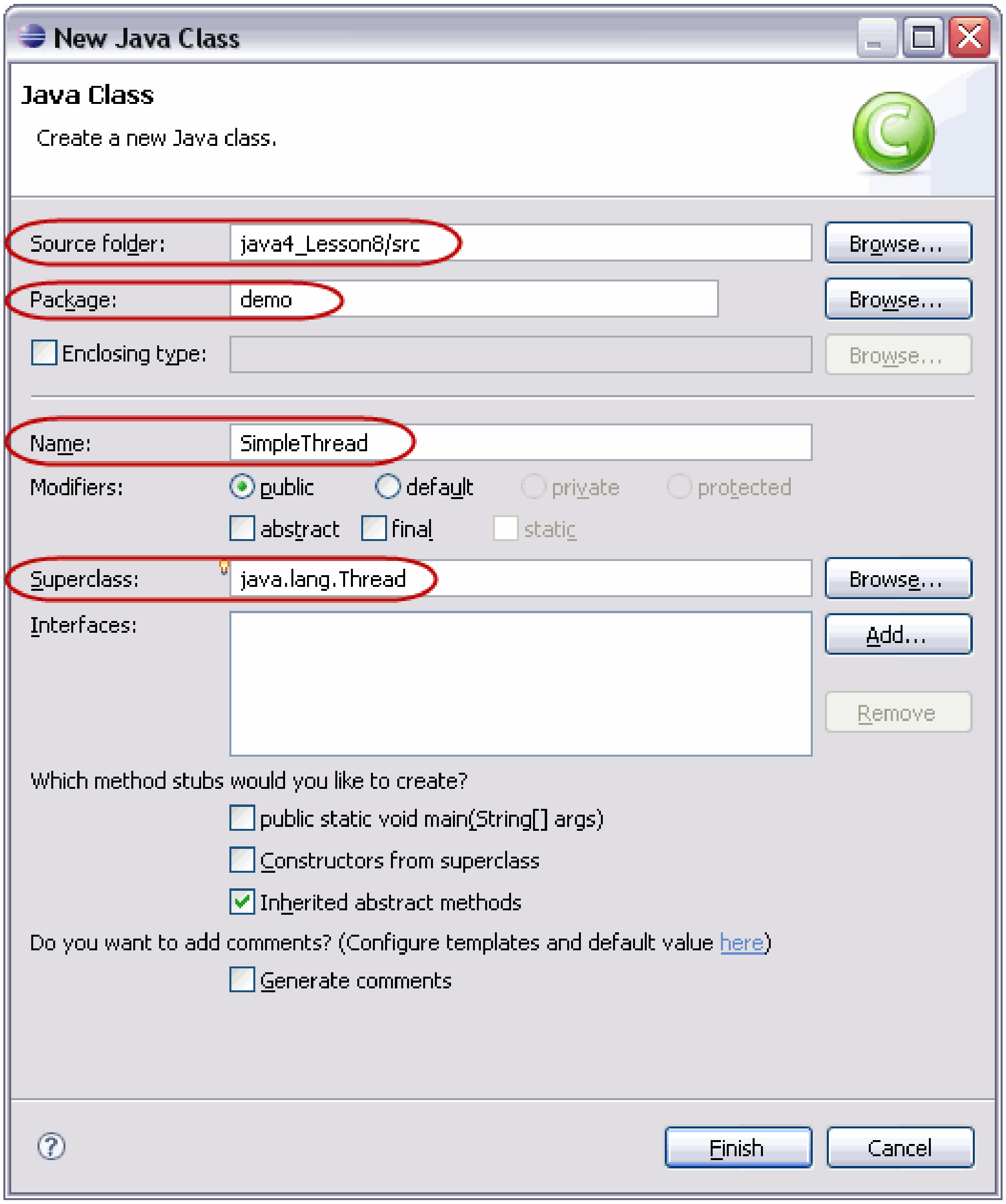
We can take advantage of the capabilities of **Thread**s in Java by:

1. subclassing the **Thread** class.
2. implementing the **Runnable** interface.

We'll demonstrate both of these techniques in this lesson.

Subclassing the Thread Class

Create a new **java4_Lesson8** project. If you're offered the option to **Open Associated Perspective**, click **No**. In this project, create a new **SimpleThread** class as shown:



Source folder: java4_Lesson8/src

Package: demo

□ Enclosing type:

Name: SimpleThread

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Thread

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **SimpleThread** as shown in **blue**:

CODE TO TYPE: SimpleThread

```
package demo;

class SimpleThread extends Thread {

    public SimpleThread(String str) {
        super(str);
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((int)(Math.random() * 1000));
            }                                       // end try
            catch (InterruptedException e) {
            }                                       // end catch
        }                                           // end for loop
        System.out.println("DONE! " + getName());
    }                                                        // end run method
}
```

There are two snippets of code here that we haven't seen before:

1. **super(str)**
2. the try/catch clause surrounding the **sleep()** method

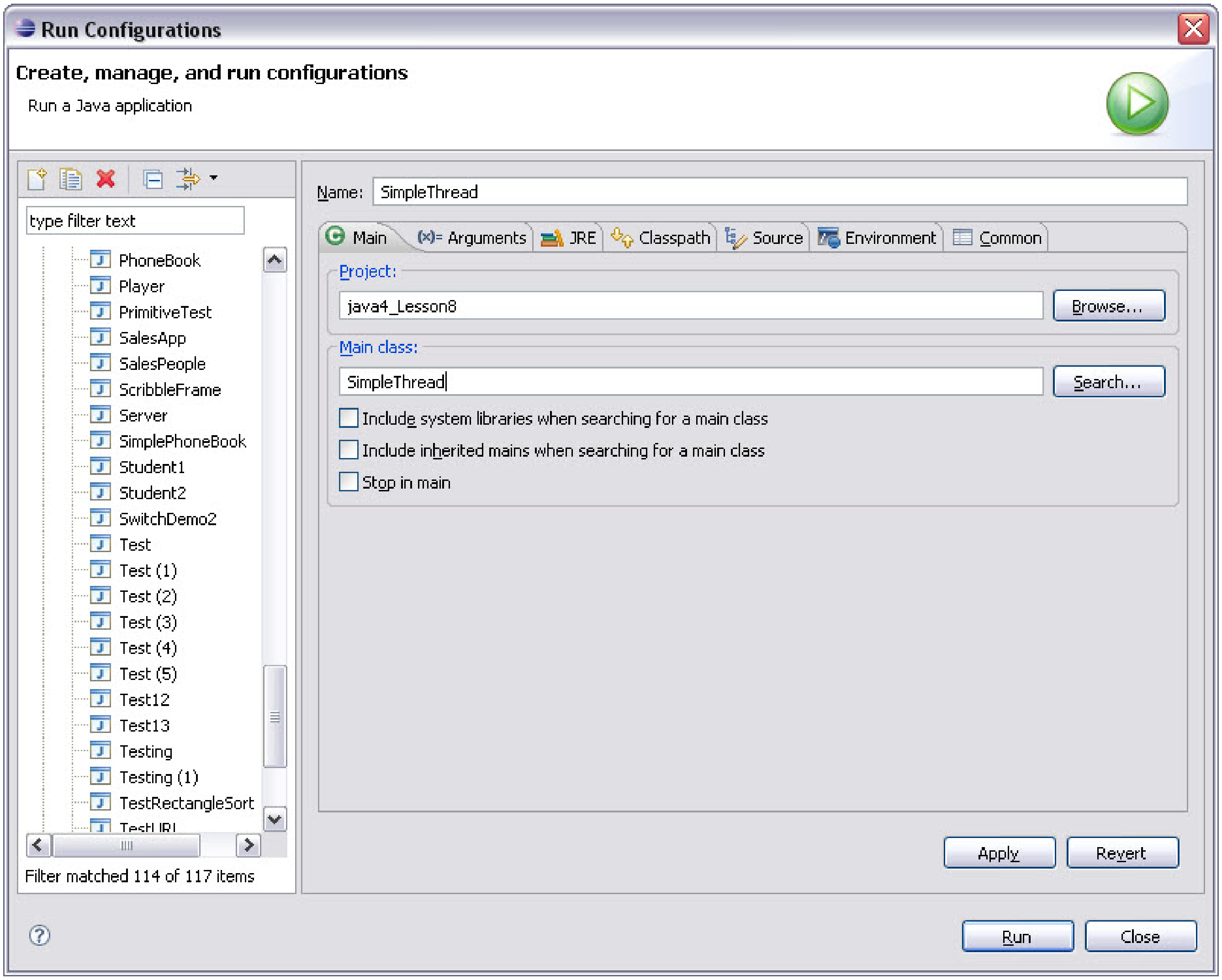
API Let's go to the API to find out more about this code. Go to **java.lang.Thread** and check out the inheritance tree. Our **SimpleThread** class inherits from **Thread**. What would **Thread Constructor** inherit if a **String** was passed to it via **super(str)**?

Go to the **java.lang.Thread Method Summary** and look at the methods there. Both methods present throw **InterruptedException**s. We need to handle them using try/catch clauses.

We've seen the static method call **Math.random()** and casting before, so this might look familiar to you already. If not, go to the **java.lang.Math** class and look at the **random()** method.

Save and run **SimpleThread**.

You see a menu choice **Open Run Dialog**, and then a window:



Run Configurations

Create, manage, and run configurations

Run a Java application

Name: SimpleThread

Project: java4_Lesson8

Main class: SimpleThread

☐ Include system libraries when searching for a main class

☐ Include inherited mains when searching for a main class

☐ Stop in main

Filter matched 114 of 117 items

Go ahead and choose **Run**. Hmm. Something's wrong. There are a number of questions to consider:

- Is your class an **Applet**?
- Does your **application** have a **main()** method?
- Have you *instantiated* your **SimpleThread** to make an instance?
- Did Eclipse simply grab the last application you ran?

Let's add a **main()** method to test. Edit **SimpleThread** as shown in **blue** below:

CODE TO EDIT: SimpleThread

```
package demo;

class SimpleThread extends Thread {

    public SimpleThread(String str) {
       super(str);
    }

    public void run() {
       for (int i = 0; i < 10; i++) {
           System.out.println(i + " " + getName());
           try {
               sleep((int)(Math.random() * 1000));
           }
           catch (InterruptedException e) {
           }
       }
       System.out.println("DONE! " + getName());
    }

    public static void main (String [] args){
        SimpleThread st = new SimpleThread("myGuy");
    }
}
```

Save and run it.

Nothing happened. That's because *you* have direct control of your thread, so *you* must start it explicitly. Java **Threads** need to be *instantiated* like any other class. However, you don't call the **run()** method explicitly; you begin by invoking the **start()** method. Edit **SimpleThread** again. Add the code in **blue** as shown:

CODE TO EDIT: SimpleThread

```
package demo;

class SimpleThread extends Thread {

    public SimpleThread(String str) {
       super(str);
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((int)(Math.random() * 1000));
            }
            catch (InterruptedException e) {
            }
        }
        System.out.println("DONE! " + getName());
    }

    public static void main (String [] args){
        SimpleThread st = new SimpleThread("myGuy");
        st.start();
    }
}
```

Save and run it.

Note Starting **Thread**s is different from starting other classes and methods. **Thread**s *must* have a **run()** method in order to operate, but you start the **run()** method by invoking the instance of the **Thread** with **start()**.

API Go to the **java.lang.Thread** class and look at its **start()** method. Here (and in general) we inherit the method **start()** from our thread **SimpleThread**'s parent **Thread**.

In the instance of the **Thread** that we're running here, we instruct it to **sleep()** for a few milliseconds. This stops it from running for *at least* the specified time and then allows it to continue.

Let's see what happens when we comment out the **try/catch** block with the sleep call:

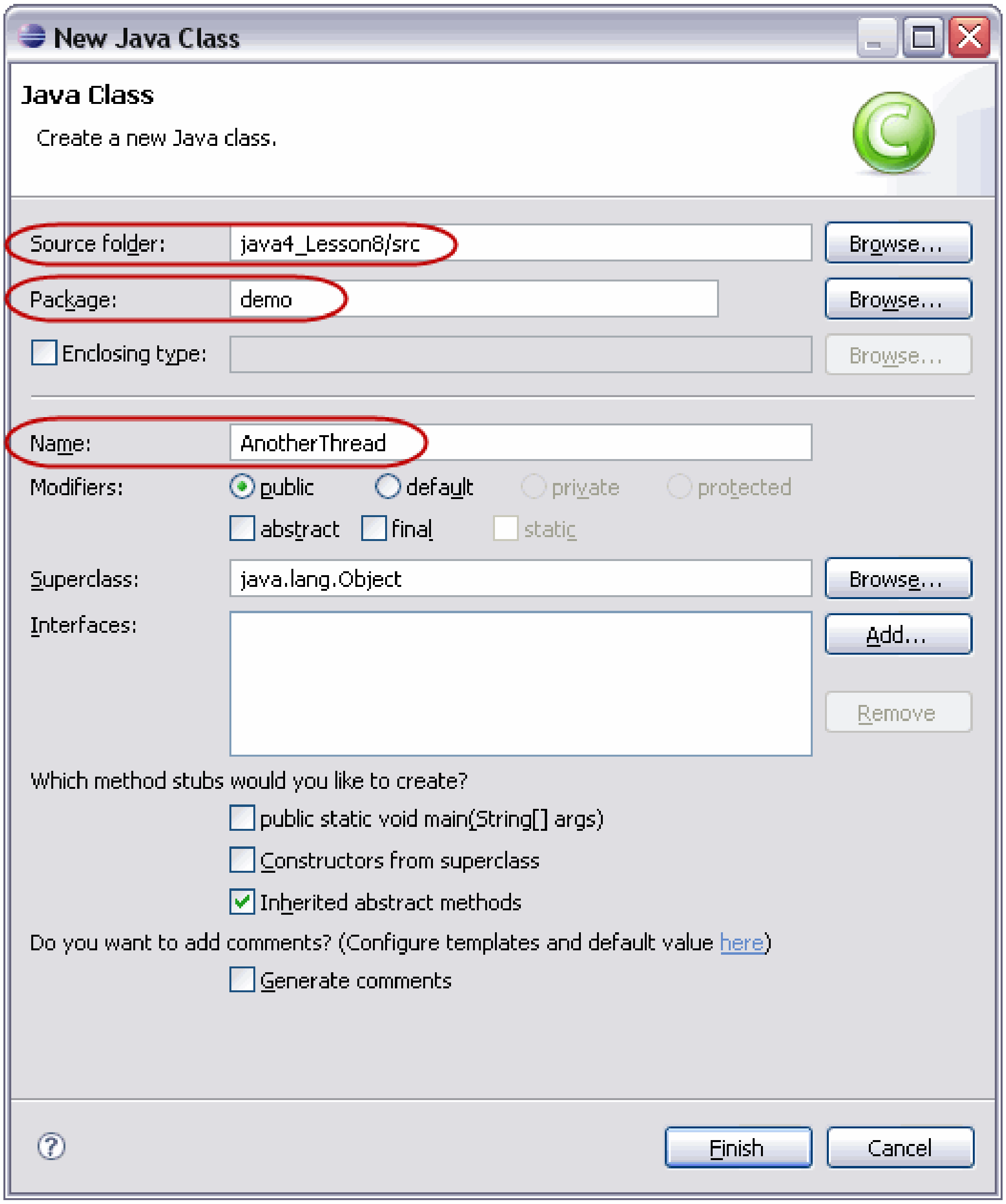
CODE TO EDIT: SimpleThread

```
package demo;

class SimpleThread extends Thread {

    public SimpleThread(String str) {
        super(str);
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
           System.out.println(i + " " + getName());
           //try {
           //     sleep((int)(Math.random() * 1000));
           //}
           //catch (InterruptedException e) {
           //}
        }
        System.out.println("DONE! " + getName());
    }

     public static void main (String [] args){
         SimpleThread st = new SimpleThread("myGuy");
         st.start();
     }
}
```

Save and run it. Take the comment slashes **(//)** out, then save and run it again. Putting threads to **sleep()** allows more time for other actions (from other threads) to take place.

Manipulating Threads

Let's experiment some more. Create a new **AnotherThread** class in the java4_Lesson8 project as shown:



Java Class

Create a new Java class.

Source folder: java4_Lesson8/src

Package: demo

□ Enclosing type:

Name: AnotherThread

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **AnotherThread** as shown in **blue**:

CODE TO TYPE: AnotherThread

```
package demo;

// Threads are in java.lang.Thread so no import is needed

public class AnotherThread {

    public static void main(String args[]) {
        T  t = new T();
        t.start();
    }
}

class T extends Thread {

    public void run() {
        while(true) {                        // forever,
            System.out.println("b: ");       // prompt the user with a b:, wait
ing for them to do something
            try {
                sleep(300);          // sleep is in milliseconds
            }
            catch (InterruptedException e){}
        }
    }
}
```

We have defined the class **AnotherThread**, which has a nested class named **T**, which extends **Thread**. The variable **t** in the **main()** method of class **AnotherClass** should contain a valid thread of execution for an instance of the subclass of **Thread** that we named **T**. We control this thread in the **run()** method.

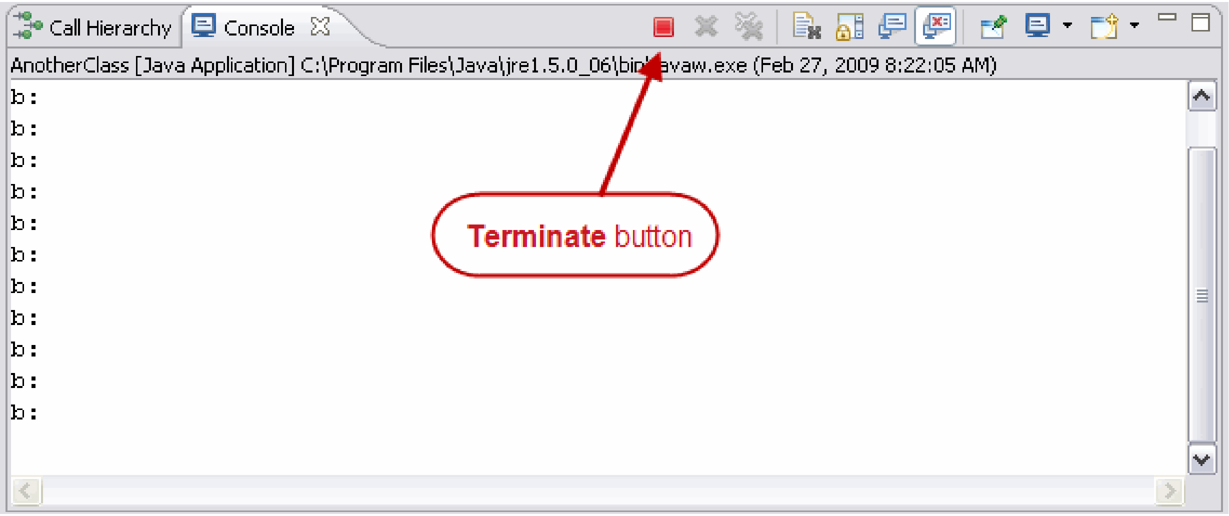
Once inside the **run()** method, we're able execute statements just like in any program. In these examples, we are pausing for a specified period of time. In our first example, the period of time was random; in the above class, it's 300 milliseconds. In our code it's written like this: **sleep(300)**.

The **sleep()** method tells a thread to pause for at least the specified number of milliseconds. The **sleep()** method does not take up system resources while the thread sleeps. Other threads can continue to work.

Save and run it.

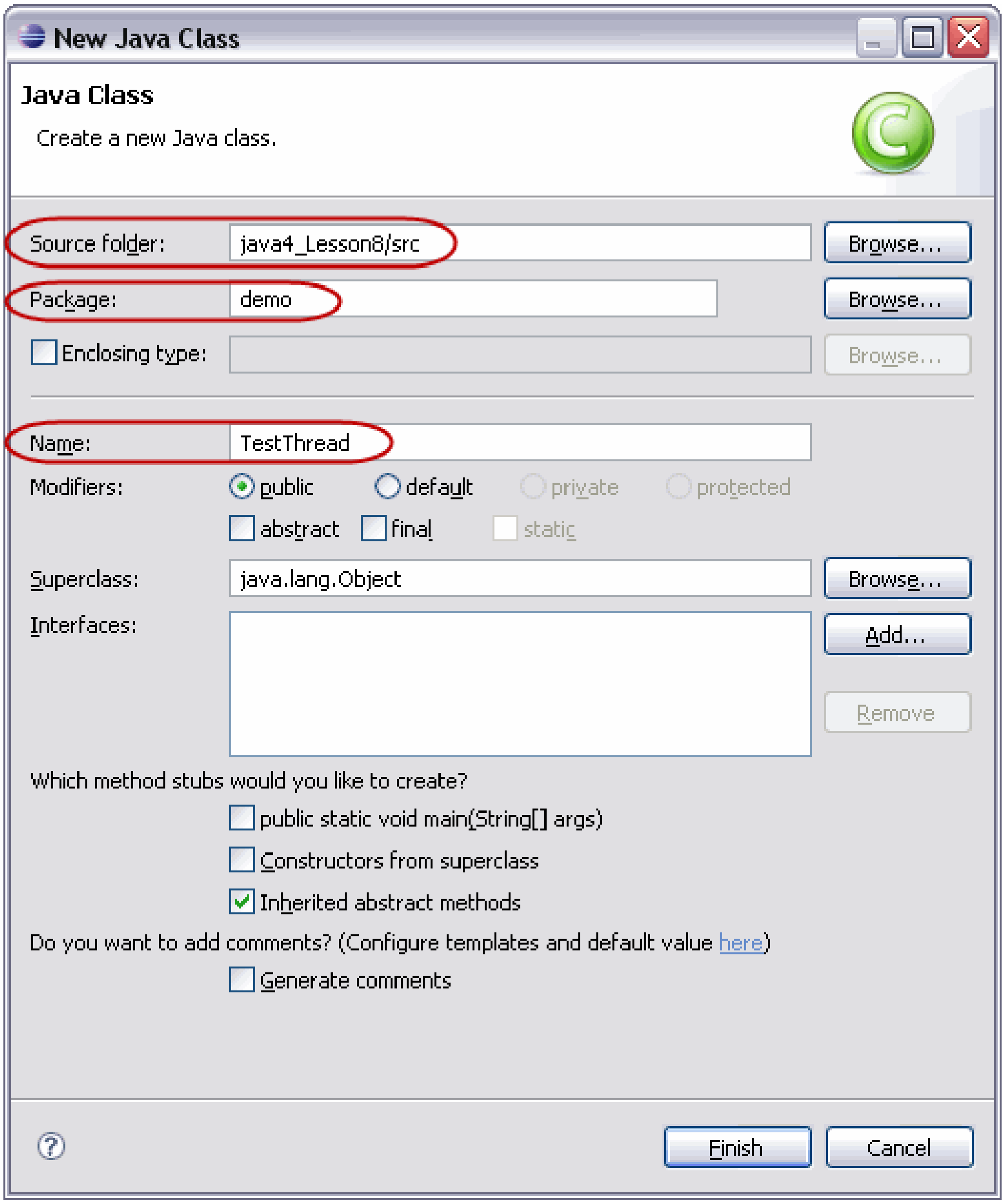
Normally, threads stop when their **run()** method is complete. In this thread, however, we have an infinite loop in the **run()** method.

[**Ctrl+c**] will stop most execution processes. *However*, in this case (that is, within Eclipse within a thread within our control), you click the **Terminate** box to stop execution:



```
AnotherClass [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Feb 27, 2009 8:22:05 AM)
b:
b:
b:
b:
b:
b:
b:
b:
b:
b:
b:
```

Here's another example that uses threads and passes parameters. In the java4_Lesson8 project, add a new **TestThread** class as shown:



Source folder: java4_Lesson8/src

Package: demo

□ Enclosing type:

Name: TestThread

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **TestThread** as shown in **blue** below:

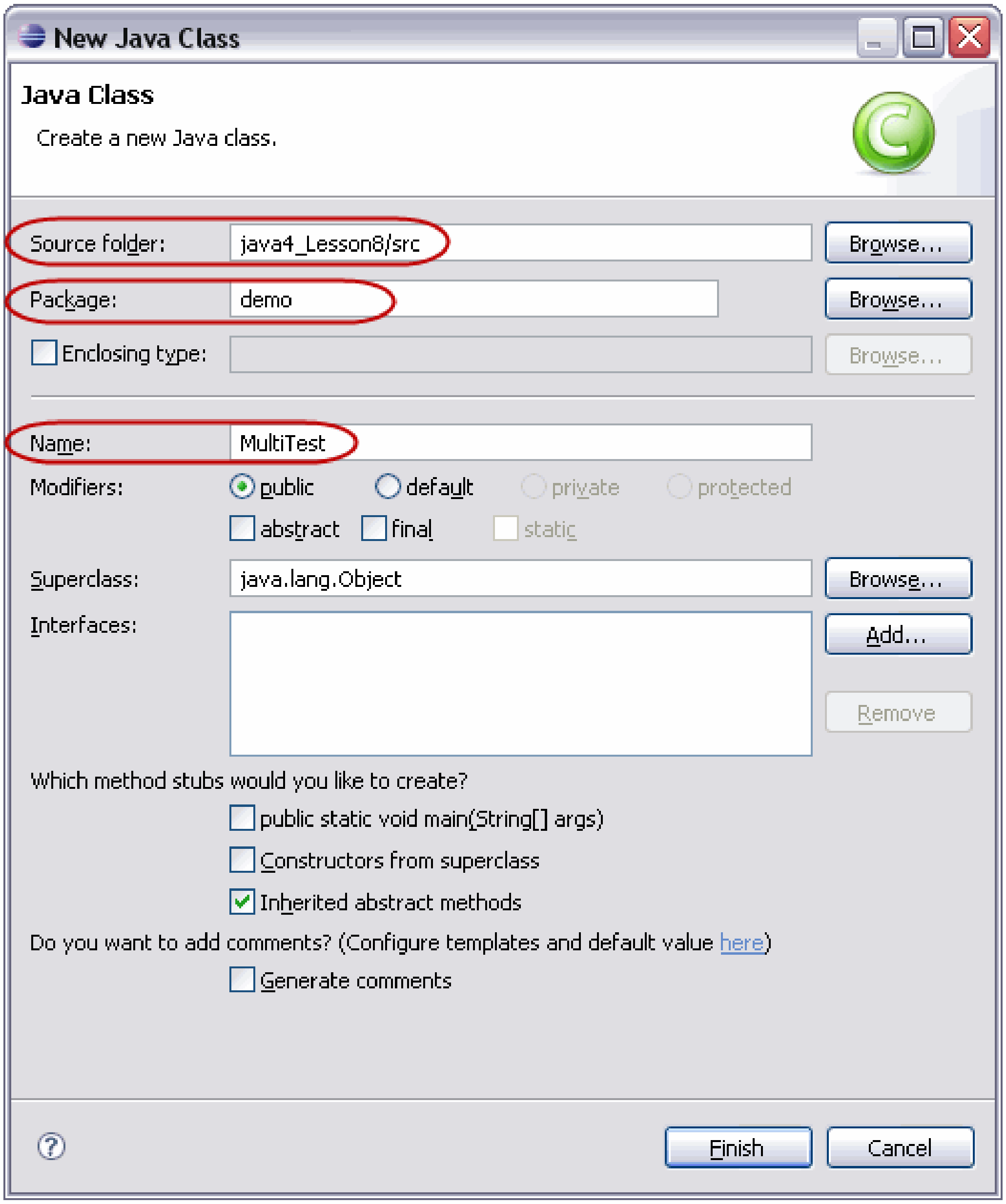
CODE TO TYPE: TestThread

```
package demo;                                        // Define our simple threads.
They will pause for a short time
                                                     // and then print out their nam
es
class TestThread extends Thread {
    private String whoAmI;
    private int delay;

    public TestThread(String s, int d) {  // Our constructor to receive the name
 (whoAmI) and time to sleep (delay)
        whoAmI = s;
        delay = d;
    }
                                          // run--the thread method similar to m
ain()--when run is finished, the thread dies.
    public void run() {                   // run is called from the start() meth
od of Thread
        try {
            sleep(delay);
        }
        catch (InterruptedException e) {
        }
        System.out.println(whoAmI + " has delay time of " + delay);
    }
}
```

The **run()** method serves as the **main()** routine for threads. Just like **main()**, when **run()** finishes, so does the thread.

Applications use the **main()** method to retrieve their arguments from the **args** parameter (which is typically set in the command line). A newly created thread must receive its arguments programmatically from the originating thread. That way parameters can be passed in through the constructor, static variables, or any other technique designed by the developer. In the **TestThread** example, we pass the parameters through the constructor. We need to create a class to instantiate a few instances of this **TestThread**.

In the java4_Lesson8 project, add a new class named **MultiTest** as shown:



Java Class

Create a new Java class.

Source folder: java4_Lesson8/src

Package: demo

□ Enclosing type:

Name: MultiTest

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **MultiTest** as shown in **blue** below:

CODE TO TYPE: MultiTest

```
//  A simple multithread test program
package demo;

public class MultiTest {

    public static void main(String args[]) {
        TestThread t1, t2, t3;
        // Instantiate/create our test threads
        t1 = new TestThread("Thread1",(int)(Math.random()*1000));
        t2 = new TestThread("Thread2",(int)(Math.random()*2000));
        t3 = new TestThread("Thread3",(int)(Math.random()*3000));

        // Start each of the threads
        t1.start();
        t2.start();
        t3.start();
        // At this point we have started 3 threads!
    }
}
```

Save both **TestThread** and **MultiTest**.

Run **MultiTest** using its **main()** method.

Now, try changing the maximum number of random times as shown in **blue**:

CODE TO EDIT: MultiTest

```
//  A simple multithread test program
package demo;

public class MultiTest {

    public static void main(String args[]) {
        TestThread t1, t2, t3;
        // Instantiate/create our test threads
        t1 = new TestThread("Thread1",(int)(Math.random()*3000));
        t2 = new TestThread("Thread2",(int)(Math.random()*2000));
        t3 = new TestThread("Thread3",(int)(Math.random()*1000));

        // Start each of the threads
        t1.start();
        t2.start();
        t3.start();
        // At this point we have started 3 threads!
    }
}
```

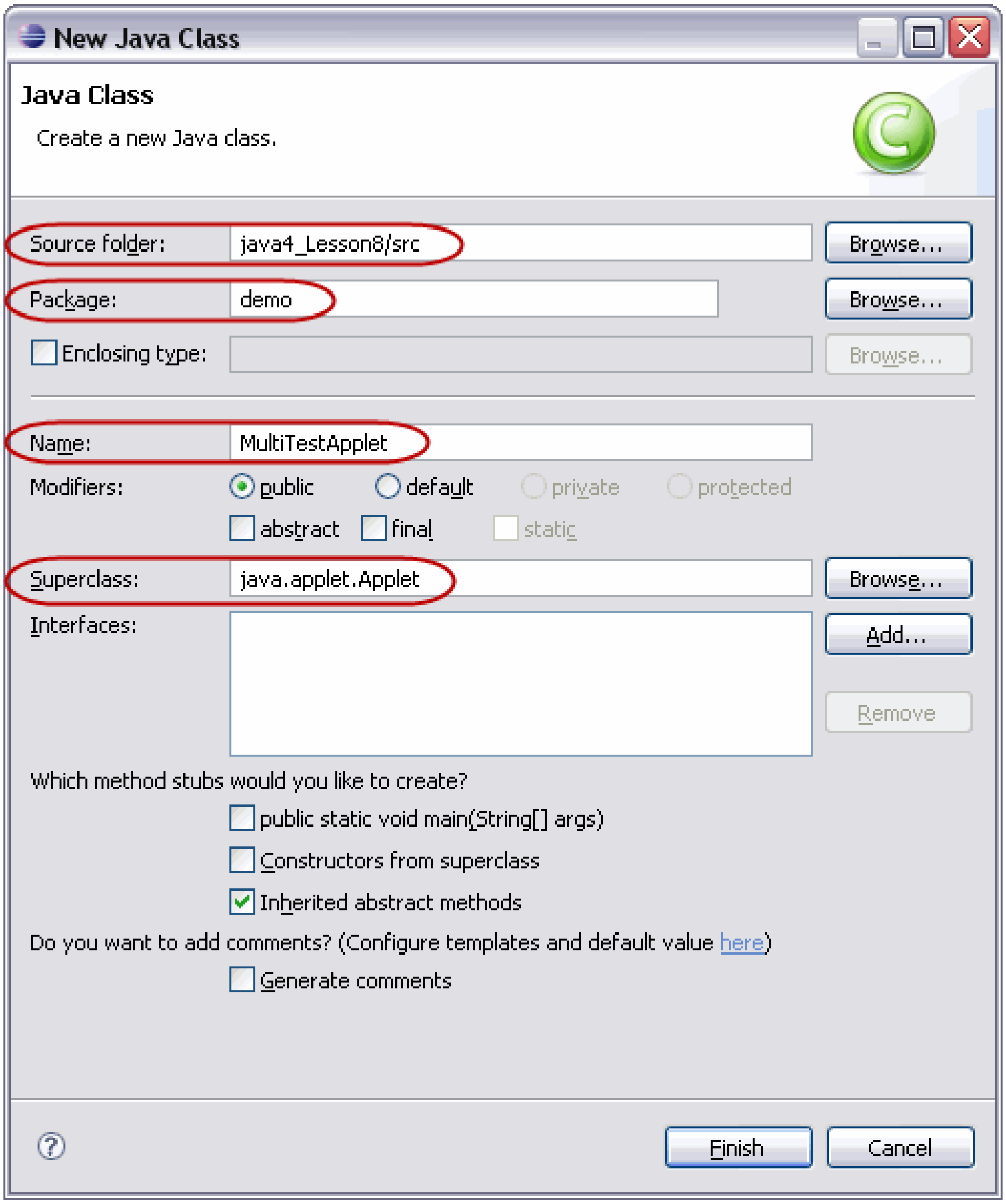
Remember that these are *random*, so the order in which they appear does not *necessarily* indicate increased or decreased delays.

Save and run **MultiTest** again.

Threads in Applets

Applets can have **Thread**s too. With a few adaptations, our application above can be turned into an Applet.

In java4_Lesson8, create a new **MultiTestApplet** class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson8/src

Package: demo

□ Enclosing type:

Name: MultiTestApplet

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.applet.Applet

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type MultiTest as shown:

CODE TO TYPE: MultiTestApplet

```
package demo;

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class MultiTestApplet extends Applet implements ActionListener {

    TestThread t1, t2, t3;

    public void init() {
        Button runUs = new Button("Run Threads");                   // create a B
utton
        add(runUs);                                                 // add it to
the Applet
        runUs.addActionListener(this);                              // add a List
ener to the Button

        t1 = new TestThread("Thread1",(int)(Math.random()*1000));  // instantiat
e our 3 TestThreads
        t2 = new TestThread("Thread2",(int)(Math.random()*2000));
        t3 = new TestThread("Thread3",(int)(Math.random()*3000));
    }

    public void actionPerformed(ActionEvent e){
        t1.start();                                                 // clicking t
he Button will allow us to start the threads
        t2.start();
        t3.start();
    }
}
```

Save and run it. Click on the button. There's output in the Console because we used **System.out.println**. For now, close this Applet. We'll come back to it later when we look at *Thread States*.

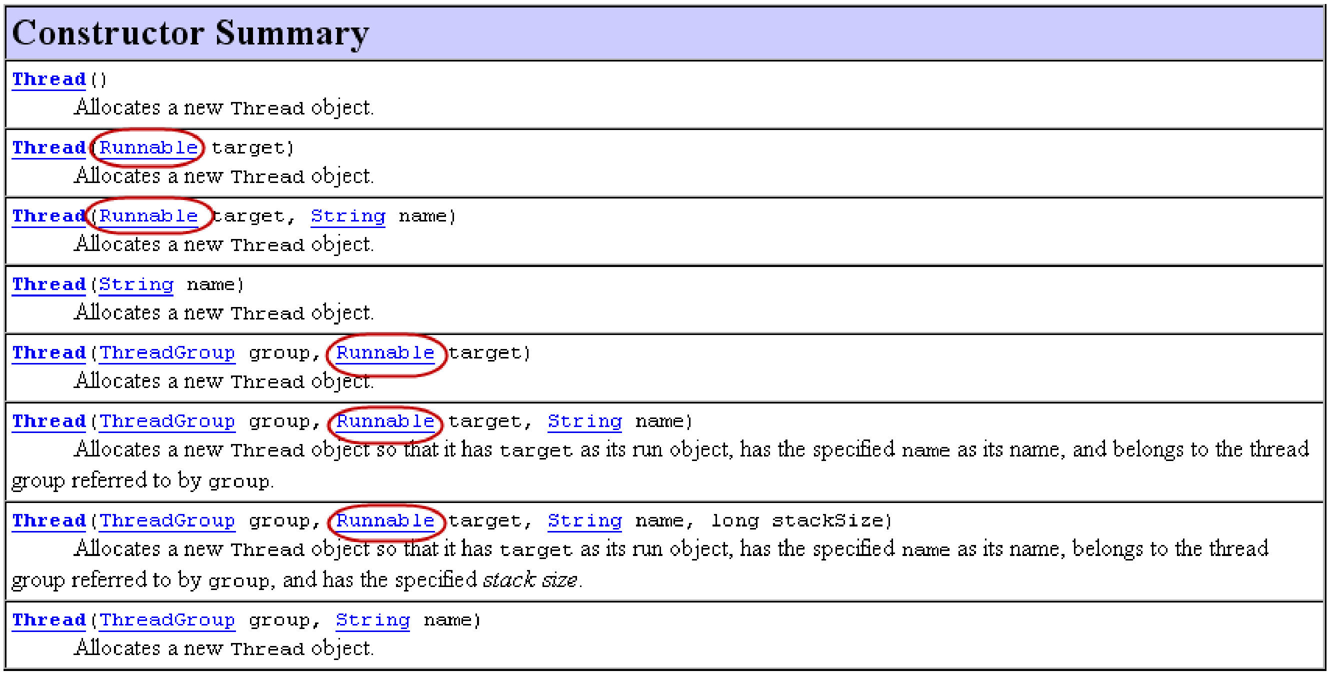
Implementing the Runnable Interface

Sometimes we want a class to *use* a **Thread**, but we do not want that class to *be* a **Thread**. Java allows only single inheritance, so if a class inherits from **Applet** it cannot inherit from **Thread** at the same time. We get around this issue using **Interface**s. You can also create a thread by declaring a class that **implements** the **Runnable** interface.

API in the **java.lang** package, go to the **Interface Summary** and choose **Runnable**. Scroll down its description to see the methods that this interface specifies. You'll actually only see one method in the **Method Summary**: **run()**.

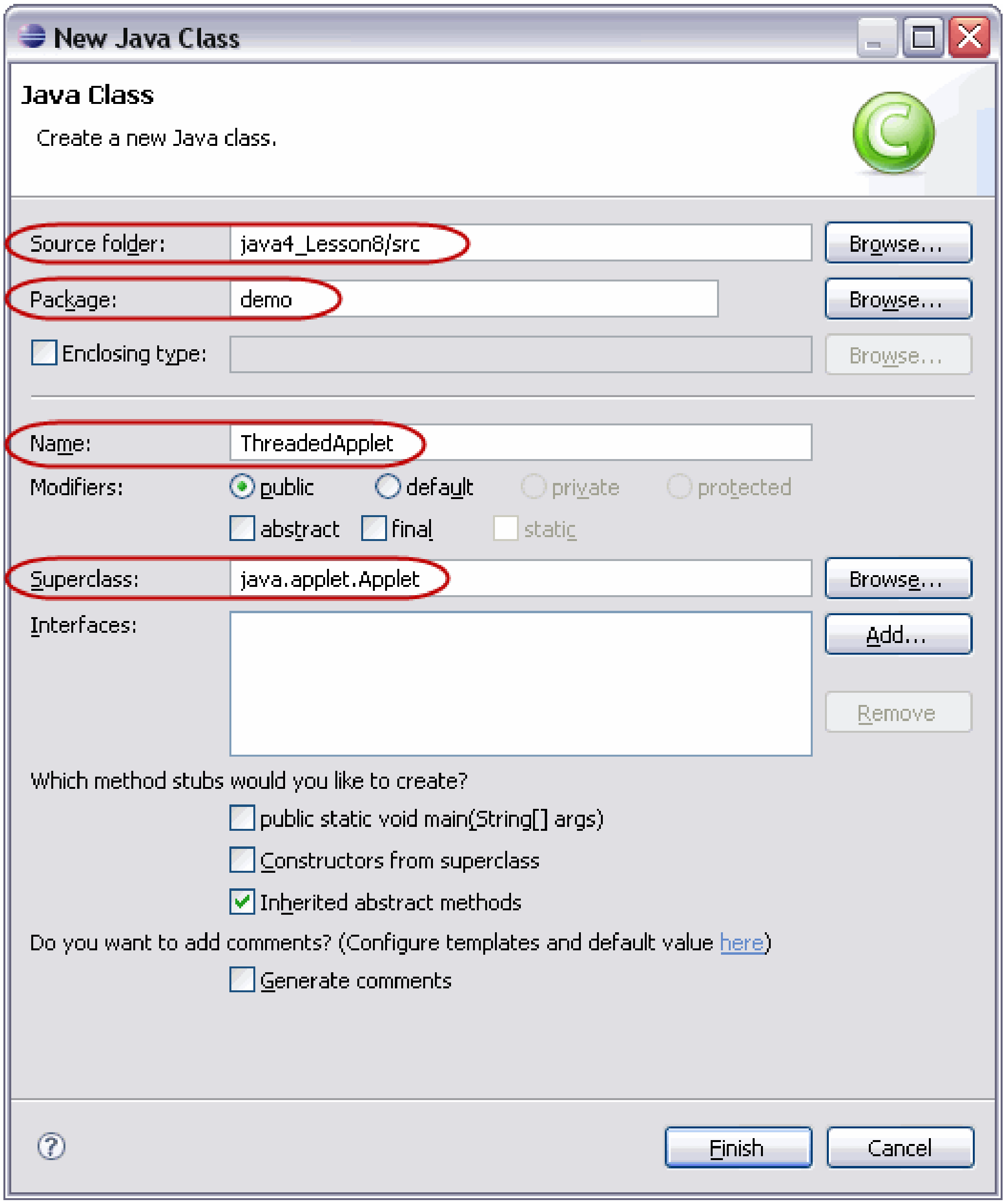
If a class **implements Runnable**, then the class must implement the **run()** method. An instance of the class can then be allocated, passed as an argument when creating a **Thread**, and started. We'll go to the API to see what this means, and demonstrate it.

API Go to the class java.lang.Thread and read through its constructors. Many of them have a parameter of type **Runnable**:



Constructor Summary
Thread() Allocates a new Thread object.
Thread(Runnable target) Allocates a new Thread object.
Thread(Runnable target, String name) Allocates a new Thread object.
Thread(String name) Allocates a new Thread object.
Thread(ThreadGroup group, Runnable target) Allocates a new Thread object.
Thread(ThreadGroup group, Runnable target, String name) Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.
Thread(ThreadGroup group, Runnable target, String name, long stackSize) Allocates a new Thread object so that it has target as its run object, has the specified name as its name, belongs to the thread group referred to by group, and has the specified *stack size*.
Thread(ThreadGroup group, String name) Allocates a new Thread object.

Let's try an example. In java4_Lesson8, create a new class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson8/src

Package: demo

□ Enclosing type:

Name: ThreadedApplet

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.applet.Applet

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **ThreadedApplet** as shown in **blue**:

CODE TO TYPE: ThreadedApplet

```
package demo;

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

public class ThreadedApplet extends Applet implements Runnable {

    Thread appletThread;      // the thread we make will be an instance of the class Th
read
    String messages[] = {"Hello Thread World!" , "I'm doing fine." , "Goodbye for now!"
};
    int i = 0;

    public void paint(Graphics g) {
     g.drawString(messages[i], 15, 50);
    }

    public void run() {
        while (true){
         i = (i+1)  % messages.length;
         repaint();
         try {
          appletThread.sleep(5000);
         } catch (InterruptedException e){}
     }
    }

    public void start() {
        appletThread = new Thread(this);
        appletThread.start();
    }
}
```

This program is an **Applet** with a **run()** method that cycles to print different messages. When the Applet starts, it instantiates the **Thread** and then starts the Thread instance. In this example, we passed the Thread Constructor an instance of an object that implements **Runnable** (that is, the Applet itself--**this**). The Thread then comes back to the Applet to get the **run()** method that it implemented. This is especially convenient, because then the **ThreadedApplet**'s **paint(Graphics g)** method and the implemented **run()** method can share the **messages[ ]** instance variable.

Save and run it. Sit back and watch for a while.

Here the ThreadedApplet class has a **start()** method **for the Applet**. In this method, the thread is instantiated and *its* **start()** method (**appletThread.start();**) is invoked. Of course, we'll want a **stop** for our thread too. For now, you can stop the thread and applet by quitting the applet.

One More Time

To create a thread, you must subclass **Thread** and define this subclass's own **run()** method **or** you must pass a **Runnable** object--which means the object must implement a **run()** method--to the **Thread** Constructor.

The **Runnable** interface specifies the **run()** method that is required. Any class that implements this interface can provide the *body* of a thread. By implementing the Runnable interface, you declare your intention to run a separate thread.

Whichever way you look at threads, the operative word is **run()**. More details on Java Threads are on the way!

Threads: Concurrent Programming

Behind the Scenes

In concurrent programming, there are two basic units of execution: process and thread.

A *process* has a self-contained execution environment. Most computer users **use** processes without knowing it. Processes are usually located at the level of operating systems software, or kernel-level entities. A process generally has a complete, private set of basic run-time resources. Specifically, each process has its own registers, program counter, stack pointer, and memory space.

Threads exist within a process-—every process has at least one thread. In Java, concurrent programming is accomplished mostly through threads. Sometimes threads are called *lightweight* processes or *execution contexts*. Both processes and threads provide an execution environment (like processes, threads have their own registers, program counters, stack pointers, etc.), but threads are closer to a user-level entity. We can create threads and tell them what to do using fewer resources than we do when we create and inform new processes.

Note Multiple threads running at the same time and *sharing* resources have the potential to cause problems. We'll go over some of these issues in this lesson. We'll talk about others in the next lesson when we explore synchronization. For now, we'll focus on plain old multi-tasking.

Multi-threaded Applications

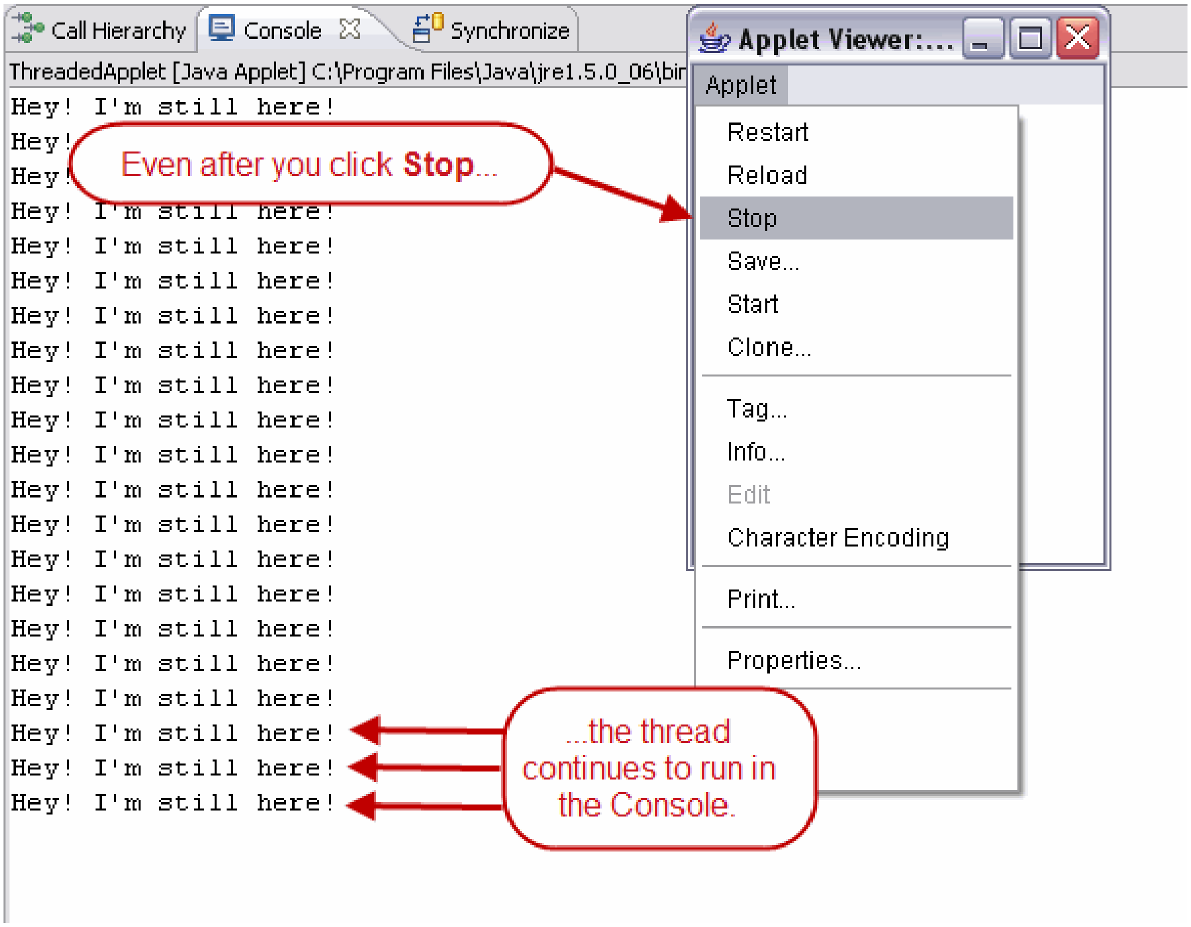
The Life of a Thread

Let's start with an example of multi-tasking. In the editor, open your java4_Lesson8 folder to the **demo** package and open the **ThreadedApplet** class. Edit **ThreadedApplet** as shown in **blue** below:

CODE TO TYPE: ThreadedApplet

```
package demo;

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

public class ThreadedApplet extends Applet implements Runnable {

    Thread appletThread;  // the thread we make will be an instance of the class
 Thread
    String messages[] = {"Hello Thread World!" , "I'm doing fine." , "Goodbye fo
r now!"};
    int i = 0;

    public void paint(Graphics g) {
        g.drawString(messages[i], 15, 50);
    }

    public void run() {
        while (true){
            i = (i+1)  % messages.length;
            repaint();
            System.out.println("Hey! I'm still here");
            try {
                appletThread.sleep(5000);
            } catch (InterruptedException e){}
        }
    }

    public void start() {
        appletThread = new Thread(this);
        appletThread.start();
    }
}
```

Save and run it. Watch it run for a while, then in the Applet Viewer Window, select **Applet | Stop** to stop it. Now, watch the Console for a minute.



ThreadedApplet [Java Applet] C:\Program Files\Java\jre1.5.0_06\bin

```
Hey! I'm still here!
Hey!
Hey!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
```

Even after you click **Stop**...

...the thread continues to run in the Console.

Our Applet has stopped (it's no longer painting), but its Thread is still running (it's still putting output into the Console). **Quit** the Applet to close the Applet Viewer Window; the action in the Console stops.

What's Happening in the Background?

Our **ThreadedApplet** illustrates that we need to take care when using multiple threads. Have you ever opened a web page that seemed to slow your machine down--even after you left the page? Our example shows us the reason behind that. We **stopped** the Applet (which happens when you leave a browser page that's running the applet), but we did not **stop** our Applet's thread. Remember--stopping the thread and stopping the applet are two distinct processes.

We will fix this by making our code cleaner, but first let's go over one more background item.

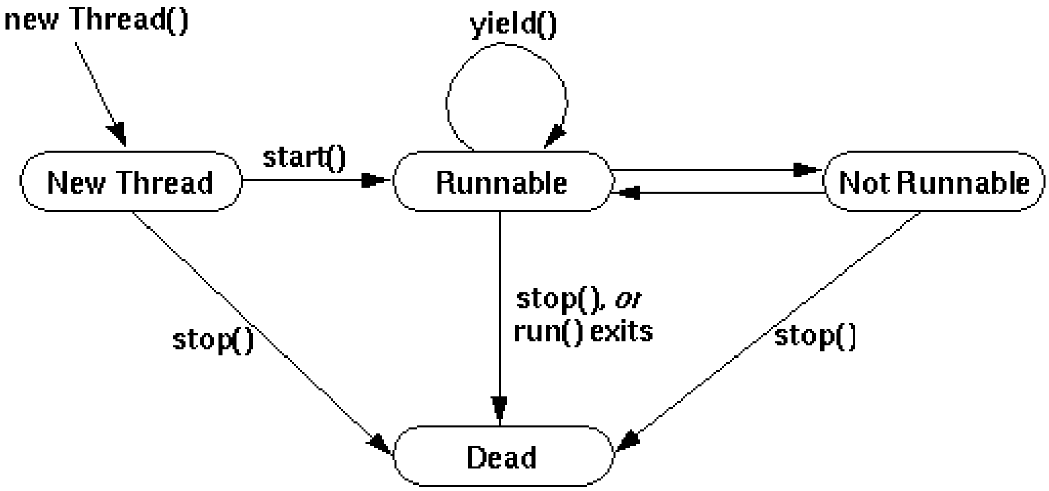
Garbage Collection

One example of a thread working in the background in Java is in *garbage collection*--retrieving memory that has been allocated, but is no longer being used. Java collects garbage using **Thread**s. While you are running your program, the Java Virtual Machine has a garbage collection thread cleaning up in the background.

Here are some links to more information on this topic. This JavaWorld article explains the concept of garbage collection. Oracle also provides a useful page that explains Tuning Garbage Collection.

Thread States

The first Java tutorial on **Thread**s provides the state transition diagram below described as "an overview of the interesting and common facets of a thread's life":



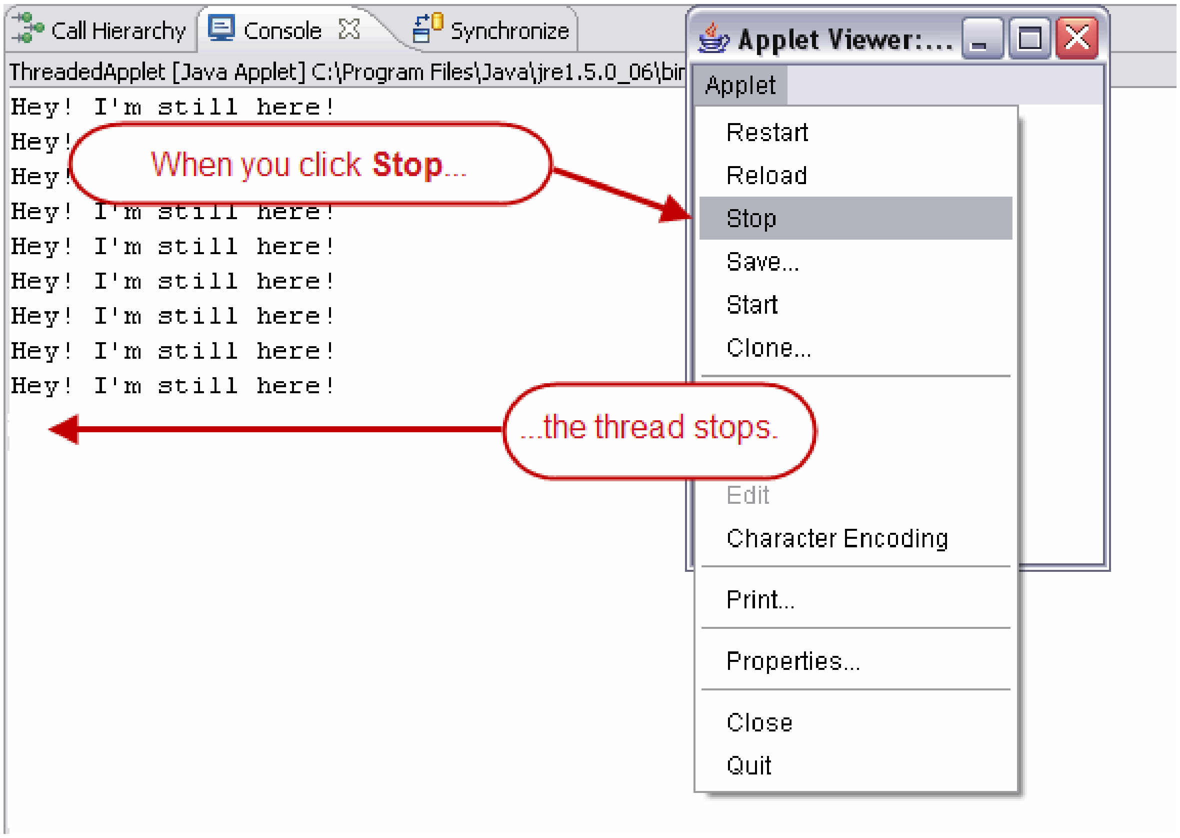
Each oval in the diagram represents a possible *state* of the thread. The arrows represent potential *transitions* among the states. We will give you a new version of our **ThreadedApplet** and comment in the code when the thread is in one of those potential states listed.

Edit **ThreadedApplet** as shown in **blue**:

CODE TO EDIT: ThreadedApplet

```
package demo;

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

public class ThreadedApplet extends Applet implements Runnable {

    Thread appletThread;
    String messages[] = {"Hello Thread World!" , "I'm doing fine." , "Good-bye for now!
"};
    int i = 0;

    public void paint(Graphics g) {
     g.drawString(messages[i], 15, 50);
    }

    public void run() {
        while (appletThread != null) {  // checks if Thread exists
            i = (i + 1) % messages.length;
            repaint();
            System.out.println("Hey! I'm still here");
            try {
                appletThread.sleep(5000);   // sleep--put in Not Runnable State (TIMED_
WAITING)
            } catch (InterruptedException e){ }
        }
    }

    public void start() {             // start of Applet
        if (appletThread == null) {
            appletThread = new Thread(this);  // new Thread()--achieve New Thread State
            appletThread.start();             // start of Thread--achieve Runnable Stat
e
        }
    }

    public void stop() {                      // stop Applet
        appletThread = null;                  // stop Thread by destroying--put in Dead
 State
    }
}
```

Save and run it. Watch a while--at least until it cycles--and then, in the Applet Viewer Window, select **Applet | Stop** to stop it. Now look at the Console for a bit:



ThreadedApplet [Java Applet] C:\Program Files\Java\jre1.5.0_06\bir

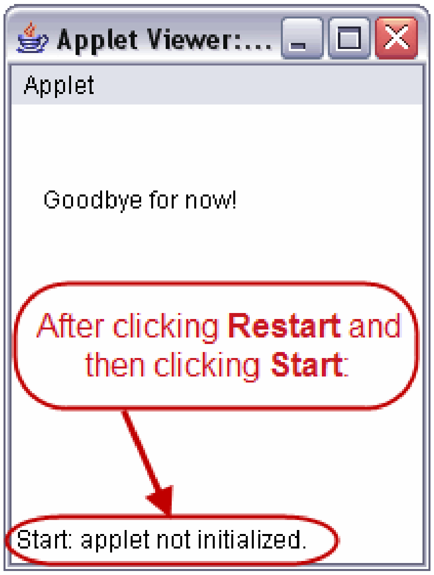
```
Hey! I'm still here!
Hey!
Hey!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
Hey! I'm still here!
```

When you click **Stop**...

...the thread stops.

This time, when our Applet was stopped (no longer painting), its Thread was also stopped (no new output in Console). This is because when we stopped the Applet, we actually killed the thread (we made it **null**), but we aren't done yet!

Restart the Applet (**Applet | Restart**). When we Restart, it calls the **start()** method of the Applet, which starts a new Thread. The thread begins again--both in the Applet's **paint()** method and in the Console. Now, try to Start the Applet (**Applet | Start**). Look at the bottom of the Applet Viewer Window:



Goodbye for now!

After clicking **Restart** and then clicking **Start**:

Start: applet not initialized.

This time, the Applet was already in the start state, as was its Thread. A thread cannot be started with the **start()** method after it has already been started (click here to learn more about **start()** for **Threads**). That's why we checked first to find out if the thread already existed in the **start()** method for the applet. If it did not exist, we would've created it. If it had existed, we wouldn't have had to do anything because it still would've been there, even if the applet had been delayed for a while. Let's look more closely at thread states and see what else is possible.

Thread.State

In order to start the thread, we called a **start()** method for the thread inside of the **start()** method for the applet. So why don't we do the same thing to stop? We wrote a **stop()** method for our applet, so why did we kill the thread (by making it **null**) rather than call a thread **stop()** method?

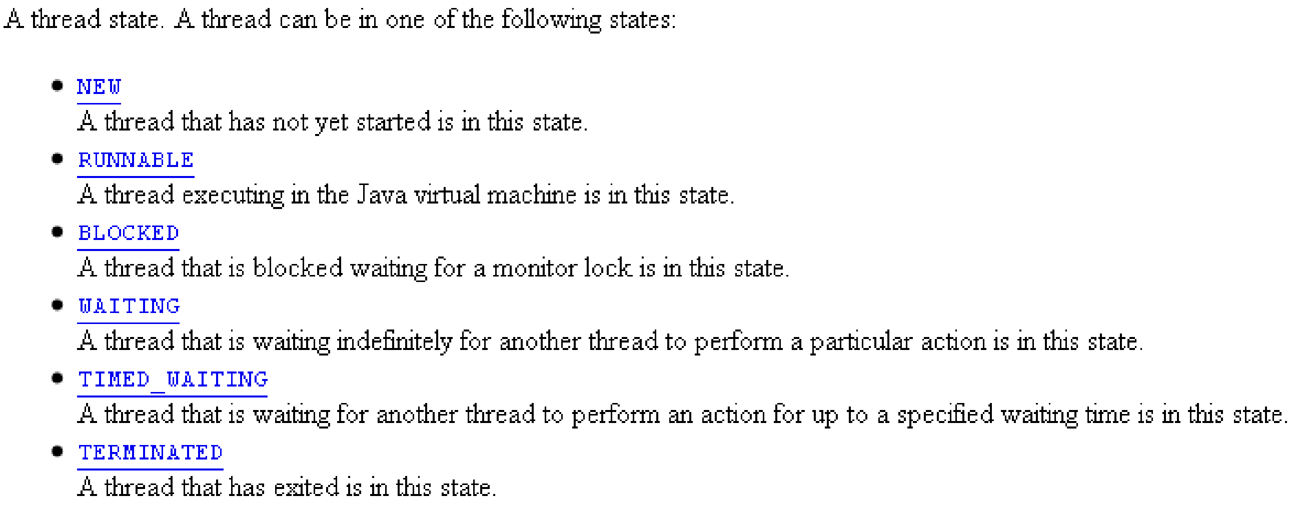
Go to **java.lang.Thread**. Scroll to the **Method Summary**. Check out these methods: **destroy()**, **resume()**, **stop()**, and **suspend()**.

In each of these methods you see the word *deprecated* and a description of the reason. Each new release of Java includes new functionality and fixes. If a method is deprecated, it means that the method should not be used in new code because a problem was found that could not be fixed, *but* that method is retained (perhaps temporarily) in order to maintain compatibility with older versions of Java. There is an inherent problem with the older implementation, so newer versions of Java should not use it. If you are responsible for maintaining code that contains a deprecated method, replace that method if possible.

Each deprecated method points to the Oracle tutorial Java Thread Primitive Deprecation.

Threads was altered in version 1.5 of Java to alleviate problems with deprecated methods. They also want to make sure that users don't encounter problems due to the speed of computers by adding **Enum** States for **Threads**. Let's take a look at those changes.

Go to **java.lang.Thread**. Scroll to the **Nested Class Summary**. Click on **Thread.State**. Scroll to the description of the states:



A thread state. A thread can be in one of the following states:

- NEW
  A thread that has not yet started is in this state.
- RUNNABLE
  A thread executing in the Java virtual machine is in this state.
- BLOCKED
  A thread that is blocked waiting for a monitor lock is in this state.
- WAITING
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- TIMED_WAITING
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- TERMINATED
  A thread that has exited is in this state.

If Oracle created **Enum Constants** in their own **Enum** class **Thread.State**, then **Thread** states must be important. This particular **Thread.State** class is dedicated solely to providing an enumeration of the states and methods that indentify those states.

More on Multi-Threaded Applications

Design Pattern: Producer/Consumer

The design pattern of *Producers and Consumers* used in conjunction with threads is common in Java programming. As is often the case, many producers *supply* a product or resource, and many consumers *take* the product or resource. A *shared resource* is one that many consume. The resource is available at some location (like a store or warehouse), and inventory must be maintained and tracked. Problems may occur when there is not enough supply available to meet the demand, or if the "storehouse" takes in more than it can handle. To avoid such problems, we need a *monitor* to keep inventory of our resource.

We'll include a monitor when we create our application. Our threaded applications will have these three components:

- **Producer**: supplies the resource.
- **Consumer**: uses the resource.

- **Monitor**: keeps a running inventory of the resource.

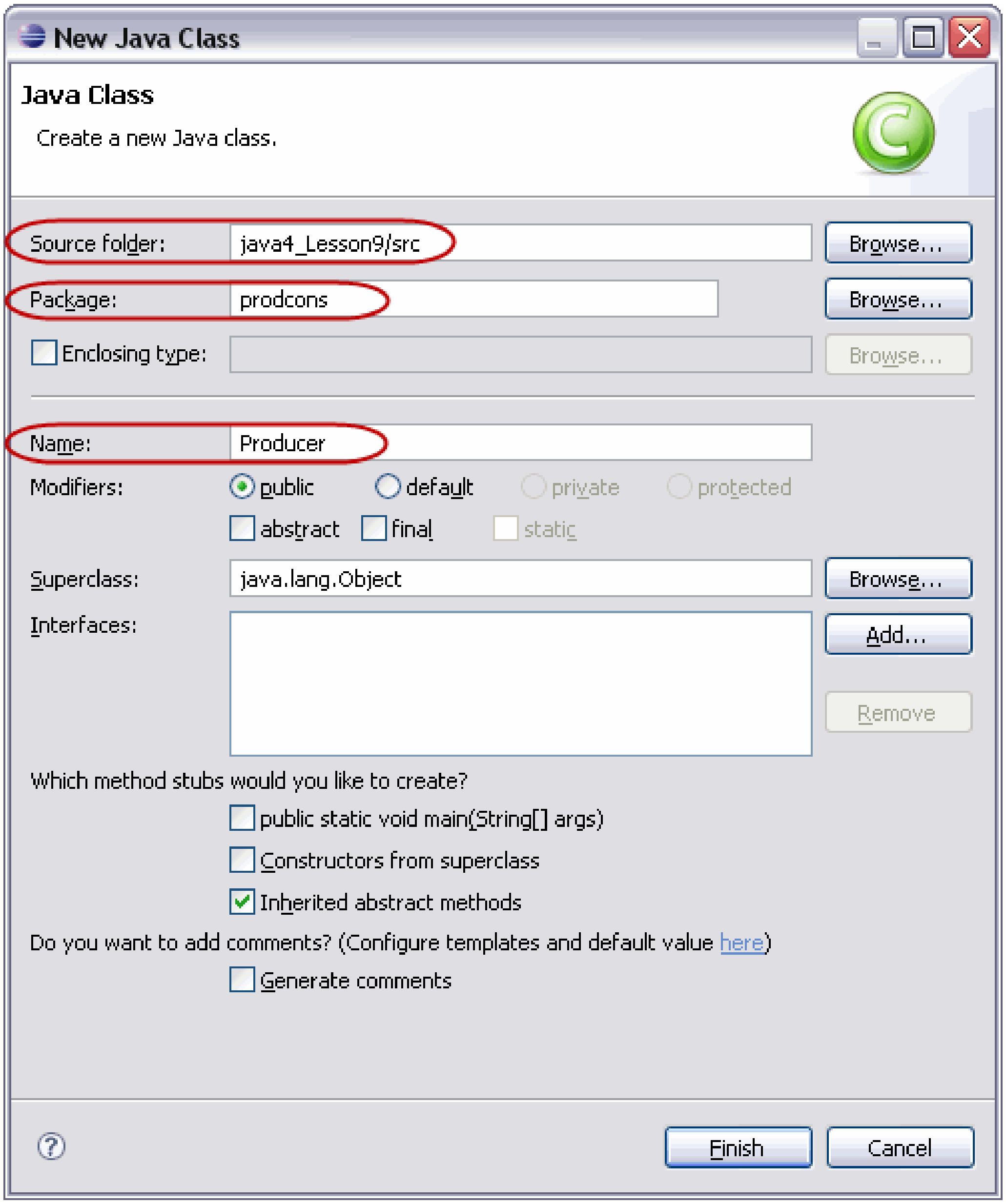
Using Threads in an Application

Our first example using threads will involve alphabet soup. In this example, there is a child who demands that his soup contain more than just broth, so his parent *produces* alphabet letters to add to his soup. The child can then *consume* these letters. The design pattern we described above materializes in our example like this:

- **Producer**: the parent.
- **Consumer**: the child.
- **Monitor**: alphabet soup (stay with me on this).

The Producer

Make a new **java4_Lesson9** project. If you're given the option to "Open Associated Perspective", click **No**. In this project, create a new Class as shown:



Java Class

Create a new Java class.

Source folder: java4_Lesson9/src

Package: prodcons

□ Enclosing type:

Name: Producer

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **Producer** as shown in **blue**:

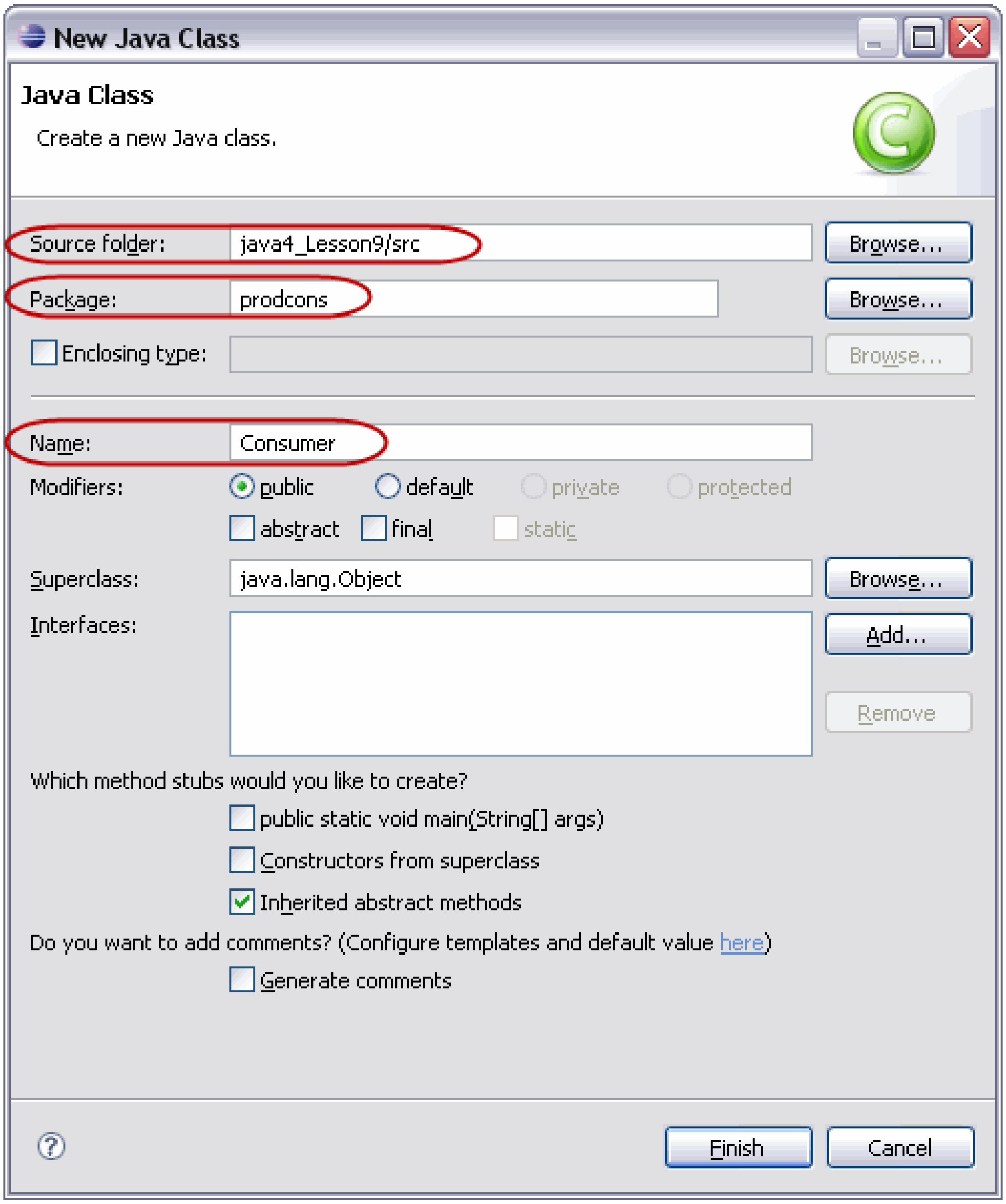
CODE TO TYPE: Producer

```
package prodcons;

class Producer extends Thread {
    private Soup soup;
    private String alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    private MyTableSetting bowlView;

    public Producer(MyTableSetting bowl, Soup s) {
        bowlView = bowl;        // the producer is given the GUI that will show 
what is happening
        soup = s;               // the producer is given the soup--the monitor
    }

    public void run() {
        String c;
        for (int i = 0; i < 10; i++) {                           // only put in 
10 things so it will stop
            c = String.valueOf(alphabet.charAt((int)(Math.random() * 26)));   //
 randomly pick a number to associate with an alphabet letter
            soup.add(c);                                          // add it to
 the soup
            System.out.println("Added " + c + " to the soup.");     // show what
 happened in Console
            bowlView.repaint();                                   // show it i
n the bowl

            try {
                sleep((int)(Math.random() * 2000));  // sleep for a while so it 
is not too fast to see
            } catch (InterruptedException e) { }
        }
    }
}
```

💾 Save it.

The Consumer

In java4_Lesson9, create a new **Consumer** class as shown:



Source folder: java4_Lesson9/src

Package: prodcons

□ Enclosing type:

Name: Consumer

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **Consumer** as shown in **blue**:

CODE TO TYPE: Consumer

```
package prodcons;

class Consumer extends Thread {
    private Soup soup;
    private MyTableSetting bowlView;

    public Consumer(MyTableSetting bowl, Soup s) {
        bowlView = bowl;                               // the consumer is given
the GUI that will show what is happening
        soup = s;                                      // the consumer is given
the soup--the monitor
    }

    public void run() {
        String c;
        for (int i = 0; i < 10; i++) {              // stop thread when know the
re are no more coming; here we know there will only be 10
            c = soup.eat();                            // eat it from the soup
            System.out.println("Ate a letter: " + c);  // show what happened in
Console
            bowlView.repaint();                        // show it in the bowl

            try {
                sleep((int)(Math.random() * 3000));    // have consumer sleep a
little longer or sometimes we never see the alphabets!
            } catch (InterruptedException e) { }
        }
    }
}
```

Save it.

The Monitor

To ensure that shared information within the monitor doesn't become corrupted, we'll *synchronize* the **add** and **remove** methods in this class. Synchronization prevents multiple methods from accessing a shared resource simultaneously. If a method in a class is **synchronized**, it BLOCKs other **Thread**s from accessing any other **synchronized** methods in that instance of that class.

A class with **synchronized** methods provides a lock and prevents what are known as *deadlock* conditions.

Here's how a deadlock condition might look in real life. Let's say we're waiting in line in a bank. I am at the front of the line, waiting to withdraw cash. The bank is out of cash, but I am willing to wait for some cash to be deposited. The bank only has one teller, who cannot handle another transaction until the current transaction is finished. I am still waiting to receive my money, so my transaction is not finished. You are in line behind me with a million dollars to deposit. You can't deposit the money until I finish my transaction, and I will not be finished until someone deposits some money. We are in a *deadlock*.

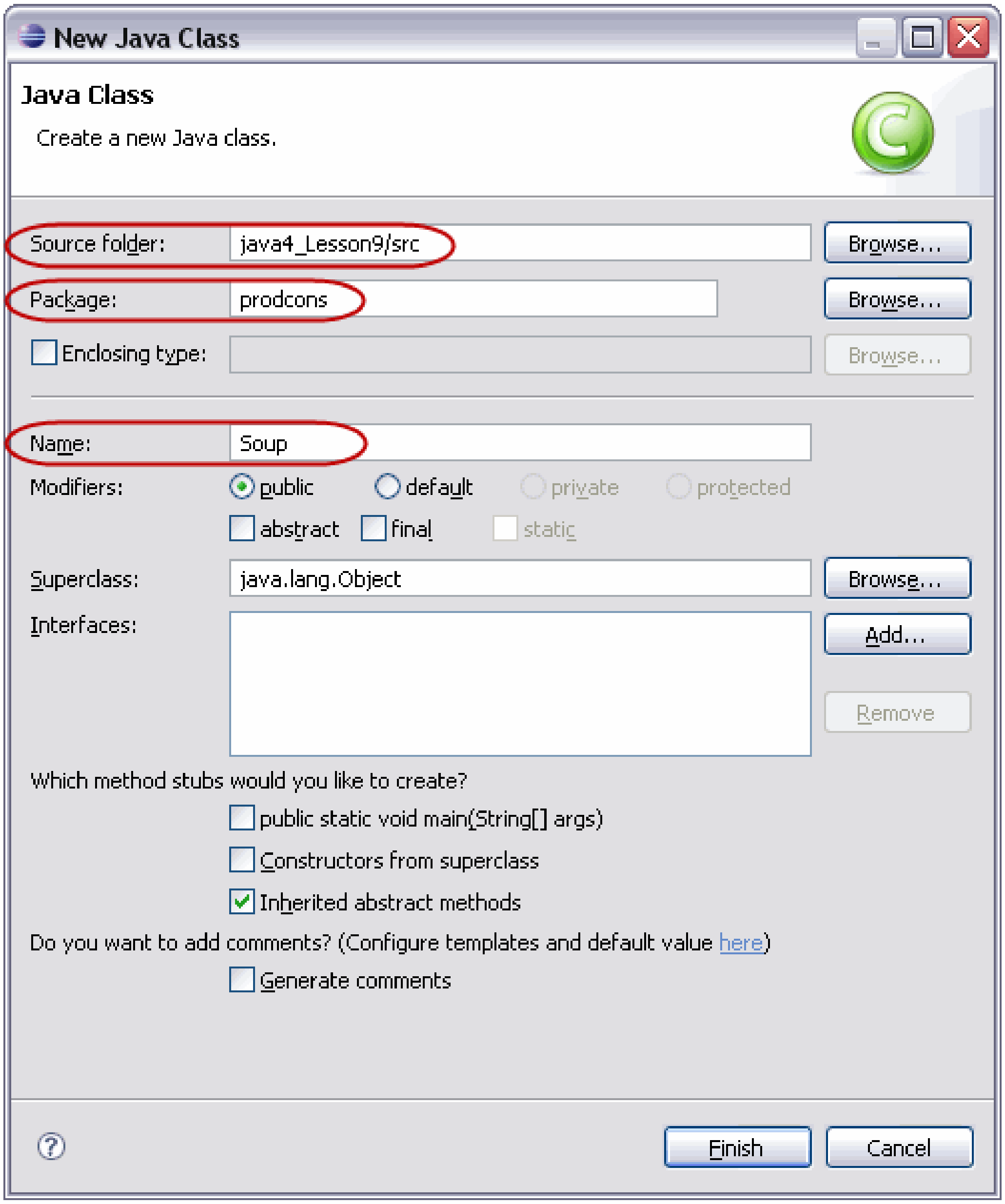
In our example, we'll use **synchronized** blocks of code to prevent deadlock. A few other elements of our example you should also be aware of:

- Only 6 alphabet letters may be in the soup at a given time, so we set the variable **capacity** to 6.
- We'll determine whether there are alphabet letters **in** the soup (the buffer) before we take any out.
- We'll determine whether the soup is full (that is, at it's capacity of 6 letters) before we add any more.

Two important methods for **Thread**s will be used during this process: **notifyAll()** and **wait()**

API Go to **java.lang.Thread** in the API. Scroll down to the **Method Summary**. Look for the methods **notifyAll()** and **wait()**. Remember to look at *all* of the methods for **Threads**--including the ones inherited from **Object**.

In java4_Lesson9, create a new **Soup** class as shown:



Java Class

Create a new Java class.

Source folder: java4_Lesson9/src

Package: prodcons

□ Enclosing type:

Name: Soup

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **Soup** as shown in **blue**:

CODE TO TYPE: Soup

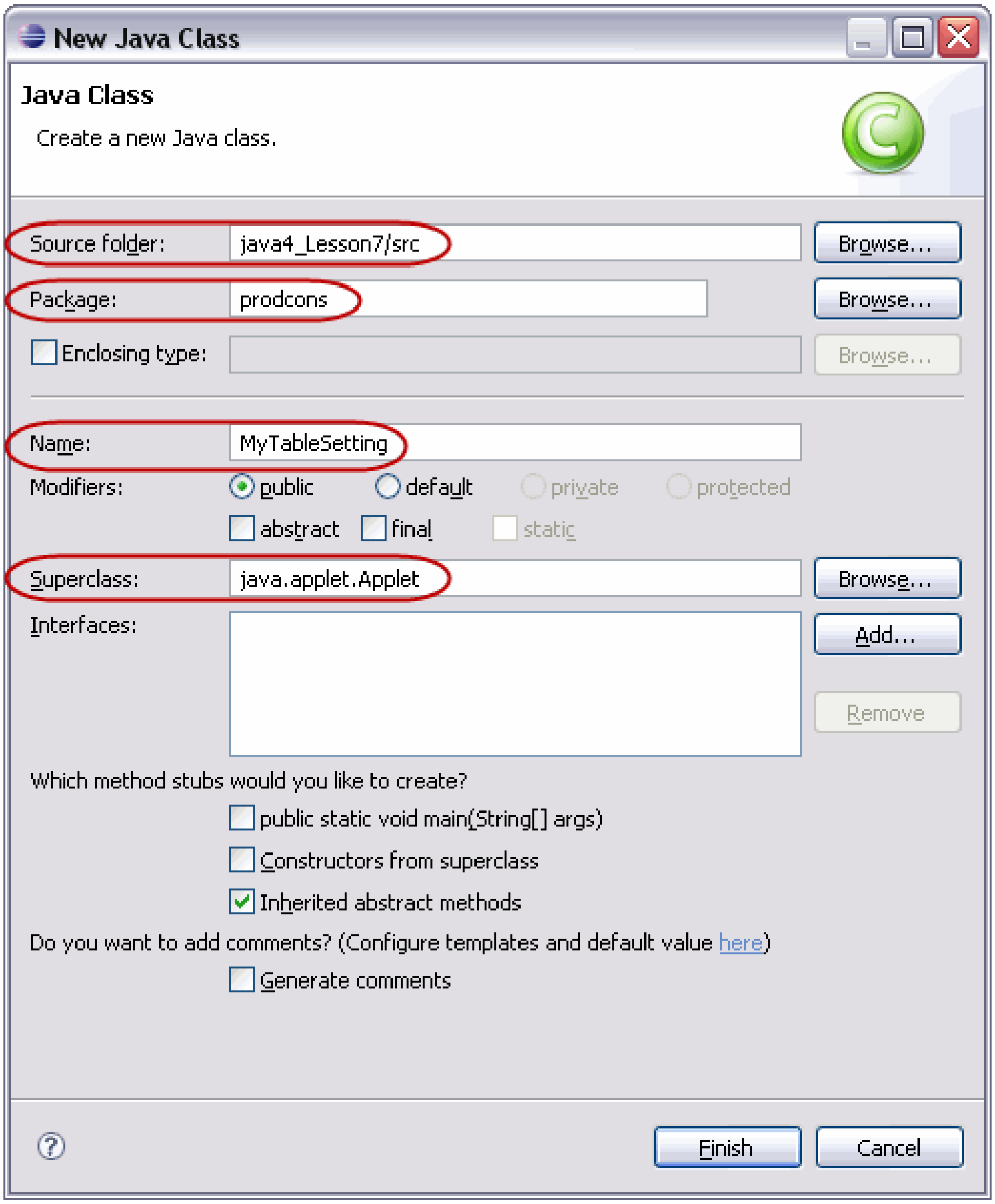
```
package prodcons;

import java.util.*;

public class Soup {
    private ArrayList <String> buffer = new ArrayList<String>();  // buffer hold
s what is in the soup
    private int capacity = 6;                                                  //
 limit to 6 alphabet pieces

    public synchronized String eat() {                                         //
 synchronized makes others BLOCKED
        while(buffer.isEmpty()){                                               //
 cannot eat if nothing is there, so check to see if it is empty

            try {
                wait();                                                        //
 if so, we WAIT until someone puts something there
            } catch (InterruptedException e) {}                                //
 doing so temporarily allows other synchronized methods to run (specifically - a
dd)
        }                                                                      //
 we will not get out of this while until something is there to eat
        String toReturn = buffer.get((int)(Math.random() * buffer.size()));    //
 get a random alphabet in the soup
        buffer.remove(toReturn);                                               //
 remove it so no one else can eat it
        buffer.trimToSize();                                                   //
 reduce the size of the buffer to fit how many pieces are there
        notifyAll();                                                           //
 tell anyone WAITing that we have eaten something and are done
        return(toReturn);                                                      //
 actually return the alphabet piece to the consumer who asked to eat it
    }

    public synchronized void add(String c) {                                   //
 synchronized makes others BLOCKED
        while (buffer.size() == capacity) {                                    //
 cannot add more pieces if the buffer is full to capacity
            try {
                wait();                                                        //
 if so, we WAIT - temporarily allows other synchronized methods to run - i.e., e
at()
            } catch (InterruptedException e) {}
        }                                                                      //
 we will not get out of this while until something has been eaten to make room
        buffer.add(c);                                                         //
 add another alphabet piece to the soup
        notifyAll();                                                           //
 tell anyone WAITing that we have added something and are done
    }

    public ArrayList <String> getContents(){                             // we wa
nt to be able to get the contents so we can show them in the GUI view
        return (buffer);                                                       //
 multiple problems with this - we will address later
    }
}
```

Save it.

The GUI View

Now let's put it all together, using an **Applet** to provide a nice **view** for our users.

In java4_Lesson9, add a new **MyTableSetting** class as shown:



Source folder: java4_Lesson7/src

Package: prodcons

□ Enclosing type:

Name: MyTableSetting

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.applet.Applet

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

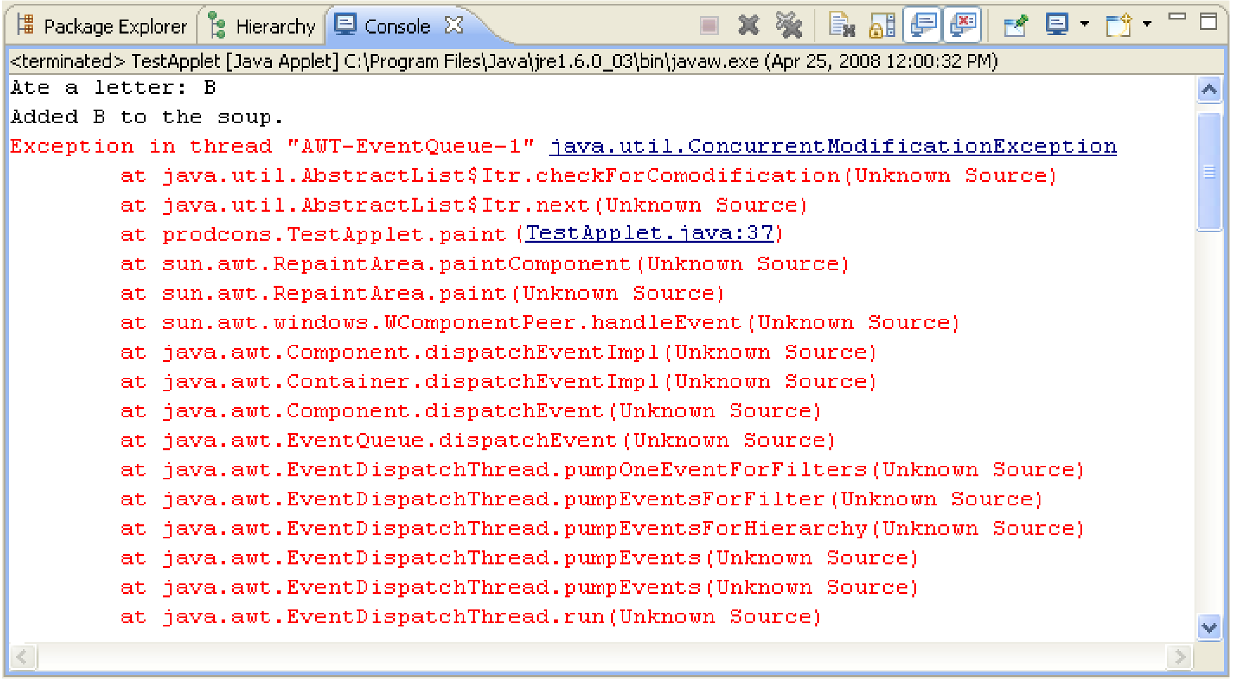
Type **MyTableSetting** as shown in **blue**:

CODE TO TYPE: MyTableSetting

```
package prodcons;

import java.applet.Applet;
import java.util.*;
import java.awt.*;

public class MyTableSetting extends Applet {
    Soup s;                                                        // we will show the soup
 bowl with the soup's alphabet pieces
    int bowlLength = 150;                                          // bowl's dimensions as
variables in case we want to change it
    int bowlWidth = 220;
    int bowlX = 60;
    int bowlY = 10;

    public void init(){
        setSize(400,200);                                            // make the applet siz
e big enough for our soup bowl
        s = new Soup();                                              // instantiate the Sou
p
        Producer p1 = new Producer(this, s);                         // declare and instant
iate one producer thread - state of NEW
        Consumer c1 = new Consumer(this, s);                         // declare and instant
iate one consumer thread - state of NEW

        p1.start();                                                  // start the producer
thread
        c1.start();                                                  // start the consumer
thread
    }

    public void paint(Graphics g){                                   // first we make the b
owl and spoon
        int x;
        int y;
        g.setColor(Color.orange);
        g.fillOval(bowlX, bowlY, bowlWidth, bowlLength);             // the bowl
        g.setColor(Color.cyan);
        g.fillOval(10, 25, 40, 55);                                  // the spoon
        g.fillOval(25, 80, 8, 75);
        g.setColor(Color.black);                                     // black outlines for
the dinnerware
        g.drawOval(10, 25, 40, 55);
        g.drawOval(25, 80, 8, 75);
        g.drawOval(bowlX,bowlY, bowlWidth, bowlLength);
        ArrayList <String> contents = s.getContents();             // get contents of the s
oup
        for (String each: contents){                                 // individually add ea
ch alphabet piece in the soup
            x = bowlX + bowlWidth/4 +(int)(Math.random()* (bowlWidth/2));   // pu
t them at random places to mimic stirring
            y = bowlY + bowlLength/4 + (int)(Math.random()* (bowlLength/2));
            Font bigFont = new Font("Helvetica", Font.BOLD, 20);
            g.setFont(bigFont);
            g.drawString(each, x, y);
        }
    }
}
```

Save and run it. Look at the Console and the *Soup Bowl*. The letters move around because they are being "stirred" as they are eaten.

Run it a few times (be sure to close the applets when you're finished so they don't pile up). If you run the program enough times, eventually you'll have problems:



```
<terminated> TestApplet [Java Applet] C:\Program Files\Java\jre1.6.0_03\bin\javaw.exe (Apr 25, 2008 12:00:32 PM)
Ate a letter: B
Added B to the soup.
Exception in thread "AWT-EventQueue-1" java.util.ConcurrentModificationException
        at java.util.AbstractList$Itr.checkForComodification(Unknown Source)
        at java.util.AbstractList$Itr.next(Unknown Source)
        at prodcons.TestApplet.paint(TestApplet.java:37)
        at sun.awt.RepaintArea.paintComponent(Unknown Source)
        at sun.awt.RepaintArea.paint(Unknown Source)
        at sun.awt.windows.WComponentPeer.handleEvent(Unknown Source)
        at java.awt.Component.dispatchEventImpl(Unknown Source)
        at java.awt.Container.dispatchEventImpl(Unknown Source)
        at java.awt.Component.dispatchEvent(Unknown Source)
        at java.awt.EventQueue.dispatchEvent(Unknown Source)
        at java.awt.EventDispatchThread.pumpOneEventForFilters(Unknown Source)
        at java.awt.EventDispatchThread.pumpEventsForFilter(Unknown Source)
        at java.awt.EventDispatchThread.pumpEventsForHierarchy(Unknown Source)
        at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
        at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
        at java.awt.EventDispatchThread.run(Unknown Source)
```

Play with the **sleep** frequency in the **Producer** and **Consumer** to see how changing that number effects the way the Applet runs.

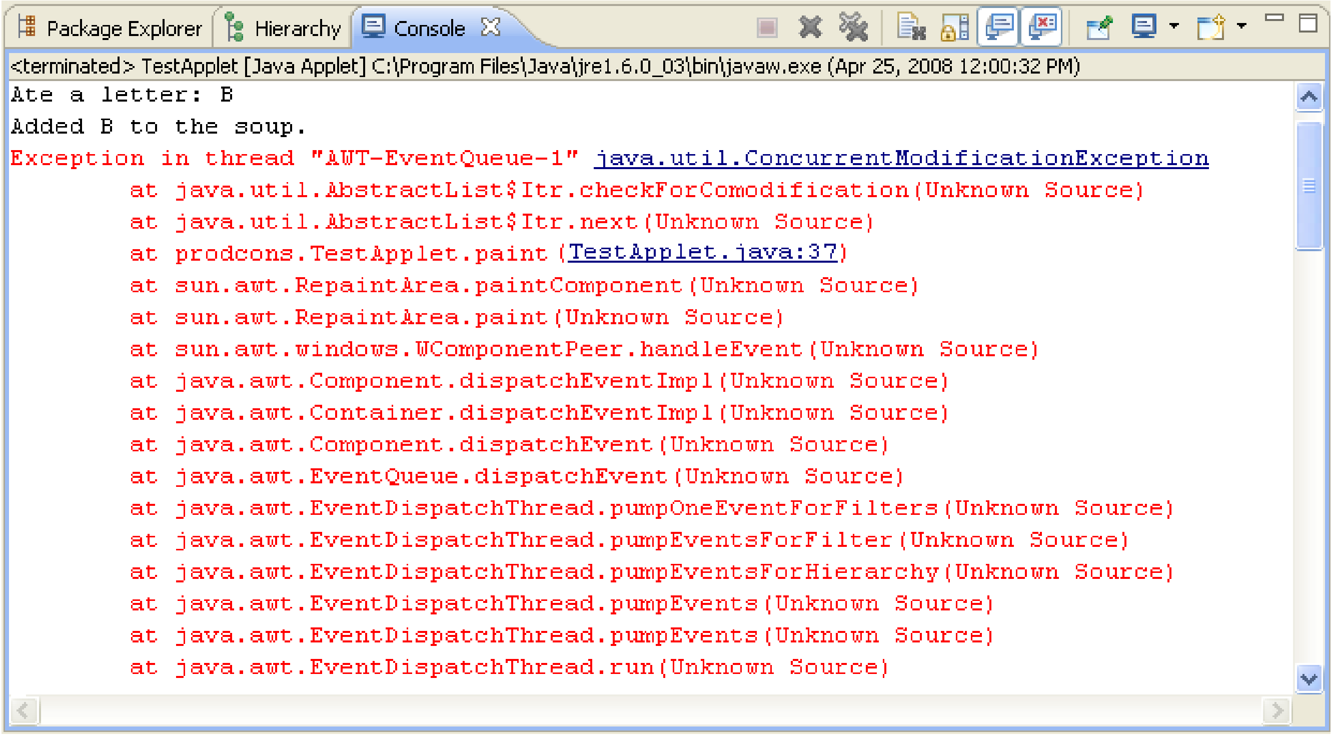
And, for your added enjoyment, here's another simple illustration of producer/consumer/monitor code at work. Enjoy! And see you in the next lesson...

Threads: Synchronization

Race Conditions

In the last lesson we made our first producer/consumer design patterns program. In this lesson, we'll explore synchronization further, expanding on our earlier examples. First we'll edit our Soup program to facilitate a cleaner design.

As you may recall, in that last lesson we left you with a conflict:



```
<terminated> TestApplet [Java Applet] C:\Program Files\Java\jre1.6.0_03\bin\javaw.exe (Apr 25, 2008 12:00:32 PM)
Ate a letter: B
Added B to the soup.
Exception in thread "AWT-EventQueue-1" java.util.ConcurrentModificationException
	at java.util.AbstractList$Itr.checkForComodification(Unknown Source)
	at java.util.AbstractList$Itr.next(Unknown Source)
	at prodcons.TestApplet.paint(TestApplet.java:37)
	at sun.awt.RepaintArea.paintComponent(Unknown Source)
	at sun.awt.RepaintArea.paint(Unknown Source)
	at sun.awt.windows.WComponentPeer.handleEvent(Unknown Source)
	at java.awt.Component.dispatchEventImpl(Unknown Source)
	at java.awt.Container.dispatchEventImpl(Unknown Source)
	at java.awt.Component.dispatchEvent(Unknown Source)
	at java.awt.EventQueue.dispatchEvent(Unknown Source)
	at java.awt.EventDispatchThread.pumpOneEventForFilters(Unknown Source)
	at java.awt.EventDispatchThread.pumpEventsForFilter(Unknown Source)
	at java.awt.EventDispatchThread.pumpEventsForHierarchy(Unknown Source)
	at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
	at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
	at java.awt.EventDispatchThread.run(Unknown Source)
```

Not to worry. We programmers can always find ways to fix a problem! Let's go over the initial text in **red** first:

Exception in thread "AWT-EventQueue-1" java.util.ConcurrentModificationException

Race conditions usually involve one or more processes accessing a shared resource (such as a file or variable), where multiple access is not controlled properly.

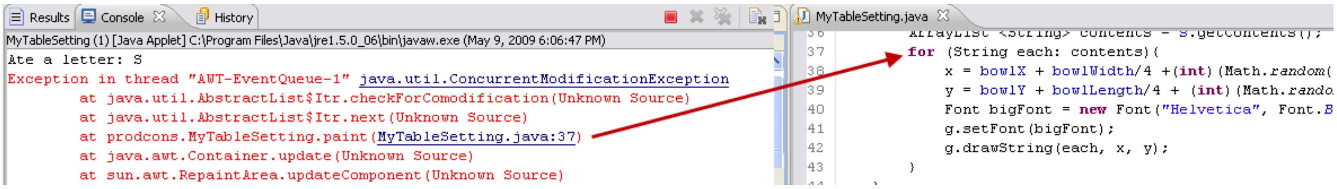
Here's how a race condition might look in real life. Suppose on a given day, a husband and wife both decide to empty the same bank account and, purely by chance, they empty the account at the same time. If the two withdraw from the bank at the exact same time, causing the methods to be called at the exact same time, both ATMs could confirm that the account has enough cash and dispense it. The two threads access the account database at the same time.

The race condition exists here because the actions of *checking the account* and *changing the account balance* are not *atomic*. An *atomic* routine is one that can't be interrupted during its execution. In our banking example, if the actions of checking the account and changing the account balance were atomic, it would be impossible for a second thread to check on the account, until the first thread had finished changing the account status.

To avoid race conditions, we synchronize the **eat()** and **add()** methods. Synchronization prevents race conditions by preventing a second method from running before the first method is complete.

Now let's go back to the error in our Soup example. Our red text informs us of the location of the error:

at prodcons.MyTableSetting.paint (MyTableSetting.java:37)



```
MyTableSetting (1) [Java Applet] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (May 9, 2009 6:06:47 PM)
Ate a letter: S
Exception in thread "AWT-EventQueue-1" java.util.ConcurrentModificationException
        at java.util.AbstractList$Itr.checkForComodification(Unknown Source)
        at java.util.AbstractList$Itr.next(Unknown Source)
        at prodcons.MyTableSetting.paint(MyTableSetting.java:37)
        at java.awt.Container.update(Unknown Source)
        at sun.awt.RepaintArea.updateComponent(Unknown Source)
```

```
36          ArrayList <String> contents = s.getContents();
37          for (String each: contents){
38              x = bowlX + bowlWidth/4 +(int)(Math.random(
39              y = bowlY + bowlLength/4 + (int)(Math.rando
40              Font bigFont = new Font("Helvetica", Font.B
41              g.setFont(bigFont);
42              g.drawString(each, x, y);
43          }
```

It looks like we we did everything right--we got the contents of the **buffer** and put them into a **contents** variable so we could go through the **ArrayList** to print them out. So what's the problem?

Because we are accessing the variable that represents our shared resource in the **getContents()** method of **Soup**, the method should be *synchronized* so it won't be initiated while it is being accessed by another method. If we access and copy our method at the very moment that it is changing, it could cause the **ConcurrentModificationException**. This is an example of a race condition.

And even if we synchronized the method, we would still have problems. Since most collections in Java (for example, arrays and **ArrayList**s) are sent *by reference*, even though we use a method to get the contents and put them into another variable, they still point to the same place in memory. So when we get the contents back to the Applet and then print them out, we continue to access the shared resource in a potentially dangerous way. This is a problem with **ArrayList**s. In order to address these issues, we'll copy the buffer and *then* pass it back.

Fixing a Race Condition

Let's make a few changes to make things a little cleaner. We'll continue working with the classes we created in java4_Lesson9. In the java4_Lesson9/src/prodcons folder, edit the **Consumer** class as shown in **blue**:

CODE TO TYPE: Consumer

```
package prodcons;

class Consumer extends Thread {
    private Soup soup;
    private MyTableSetting bowlView;

    public Consumer(MyTableSetting bowl, Soup s) {
        bowlView = bowl;
        soup = s;
    }

    public void run() {
        String c;
        for (int i = 0; i < 10; i++) {        // stop thread when we know there are no m
ore coming
            c = soup.eat();
            System.out.println("Ate a letter: " + c);
            bowlView.recentlyAte(c);          // tell what alphabet character to put in
 the spoon
            bowlView.repaint();

            try {
                sleep((int)(Math.random() * 3000));  // have consumer sleep a little lo
nger or sometimes we never see them!
            } catch (InterruptedException e) { }
        }
    }
}
```

Save it.

A thread stops when its **run()** method has finished. So in our example, after 10 alphabet letters have been produced and then consumed, their respective threads will stop and as such, they are considered "dead."

In java4_Lesson9/src/prodcons, edit the **Soup** class as shown in **blue**:

CODE TO EDIT: Soup

```
package prodcons;

import java.util.*;

class Soup {
    private ArrayList <String> buffer = new ArrayList<String>();;
    private int capacity = 6;

    public synchronized String eat() {
        while(buffer.isEmpty()){
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        String toReturn = buffer.get((int)(Math.random() * buffer.size()));
        buffer.remove(toReturn);
        buffer.trimToSize();
        notifyAll();
        return(toReturn);
    }

    public synchronized void add(String c) {
        while (buffer.size() == capacity) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        buffer.add(c);
        notifyAll();
    }

    public synchronized Object [] getContents(){  // see ArrayList about ConcurrentModi
ficationException.
        Object [] temp = buffer.toArray();        // check out the API for ArrayList to
 see this toArray() method
        return (temp);                            // Make a clean copy so contents do n
ot change when getting and/or displaying it
    }
}
```

Save it.

In java4_Lesson9/src/prodcons, edit the **MyTableSetting** class as shown in **blue** and **red**:

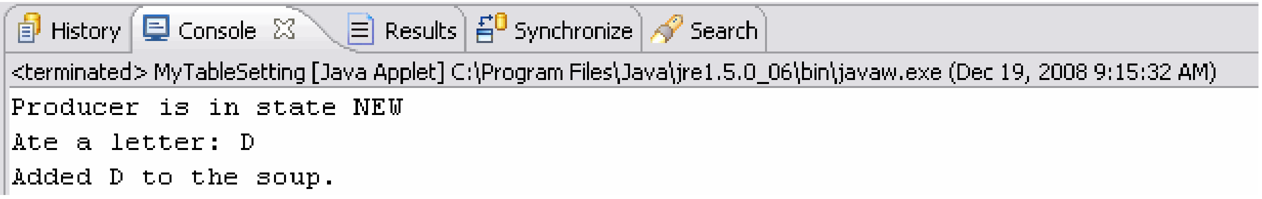
CODE TO EDIT: MyTableSetting

```
package prodcons;

import java.applet.Applet;
//import java.util.*;            // don't need anymore because we have array copy
import java.awt.*;

public class MyTableSetting extends Applet {
    Soup s;
    Producer p1;                // we need as Instance Variables so we can access outsi
de of the init()
    Consumer c1;
    int bowlLength = 150;
    int bowlWidth = 220;
    int bowlX = 60;
    int bowlY = 10;
    String justAte;

    public void init(){
        setSize(400,200);
        s = new Soup();
        p1 = new Producer(this, s);                                   // don't declare
here again or it is only local
        System.out.println("Producer is in state " + p1.getState());   // show the state
 of the thread at this point
        c1 = new Consumer(this, s);

        p1.start();
        c1.start();
        System.out.println("Consumer is in state " + c1.getState());    // show the stat
e of the thread at this point
    }

    public void paint(Graphics g){
        int x;
        int y;

        g.setColor(Color.yellow);
        g.fillOval(bowlX,bowlY, bowlWidth, bowlLength);
        g.setColor(Color.cyan);
        g.fillOval(10,25, 40, 55);
        g.fillOval(25,80, 8, 75);
        g.setColor(Color.black);
        g.drawOval(10,25, 40, 55);
        g.drawOval(25,80, 8, 75);
        g.drawOval(bowlX,bowlY, bowlWidth, bowlLength);
        Font standardFont = getFont();                        // tell what just ate in
spoon
        Font bigFont = new Font("Helvetica", Font.BOLD, 20);
        g.setFont(bigFont);
        if (justAte != null) {
            g.drawString(justAte, 25, 55);
            justAte = null;
        }
        else {
            g.setFont(standardFont);
            g.drawString("waiting", 13, 55);
            g.setFont(bigFont);
        }
        Object [] contents = s.getContents();  // bring back a fresh array of Object
        for(Object each : contents){           // no longer tied in memory to buffer in
 Soup
            x = bowlX + bowlWidth / 4 + (int)(Math.random() * (bowlWidth / 2));
            y = bowlY + bowlLength / 4 + (int)(Math.random() * (bowlLength / 2));
            g.drawString((String)each, x, y);                            // show the al
phabet piece being eaten
        }
        System.out.println("Producer is in state " + p1.getState());   // show state of
 Producer (remember that we put it to sleep)
        System.out.println("Consumer is in state " + c1.getState());
        if(c1.getState()==Thread.State.TIMED_WAITING){                 // note access t
o enumerated types for Thread States
            checkState();                                               // get last rep
aint() in so see TERMINATED
        }
    }

    public void recentlyAte(String morsel){
        justAte = morsel;
    }

    public void checkState(){
        try{Thread.sleep(1000);
        } catch(InterruptedException e) { }                           // Even the Apple
t has a Thread.  This command puts this (Applet's) Thread to sleep
        repaint();
    }
}
```

Save and run it. Look over the output in both the Console and the Applet's bowl and follow the progression of the states. Comment out the **printlns** and run it again without those distractions. You'll see that some states of the **Thread** are accessible through their inner enumerated class **Thread.State**.

Another Race Condition

The **Applet** itself is the main thread and can produce race conditions. When you ran your code, the code sometimes indicated that the first alphabet piece was eaten *before* the piece was even added to the Soup:



```
<terminated> MyTableSetting [Java Applet] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Dec 19, 2008 9:15:32 AM)
Producer is in state NEW
Ate a letter: D
Added D to the soup.
```

This shouldn't happen though, because our program would throw a **NullPointerException** if it had tried to eat something from the buffer that wasn't there yet. So the letter *must* have been in the buffer in order to then have been taken out. Hmm.

Maybe the **System.out.printlns** of **MyTableSetting** were running first and gave us a race condition. The *main* thread (the application or the applet) always takes priority. Fortunately, the class **Thread** contains methods to handle such situations, including **join()**, **wait()**, **sleep()**, **getPriority()**, and **setPriority()**, to name just a few.

Additional Resources

There's lots of material available to help you to work through concurrency and threads. Here's are just a few of them:

- Oracle's tutorial lesson on Threads and Concurrency
- Chapter 17: Threads and Locks (in *Java Language Specification*)
- Of course, the API at java.lang.Thread
- Books specifically on threads, such as this one published by O'Reilly Media: Understanding and Mastering Concurrent Programming: Java Threads, by Scott Oaks and Henry Wong.

Using Threads in a Game

Now let's have some fun. We'll put threads and concurrency together into a game that tests your typing skills. A group of letters will be presented. The object of the game is to type them fast enough to keep a virtual bomb from exploding. You may have five bombs (attempts to type) visible at any given time, but in this version, after three bombs explode, the game is over.

A running example is provided here for you. Play around with it before we write the code, then as you write the code, the classes and their methods will make sense.

Creating the Typing Game

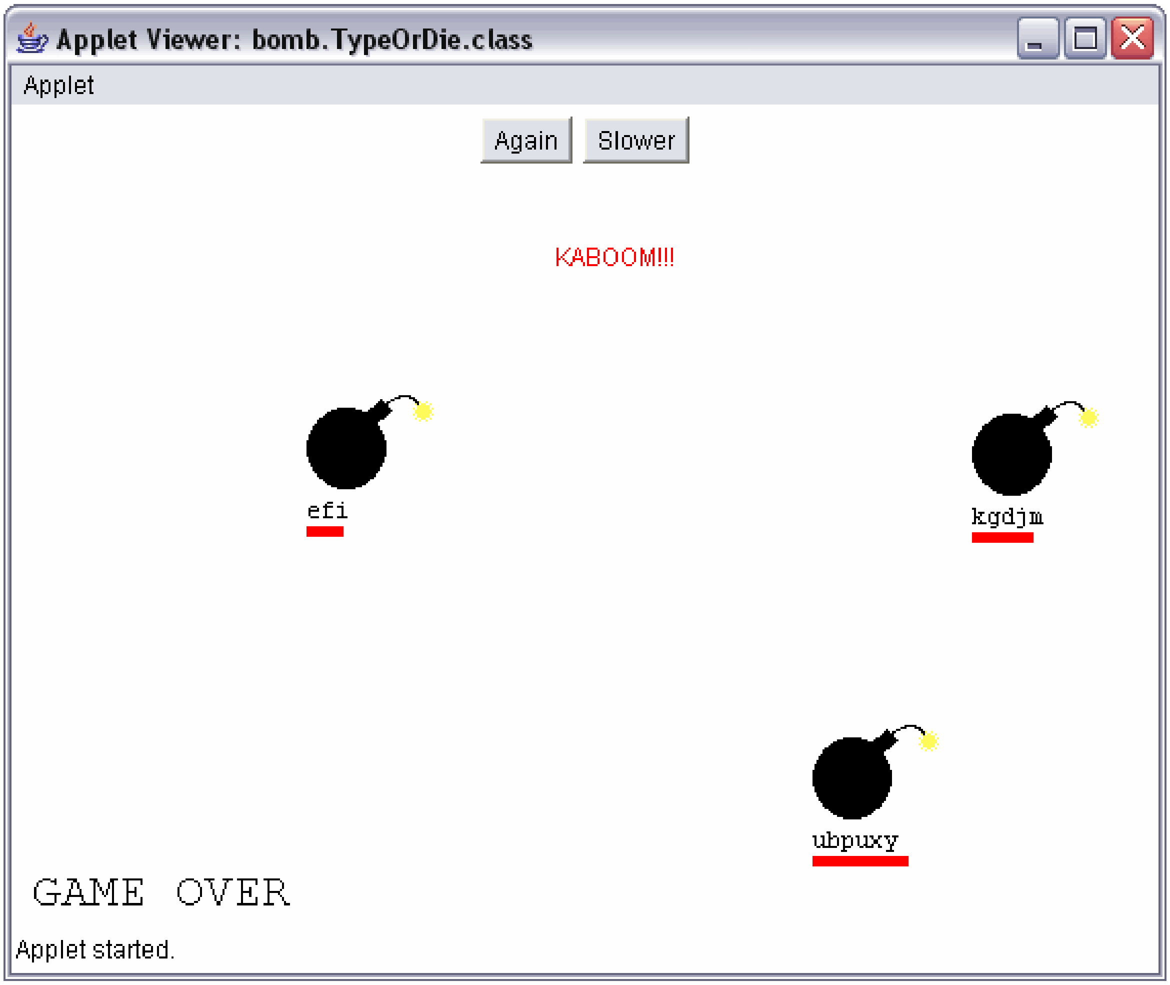
Preview: The Classes

Our game program will include the following classes:

- **Bomb**: extends **Thread**.
- **Producer**: extends **Thread** and produces the letters for players to type.
- **Consumer** (not a **Thread** since in this version we have only one typist on a single keyboard): consumes the letters presented.
- **World**, which is the monitor.
- **TypeorDie**: extends **Applet** and provides the user interface.

The Bomb

Each bomb has a life of its own and is its own thread. Each bomb is associated with the set of letters or *word* presented underneath it and each has its own *fuse* (the red line under the word) that displays the amount of time remaining.



KABOOM!!!

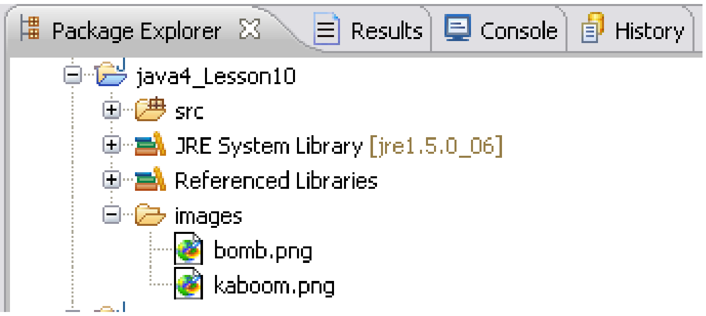
GAME OVER

Applet started.

As you type the code, comments in the code provide information about the reason for each method. Also, the formal parameter specifies **(char c)** for methods that have a **char** passed--this does not mean the *value* is **c**. It just means that **c** is the variable name for the character passed.

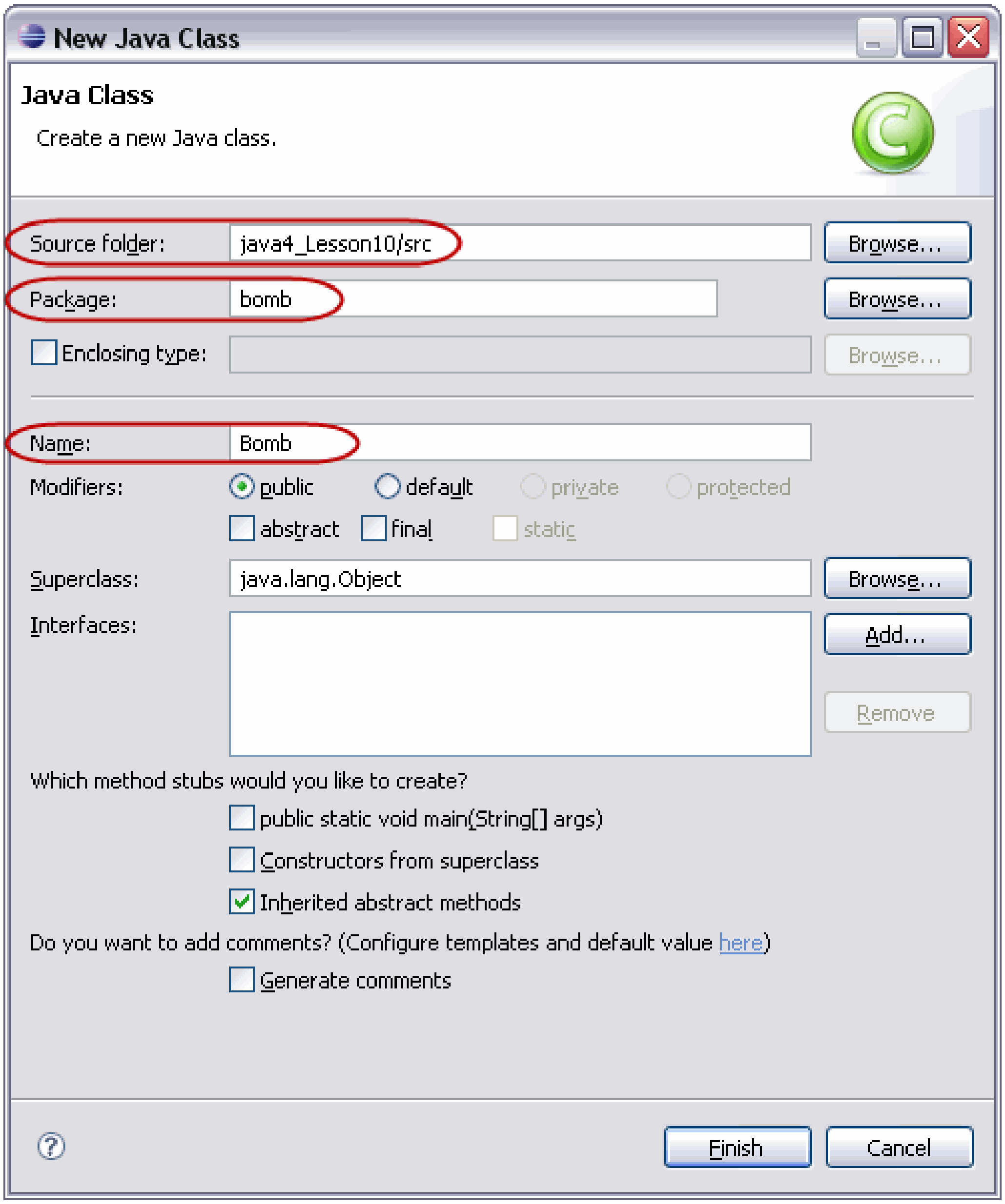
Our java4_Lesson10 project comes with an images folder for this example. Click here, and then open it in the

Package Explorer. It should look like this:



```
java4_Lesson10
    src
    JRE System Library [jre1.5.0_06]
    Referenced Libraries
    images
        bomb.png
        kaboom.png
```

In java4_Lesson10, create a new **Bomb** class as shown:



Source folder: java4_Lesson10/src

Package: bomb

□ Enclosing type:

Name: Bomb

Modifiers: ● public ○ default ○ private ○ protected
□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?
□ public static void main(String[] args)
□ Constructors from superclass
☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)
□ Generate comments

Type **Bomb** as shown in **blue**:

CODE TO TYPE: Bomb

```
package bomb;

import java.awt.*;
import java.applet.Applet;

public class Bomb extends Thread {
    String word;
    int x, y, ticker;
    int width = 62;
    int height = 65;
    Applet apl;
    boolean being_disarmed = false;
    boolean disarmed = false;
    boolean exploded = false;
    int amount_disarmed = 0;
    Image bomb;

    public Bomb(String word, int x, int y, Applet apl){
        super(word);
        this.word = word;
        this.x = x;
        this.y = y;
        this.ticker = word.length()*6;  // time to type is relative to length of
 word
        this.apl = apl;
        bomb = apl.getImage(this.apl.getDocumentBase(), "../images/bomb.png");
    }

    public void run(){
        while (ticker > 0)                                    // have sleep in w
hile loop to show kaboom!!!!!
        {
            try {
                sleep(600);
            }
            catch (InterruptedException e) {}
            ticker--;
            if (disarmed)                                      //check if disarm
ed here
            {
                break;                                          // jump out of w
hile - this bomb is done
            }
            //System.out.println(word+":"+ticker);
            apl.repaint();
        }
        exploded = true;                                    // KABOOM!!! in draw
 method of this class -
    }                                                       // will draw in Grap
hics passed from Applet

    public int getX(){
        return x;                                            // The horizontal c
omponent of the bomb's location is returned
    }

    public int getY(){
        return y;                                            // the vertical com
ponent of the bomb's location is returned
    }

    public int getWidth(){
        return width;                                        // The width of the
 bomb is returned
    }
    public int getHeight(){
        return height;                                       // The height of th
e bomb is returned
    }

    public void draw(Graphics g){                           // The bomb will be
drawn to the Graphics object passed in
        if (!exploded)
        {
            g.drawImage(bomb, x, y,  Color.WHITE, apl);       // not exploded so
 show bomb
            g.setFont(new Font("Monospaced", Font.PLAIN, 12));
            g.setColor(Color.RED);
            g.drawChars(word.toCharArray(), 0, amount_disarmed, x, y+60);  // le
tters user typed turn red
            g.setColor(Color.BLACK);
            if (amount_disarmed != word.length())              // letters not ty
ped stay black
            {
                g.drawChars(word.toCharArray(), amount_disarmed, word.length()-a
mount_disarmed, x+(amount_disarmed*7), y+60);
            }
            if (being_disarmed)
            {
                //System.out.println(word+" is being_disarmed");  // commented o
ut System.outs that helped debug code
                g.setColor(Color.BLUE);                           // word being
"worked on" is circled in blue
                g.drawRoundRect(x-2, y+49, word.length()*9, 14, 10, 10);
            }
                                                              // draw fuse
            g.setColor(Color.RED);
            double bar = (double)ticker/(word.length()*5);
            g.fillRect(x, y+64, (int)(word.length()*7*bar), 5);  // red bar unde
rneath shows progress
        }
        else
        {
            g.setColor(Color.RED);                               // else - bomb
explodes
            g.setFont(new Font("Courier Bold", Font.PLAIN, 12));
            g.drawString("KABOOM!!!", x, y+30);
        }
    }

    public boolean startsWith(char c)   {                     // does the curren
t word start with the value typed (passed in here)
        if (word.charAt(0) == c)
            return true;
        return false;
    }

    public boolean exploded(){
        return exploded;                                       // The Bomb's exp
loded variable will be returned
    }

    public boolean hasPoint(int x, int y){                    // If the Bomb occ
upies the location passed in, a boolean will be returned true
        if ( (this.x <= x && x <= (this.x + this.width)) && (this.y <= y && y <=
 (this.y + this.height)) )
            return true;
        else
            return false;
    }

    public void setdisarming(){
        being_disarmed = true;                                 // The bomb will
be set to being disarmed
    }

    public void setarming(){
        being_disarmed = false;                                     // The bomb will
be set to  not being disarmed
    }

    public boolean attemptDisarm(char c){
        assert amount_disarmed < word.length();                  // assert - another
debugging tool
        if (word.charAt(amount_disarmed) == c)                      // If the Bomb ha
s been totally disarmed i.e. the char passed in was the last char needed to diff
use the bomb,
        {                                                           // then true is r
eturned, otherwise false
            //System.out.println(c+" is a hit on "+word);
            amount_disarmed++;
                                                                  //check if bomb is
totally disarmed
            if (amount_disarmed == word.length())
            {
                //System.out.println(word+" is defused");
                disarmed = true;
                return true;
            }
            return false;
        }
        //System.out.println(c+" is a miss on "+word);
        return false;
    }
}
```

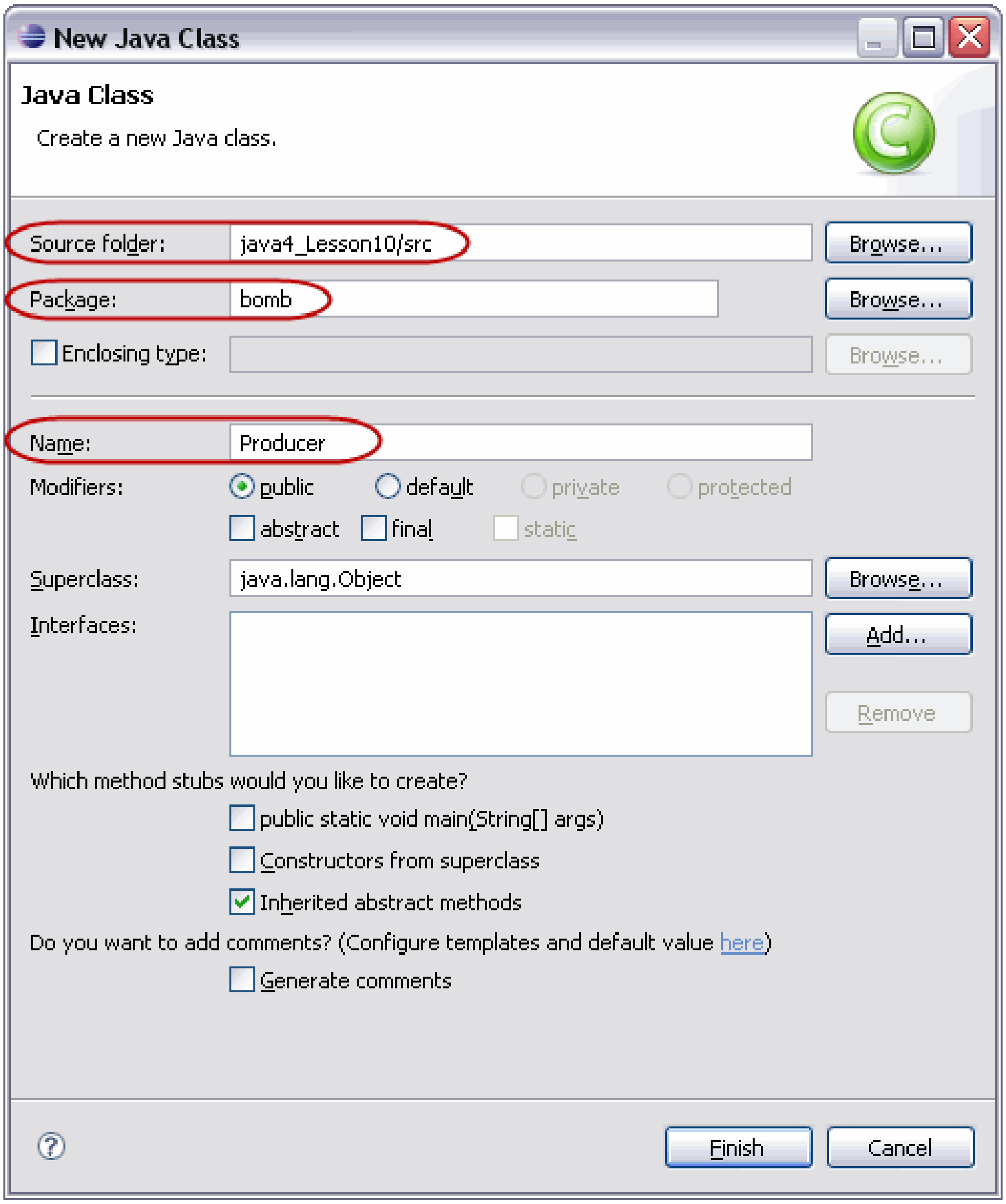
Save it.

Now our code contains the **assert** statement. The **assert** statement was added to Java version 1.4 as a debugging tool. We'll address debugging practices in depth later, but for now, check out Oracle's documentation pages about Programming with Assertions.

Producing Words

The **Producer** in our Bomb game program is also a **Thread** that produces the *words* to type.

In the java4_Lesson10 project, create a new **Producer** class as shown:



Java Class

Create a new Java class.

Source folder: java4_Lesson10/src

Package: bomb

□ Enclosing type:

Name: Producer

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Now type **Producer** as shown in **blue**:

CODE TO TYPE: Producer

```
package bomb;

import java.applet.Applet;

public class Producer extends Thread {
    private World myWorld;
    private String bank = "qwertyuiopasdfghjklzxcvbnm";  // alphabet characters from standard keyboard
    private Applet apl;
    private int bombRate = 2000;                         // rate can be fast or slow

    public Producer(World myWorld, Applet apl) {
        this.myWorld = myWorld;
        this.apl = apl;
    }

    public void toggleBombRate(){                        // user can change speed with GUI button
       if (bombRate == 2000)
           bombRate = 4000;
       else
           bombRate = 2000;
    }

    public void run() {
        String str;
        while (true)
        {
            int length = (int)Math.ceil(Math.random() * 6 );             // random length
            char []str_arry = new char[length];
            for (int i = 0; i < length; i++)
            {
                str_arry[i] = bank.charAt((int)(Math.random() * bank.length() ));   // random placement in string
            }
            str = new String(str_arry);
            //str = bank[((int)(Math.random() * 10))];
            int x = ((int)(Math.random() * 500));                       // random location
            int y = ((int)(Math.random() * 335));
            Bomb b = new Bomb(str, x, y, apl);
            while (myWorld.overlaps(b) )
            {
                b = new Bomb(str, b.getX()+10, y, apl);                  // prevent bomb overlaps
            }
            myWorld.add(b);
            System.out.println("Added bomb " + str + " to the world at "+b.getX()+", "+b.getY() );
            apl.repaint();
            try {
                sleep(bombRate);                                        // put up new words at speed of rate
            }
            catch (InterruptedException e) { }
        }
    }
}
```

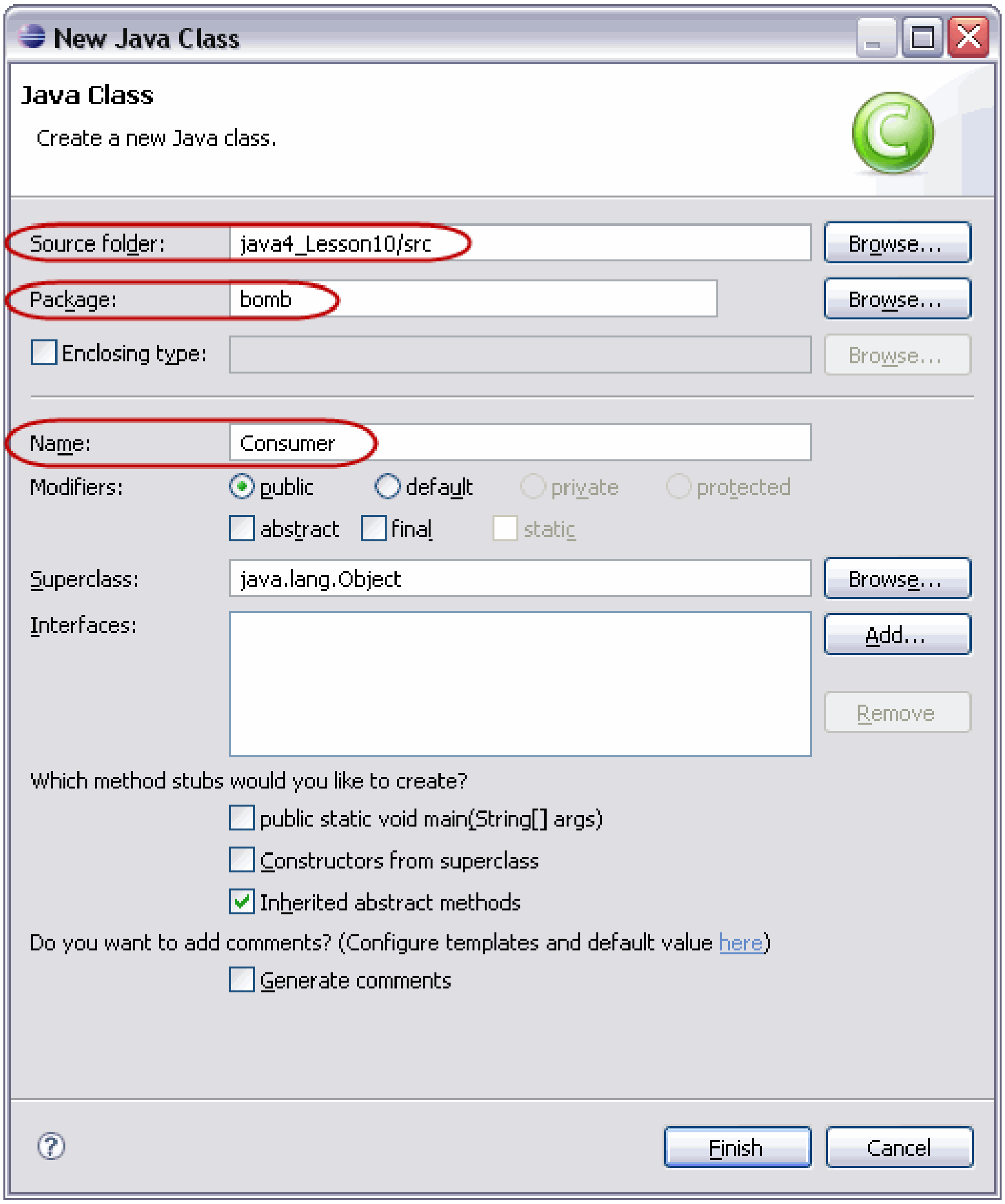
Save it.

Consuming Keys

In our example, *users* are the consumers. As users type, they consume the words that have been placed in the **World**. To allow our users to do this, we include a **KeyListener** class. But since we won't need all of the **KeyListener** methods, we'll also use a **KeyAdapter**.

API Look at the Adapter Class **java.awt.event.Adapter** and the interface that it implements **java.awt.event.KeyListener** to find out which methods are available.

In the java4_Lesson10 project, create a new **Consumer** class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson10/src

Package: bomb

□ Enclosing type:

Name: Consumer

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **Consumer** as shown in **blue**:

CODE TO TYPE: Consumer

```
package bomb;

import java.applet.Applet;
import java.awt.event.*;

public class Consumer extends KeyAdapter {
    private World myWorld;
    private Applet apl;

    public Consumer(World myWorld, Applet apl) {
        this.myWorld = myWorld;
        this.apl = apl;
        apl.addKeyListener(this);
    }

    public void keyTyped(KeyEvent e) {
        myWorld.type(e.getKeyChar() );
        apl.repaint();
    }
}
```

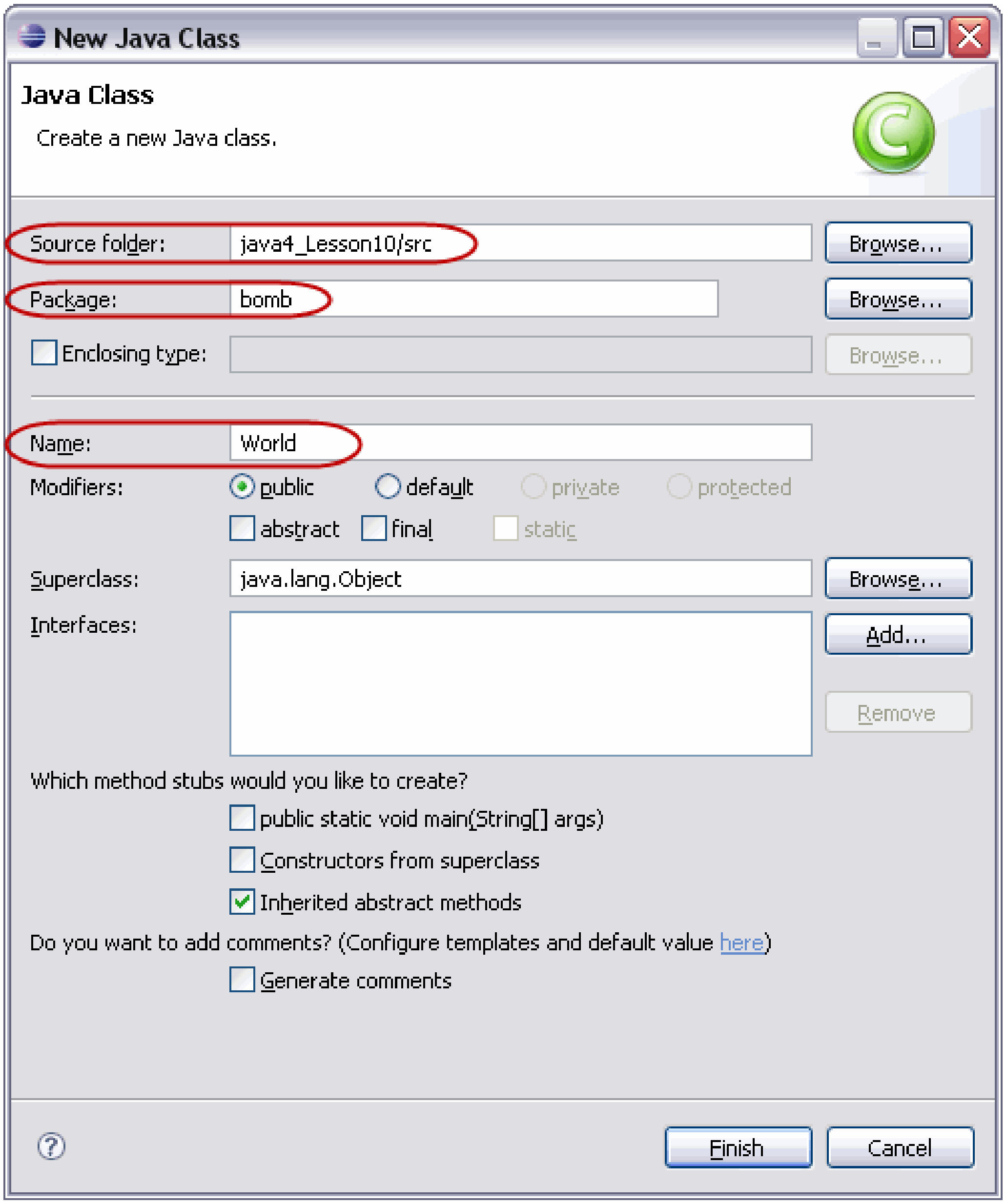
Save it.

Monitoring Words

The monitor in this example watches over all of the words presented, as well as the user's typing speed and accuracy. Each new *word* has a bomb associated with it. This monitor considers the consumption of the words in two ways:

1. Only five *words* are presented at one time.
2. If a word is not typed fast enough, its bomb explodes. If three bombs explode, the monitor stops paying attention to the user.

In the java4_Lesson10 project, create a new **World** class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson10/src

Package: bomb

□ Enclosing type:

Name: World

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **World** as shown in **blue**:

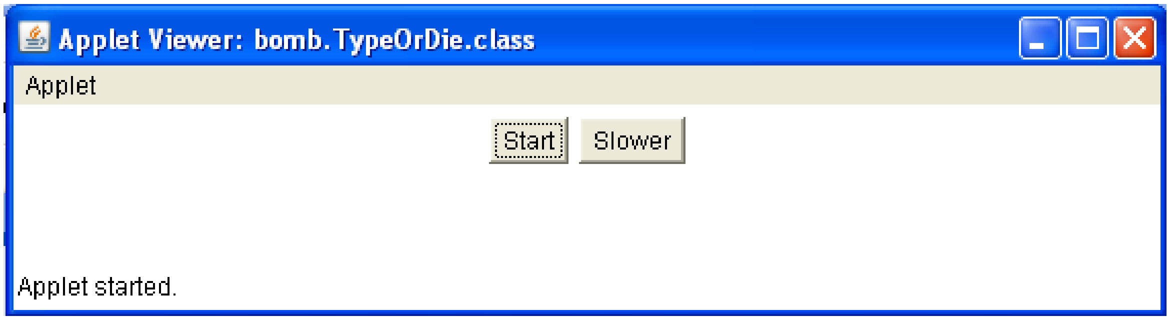
CODE TO TYPE: World

```
package bomb;

import java.awt.*;

public class World {
    private final int MAX_BOMBS = 5;
    private Bomb bombs[] = new Bomb[MAX_BOMBS];  // shared resource - 5 bombs shown at a time
    private int typeNext = -1;
    private int addNext = 0;
    private int num_bombs = 0;
    private boolean isFull = false;
    private boolean isEmpty = true;
    private boolean gameOver = false;

    public synchronized void type(char c) {
        while (isEmpty == true)                   // if no words/bombs in buffer, wait for some
        {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
                                                  // check if three bombs have already blown up
        int num_exploded = 0;
        for(int i = 0; i < MAX_BOMBS; i++)     // check bombs in current value of shared resource
        {
            if (bombs[i] != null && bombs[i].exploded)
                num_exploded++;
        }
        if (num_exploded >= 3)
        {
            gameOver = true;
            return;
        }
                                                             // check if entered char matches a bomb and if its fully disarmed
        if (typeNext < 0 || (bombs[typeNext].exploded) )   // no current diffusing bomb or current bomb blew up
        {
            for(int i = 0;i<MAX_BOMBS; i++)
            {
                if (bombs[i] != null && !bombs[i].exploded && bombs[i].startsWith(c) )
                {
                     typeNext = i;
                     bombs[typeNext].setdisarming();
                     break;
                }
            }
        }
        if (typeNext > -1 && !bombs[typeNext].exploded && bombs[typeNext].attemptDisarm(c) )
        {
            bombs[typeNext] = null;
            num_bombs--;
            typeNext = -1;
            if (num_bombs == 0 )
            {
                isEmpty = true;
            }
            isFull=false;
            notifyAll();
        }
    }

    public synchronized boolean overlaps(Bomb b){  // cannot put bombs on top of
 each other
       for (int i = 0; i<MAX_BOMBS; i++)
       {
           if (bombs[i] != null && (bombs[i].hasPoint(b.getX(), b.getY()) ||
            bombs[i].hasPoint(b.getX()+b.getWidth(), b.getY()) ||
            bombs[i].hasPoint(b.getX(), b.getY()+b.getHeight()) ||
            bombs[i].hasPoint(b.getX()+b.getWidth(), b.getY()+b.getHeight()) ) )
           {
               return true;
           }
       }
       return false;
    }

    public synchronized void clearBombs(){   // clear bombs to reset
        for (int i = 0; i<MAX_BOMBS; i++)
        {
            bombs[i] = null;
        }
        typeNext = -1;
        addNext = 0;
        num_bombs = 0;
        isFull = false;
        isEmpty = true;
        gameOver = false;
        notifyAll();
    }

    public synchronized void add(Bomb b) {  // add a bomb
        while (isFull == true)              // cannot add if full
        {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        //check for empty bomb spot
        int i;
        for (i = 0; i<MAX_BOMBS; i++)
        {
            if (bombs[i] == null)           // find empty space in array
            {
                bombs[i] = b;
                break;                      // once find space, get out of for l
oop
            }
        }
        assert bombs[i] == b;
        num_bombs++;
        bombs[i].start();                   // light the fuse baby!

        if (num_bombs == MAX_BOMBS)
        {
            isFull = true;
        }
        isEmpty = false;
        notifyAll();
    }

    public void draw(Graphics g){           // draw all of the bombs--called fro
m Applet
        for (int i=0; i<MAX_BOMBS; i++)
        {
            if (bombs[i] != null)
                bombs[i].draw(g);
        }
        if (gameOver)
        {
            g.setColor(Color.BLACK);
            g.setFont(new Font("Monospaced", Font.PLAIN, 23));
            g.drawString("GAME OVER", 10, 390);
        }
    }
}
```

Save it.

The User Interface

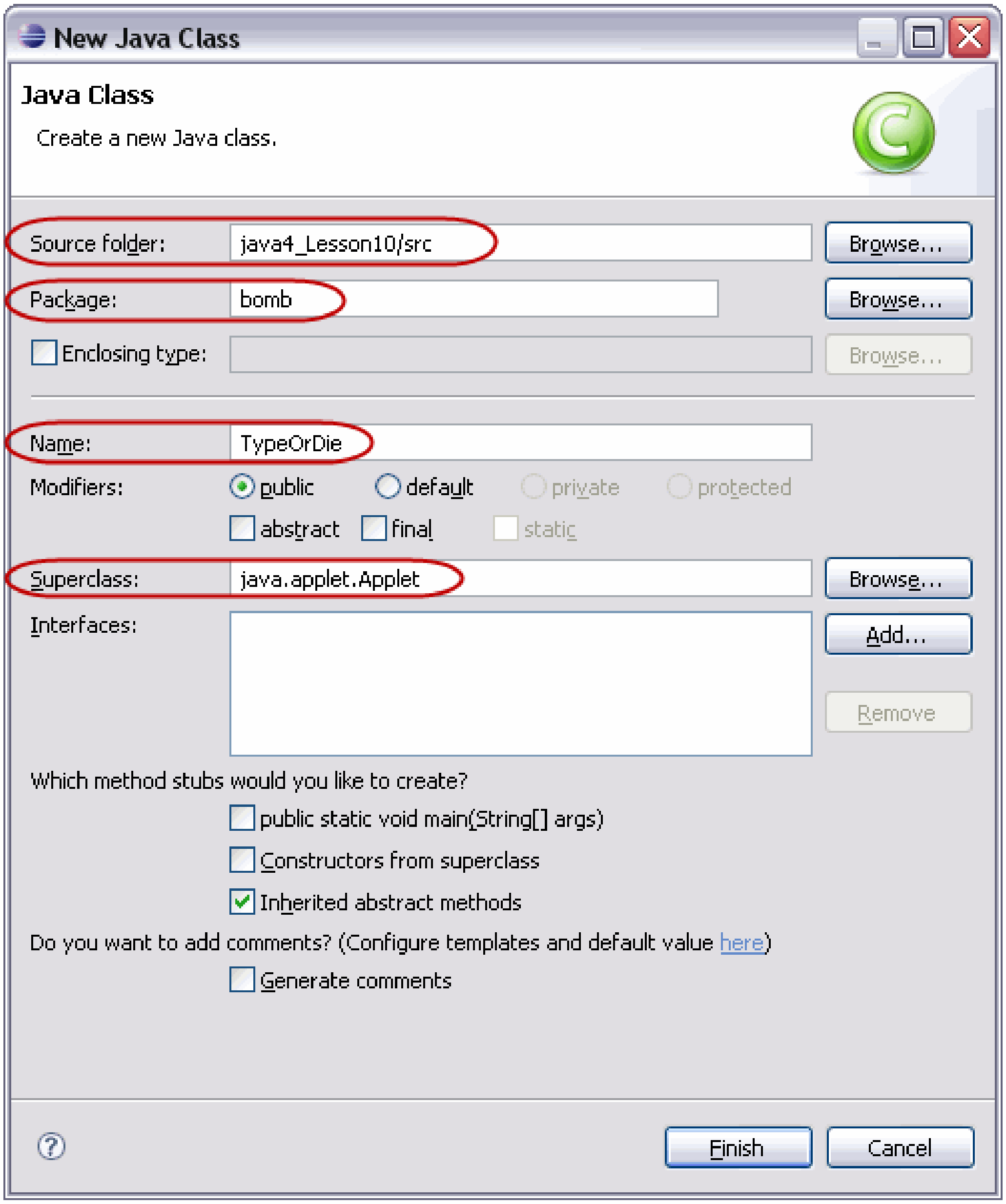
The graphical user interface for our example is relatively straightforward:



Applet started.

Double-buffering, specifically a Double-Buffer Applet, was used here to create the GUI.

In the java4_Lesson10 project, create a new **TypeOrDie** class as shown:



Java Class

Create a new Java class.

Source folder: java4_Lesson10/src

Package: bomb

□ Enclosing type:

Name: TypeOrDie

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.applet.Applet

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

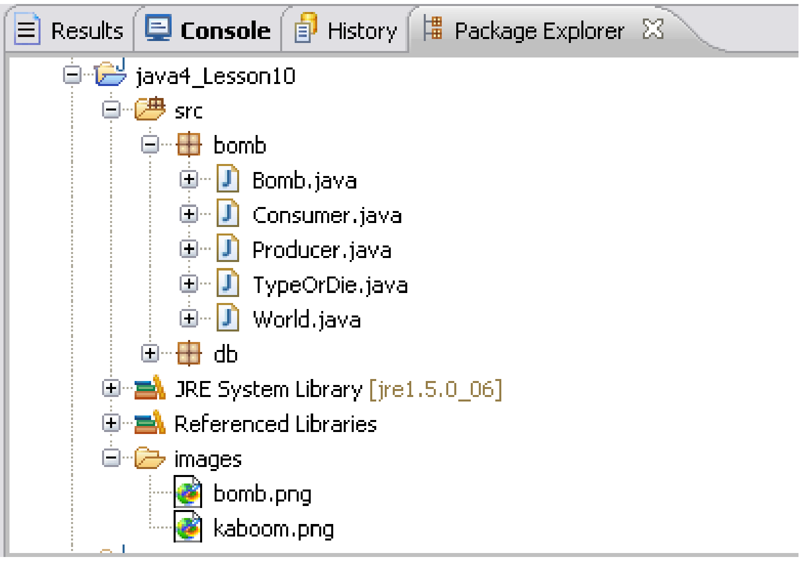
□ Generate comments

Type **TypeorDie** as shown in **blue**:

CODE TO TYPE: TypeOrDie

```
package bomb;

import java.awt.event.*;
import java.awt.*;
import java.applet.Applet;

public class TypeOrDie extends Applet implements ActionListener {
    World myWorld;
    Button start_btn;
    Button slow_fast_btn;
    boolean started = false;
    Producer p1;
    Consumer c1;

    public void init(){
        setSize(560,400);

        start_btn = new Button("Start");    // add the buttons and listeners
        slow_fast_btn = new Button("Slower");
        start_btn.addActionListener(this);
        slow_fast_btn.addActionListener(this);
        this.add(start_btn);
        this.add(slow_fast_btn);
        myWorld = new World();              // instantiate everyone
        p1 = new Producer(myWorld, this);
        c1 = new Consumer(myWorld, this);
    }

    public void paint(Graphics g) {
        Dimension dim = getSize();          // set up double buffer
        Image offscreen = createImage(dim.width, dim.height);
        Graphics bufferGraphics = offscreen.getGraphics();
        bufferGraphics.clearRect(0,0, dim.width, dim.height);

        myWorld.draw(bufferGraphics);

        g.drawImage(offscreen, 0, 0, this);
    }

    public void update (Graphics g){
        paint(g);
    }

    public void actionPerformed(ActionEvent e){
        if (e.getActionCommand() == "Slower")
        {
            p1.toggleBombRate();
            slow_fast_btn.setLabel("Faster");
        }
        else if(e.getActionCommand() == "Faster")
        {
            p1.toggleBombRate();
            slow_fast_btn.setLabel("Slower");
        }
        else
        {
            myWorld.clearBombs();
            if (!started)                    // start the word creation
            {
                p1.start();
                started = true;
                start_btn.setLabel("Again");
            }
        }
        this.requestFocus();
    }
}
```

Now you see this in the Package Explorer:



```
java4_Lesson10
  src
    bomb
      Bomb.java
      Consumer.java
      Producer.java
      TypeOrDie.java
      World.java
    db
  JRE System Library [jre1.5.0_06]
  Referenced Libraries
  images
    bomb.png
    kaboom.png
```

Save and run it. Alternate between running the code and reading what each object does so you understand the reasons behind the implementation.

We Love Threads

Threads are a challenge, but with practice they'll serve you well. Check out these implementations of threads:

Real Games

You'll find lots of real games powered by Java here.

Blackjack

Blackjack is one of the most popular casino games in the world. Given our knowledge of threads, and the card capabilities we acquired in the last course, we could create our own blackjack game that allows multiple players.

Now suppose you have two or three players spread across the internet and they all choose to be *Hit* at the same time. You *could* have a race condition, and you *could* corrupt the game by giving all of the players the same card! Having the game set up so that multiple players can each play on their own separate thread will allow your game to go off without a hitch. Successful java programming--it's all about the threads.

You're doing great so far, keep it up!

Databases: Connectivity through Java

The JDBC API

The next few lessons focus on connecting Java to databases to access information.

What is JDBC?

JDBC is a Java API associated with accessing tabular data; that is, data that you'd generally want to record in a table. A table is a single store of related information; a database can consist of one or more tables of information that are related in some way. JDBC is used to process SQL statements and enable your programs to access relational databases. Like other Java APIs, the JDBC API consists of classes and interfaces written in the Java programming language. Their main purpose is to provide a standard API for database developers and make it possible to write database applications using a pure Java API, which in turn ensures that your code will be portable.

API Go to the **java.sql** package to see some of the available classes and interfaces. Skim through the Package java.sql Description and its history. Go to the **javax.sql** package to see some of the extended server-side functionality that its classes and interfaces provide. This course focuses on the **java.sql** package. (The server-side information in **javax.sql** applies in the J2EE course in this series.)

What Does JDBC Help Us Do?

JDBC helps us to write platform-independent Java programs. These programs can be used to connect to a wide range of SQL databases and manipulate the data without modifying and/or recompiling the Java programs, even when moving from platform to platform or from DBMS to DBMS. In short, JDBC makes it possible to:

1. Establish a connection with a database.
2. Send SQL statements.
3. Process the results.

Although JDBC provides many classes and interfaces for use with databases, those three are the most commonly used. Here we've set up a table of the common JDBC tasks and their corresponding classes or interfaces:

Task	Most Used Class or Interface
establish a connection with a database	**java.sql.Connection**
send SQL statements	**java.sql.Statement**
process the results	**java.sql.ResultSet**

We'll illustrate each of these tasks in an example.

Connecting to the Database

We can't do any work with the database until we connect to it.

Access to a Database

SQL can communicate with most database systems without changing the SQL commands. The MySQL database server is likely the most popular open source database software among programmers and is available for use on a wide variety of platforms. We'll use the MySQL database for our examples.

In this course, you have been granted access to the MySQL database on the O'Reilly School of Technology server. You can access that database using the same username and password that you use to log onto your courses.

Note OST uses a master password (and Sandbox login) for all login instances. In some cases, the password for MySQL can become out of sync and cause this or a similar error:

ERROR 1045 (28000): Access denied for user '<Sandbox Login>'@'cold1.useractive.com' (using password: YES)

This error will also appear if the login or password is incorrect. However, for OST, it is most likely a “password out-of-sync” issue.

To fix this, update the password for the OST Sandbox from the My Account section of the Student Start Page (students.oreillyschool.com/student/). This will reset the password for all login instances, including MySQL. You can use your current password without changing it. Remember that the Sandbox login and password are case sensitive.

Even though SQL is standardized, each database vendor has a different interface, as well as different extensions of SQL. Our notes and examples here will be all-purpose so you can use them on this, as well as other databases.

The Driver

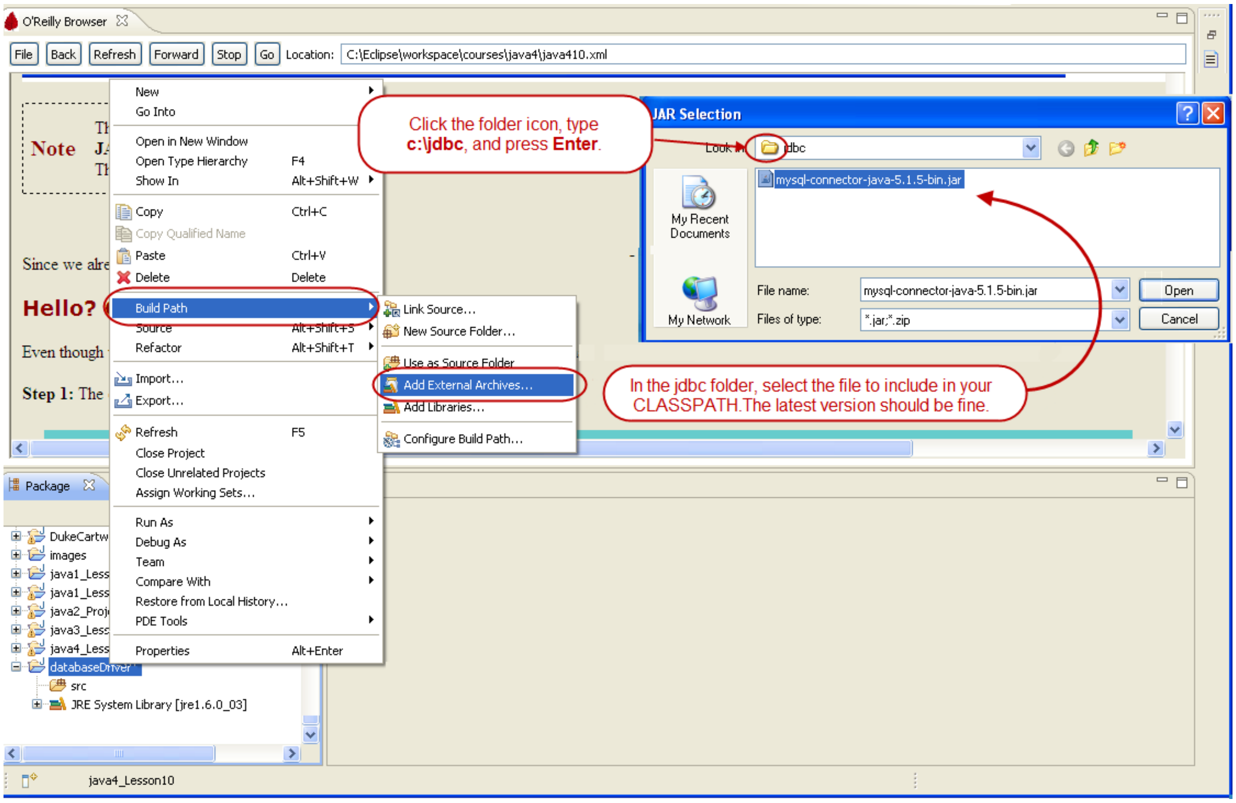
Programmers use many different databases. Each database needs a *driver* to connect it to the Java program.

There are four types of JDBC drivers. We will use a Type 4 driver, because it will communicate directly with our data source. Let's go ahead and start the Project and get the driver. Create a new **databaseDriver** project for this lesson.

Note The *driver* is actually a collection of Java classes that assist connectivity that are packaged into one *.jar* (Java ARchive) file, similar to a .zip or .tar file, used to hold multiple files. The Java VM can get into .jar files for classes and files it needs if the .jar is in the CLASSPATH.

Put the driver into the CLASSPATH for your project. To do this, download the appropriate driver (we've already done this for you), then right-click the **databaseDriver** project, choose **Build Path | Add External Archives**, browse to the **C:\jdbc** folder and select the **mysql-connector-java-5.1.5-bin.jar** file:

Note You might find a newer version of the JDBC driver in this folder. The newer versions should be backward compatible with the versions referenced in this lesson.



Location: C:\Eclipse\workspace\courses\java4\java410.xml

Click the folder icon, type **c:\jdbc**, and press **Enter**.

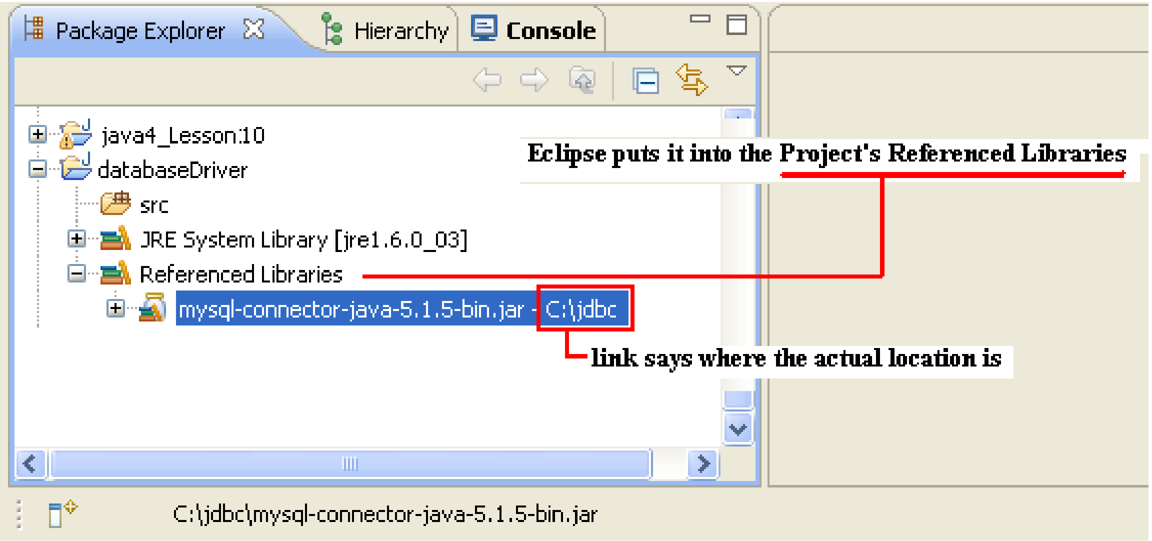
Look in: jdbc

File name: mysql-connector-java-5.1.5-bin.jar

Files of type: *.jar;*.zip

In the jdbc folder, select the file to include in your CLASSPATH. The latest version should be fine.

Then you'll see this file:



Eclipse puts it into the Project's Referenced Libraries

link says where the actual location is

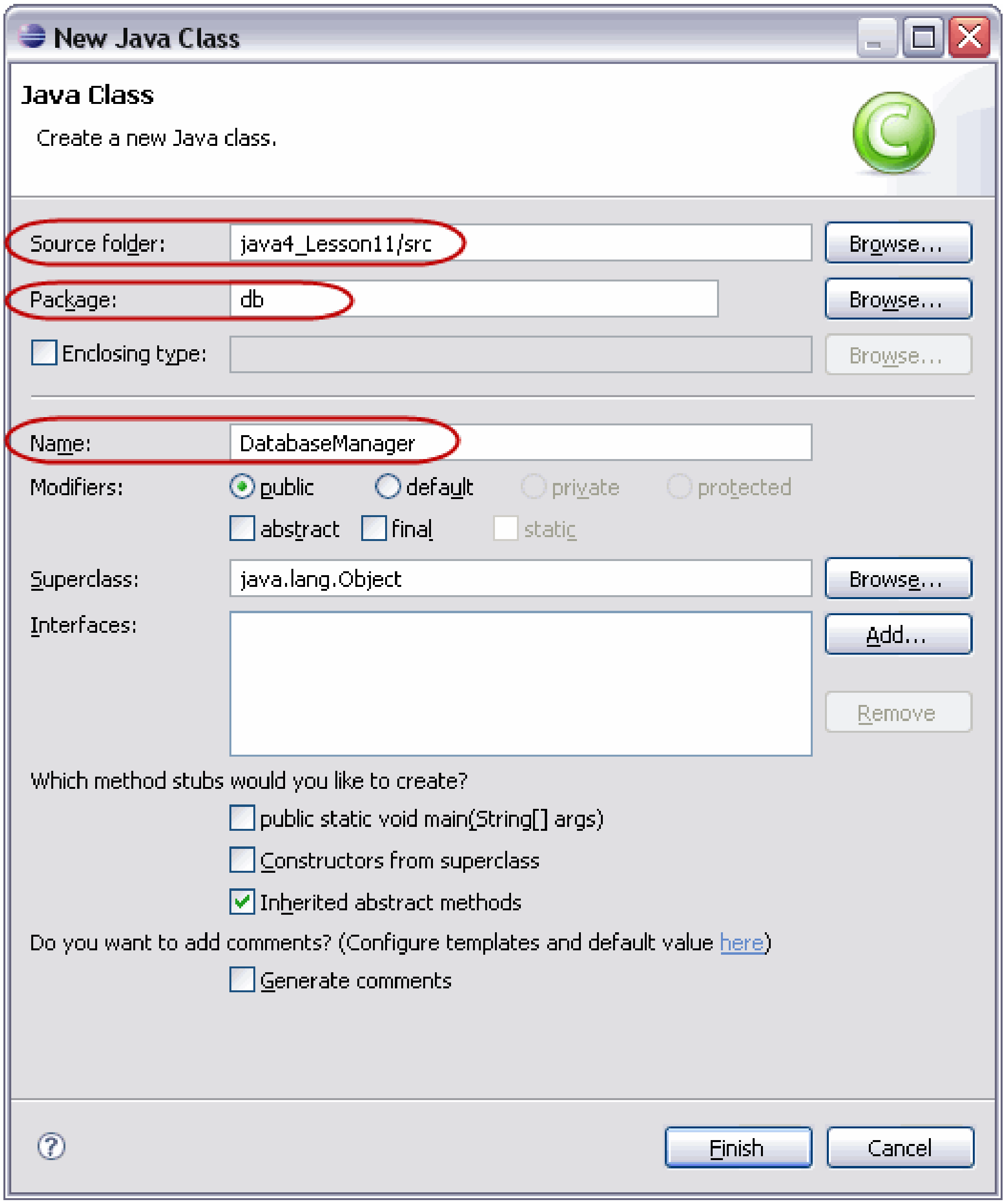
C:\jdbc\mysql-connector-java-5.1.5-bin.jar

It provides Java with a location for storing classes that it needs for this application. You will need to access this path for each project that uses the database, so remember this procedure.

It appears that Eclipse places these "external archives" into a folder called **Referenced Libraries**, but because this is the *Referenced* Library, Eclipse actually just points to the location of the real file. A CLASSPATH is a PATH to the CLASSES that Java requires for an application.

Verifying the Connection

Now let's experiment with connectivity. Create a new **java4_Lesson11** project. If you're given the option to **Open Associated Perspective**, click **No**. Create a **DatabaseManager** class in the project as shown:



Java Class

Create a new Java class.

Source folder: java4_Lesson11/src [Browse...]

Package: db [Browse...]

□ Enclosing type: [Browse...]

Name: DatabaseManager

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object [Browse...]

Interfaces: [Add...] [Remove]

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **DatabaseManager** as shown in **blue**:

CODE TO TYPE: DatabaseManager

```
package db;

import java.sql.*;

public class DatabaseManager {
    private Connection connection;                                             /
/ The database connection object.
    private Statement statement;                                               /
/ the database statement object, used to execute SQL commands.

    public DatabaseManager (String username, String password ) {               /
/ the constructor for the database manager.
        String url = "jdbc:mysql://sql.useractive.com:3306/" + username;       /
/ our database--username is your O'Reilly login username and is passed in.
        try {
            Class.forName ("com.mysql.jdbc.Driver");                           /
/ get the driver for this database.
            System.out.println("Driver is set; ready to go!");
        }
        catch (Exception e) {
            System.out.println("Failed to load JDBC/ODBC driver.");
            return;                                                            /
/ cannot even find the driver--return to caller since cannot do anything.
        }

        try {
  // Establish the database connection, create a statement for execution of SQL
commands.
            connection = DriverManager.getConnection (url, username, password );
  // username and password are passed into this Constructor.
            statement  = connection.createStatement();
  // statement used to do things in the database (e.g., create the PhoneBook tab
le).
        }
        catch (SQLException exception ) {
            System.out.println ("\n*** SQLException caught ***\n");
            while (exception != null)
            {
  // grab the exception caught to tell us the problem.
                System.out.println ("SQLState:   " + exception.getSQLState()   );
                System.out.println ("Message:    " + exception.getMessage()    );
                System.out.println ("Error code: " + exception.getErrorCode() );
                exception = exception.getNextException ();
                System.out.println ( "" );
            }
        }
        catch (java.lang.Exception exception) {
 // perhaps there is an exception that was not SQL related.
            exception.printStackTrace();
 // shows a trace of the exception error--like we see in the console.
        }
    }
}
```

Save it.

Let's go over this particular line of code first: **String url = "jdbc:mysql://sql.useractive.com:3306/" + username;** provides JDBC a way to identify a database. Its structure is such that the appropriate driver recognizes and establishes a connection with it. The driver writer determines the JDBC URL that identifies their particular driver. You need not worry about how to form a JDBC URL; just use the URL supplied with the drivers. For instance, if your username is *blob*, then your url to access the MySQL server at the O'Reilly School of Technology is **jdbc:mysql://sql.useractive.com:3306/blob**.

(The rest of the code will be explained in the next few sections of the lesson.)

The Factory Design Pattern

There are certain design patterns in Java that are used frequently. By becoming familiar with design patterns, programmers can avoid common pitfalls that may arise when using particular designs. We have already seen two such frequently used patterns: *Model/View/Controller (MVC)* and *Producer/Consumer* (in threads). The code in our example above incorporates another common design pattern called the *Factory* Design Pattern.

The factory method within the factory pattern *produces* an object. Factory methods are **static** methods that return an instance of the interface or class, usually through the use of a Constructor and the **new** command. Factory methods enable programs to produce objects without specifying the precise class that will access the object because they are often instantiated as an interface rather than a class.

Factory methods:

- unlike constructors, may have meaningful names, which can clarify code.
- do not need to create a new object on each invocation--objects can be cached and reused, if necessary.
- can return a subtype of their return type. That is, they can return an object whose implementation class is unknown to the caller. This is a very valuable and widely used feature in many frameworks that use interfaces as the return type of static factory methods.

Common names for factory methods include **getInstance()** and **valueOf()**, though using these names is optional--choose whatever makes sense for your particular usage.

In our example, when connecting to the database, we don't use **new**, but obtain objects using the statements **Class.forName("com.mysql.jdbc.Driver")**, **connection = DriverManager.getConnection(url, username, password)**, and **statement = connection.createStatement()**.

API Go to the **java.lang.Class** class and look at the **static** method **forName(String className)**. This method automatically creates an instance of a driver and registers it with the DriverManager, so you don't have to create an instance of the class.

API Go to the **java.sql.DriverManager** class and look at the **static** method **getConnection(String url, String user, String password)**. The method returns an **instance** of the **interface** **Connection**--because it is an interface, it does not have a Constructor and cannot be created with the **new** command. By returning an **instance** of the interface, the methods are implemented and you can use them.

In the **java.sql.DriverManager** class, the **Class.forName()** is no longer needed. Later we'll comment it out and run the code to see if it works as specified.

API Go to the **java.sql.Connection** interface and then to the **createStatement()** method. It returns an instance of the **interface** **Statement**. Again, because **Statement** is an interface, it does not have a Constructor and cannot be created with the **new** command.

Finally, when a method is **static**, it can be called from the class. Many of the methods we'll use for databases and networking will be **static** and many of the classes will use the factory method pattern.

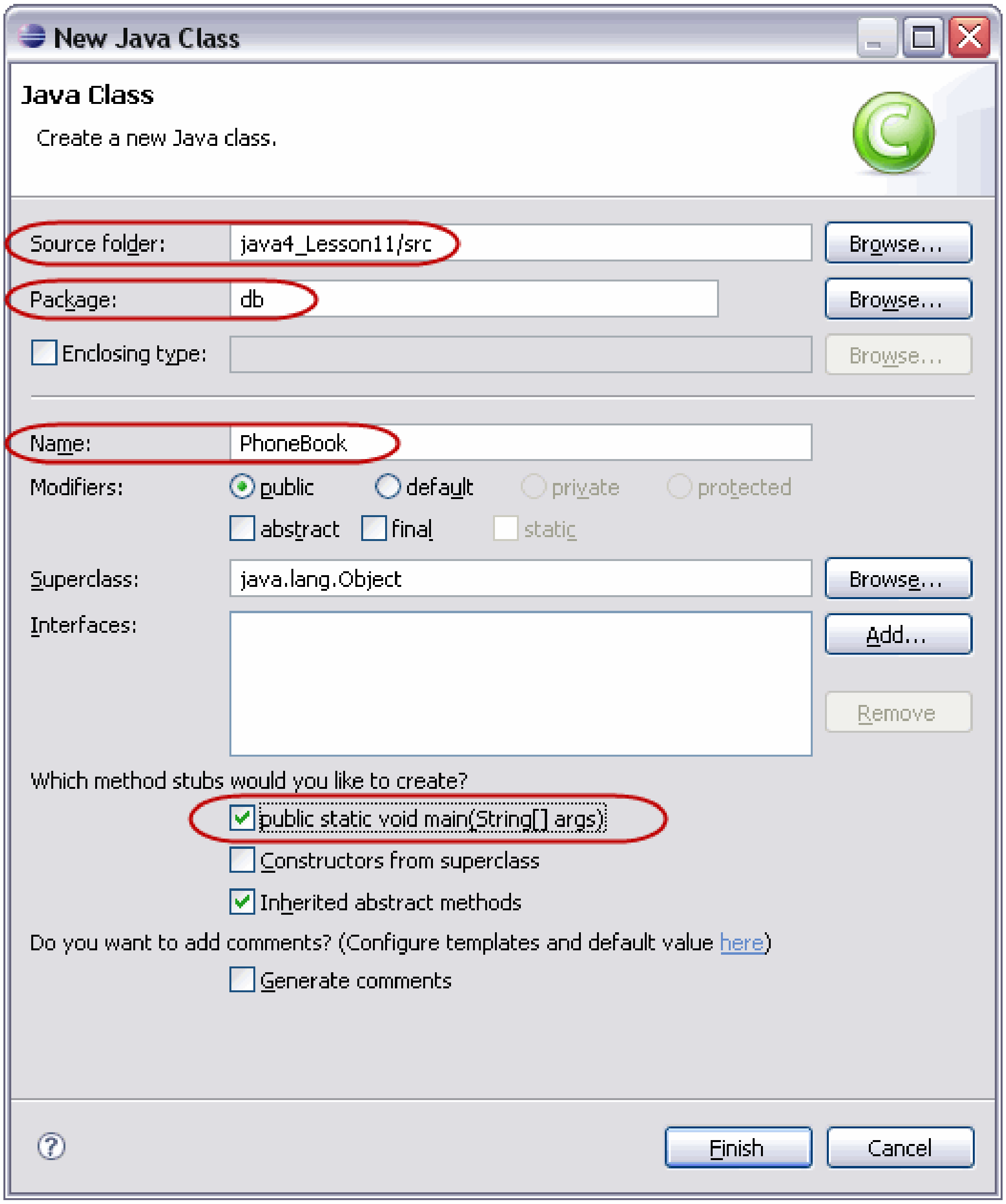
Now, let's get back to our database implementation.

Testing Our Connection

It's wise to check the **DatabaseManager** code one step at a time, so let's make a class to instantiate it and check our connection.

The Main

In the java4_Lesson11 project, create a new **PhoneBook** class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson11/src

Package: db

□ Enclosing type:

Name: PhoneBook

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☑ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **PhoneBook** as shown in **blue**:

CODE TO TYPE: PhoneBook

```
package db;

public class PhoneBook {

    public static void main(String[] args) {  // args[0] must be the username and args[1] must be the password to connect to the mysql database

        DatabaseManager databaseManager = new DatabaseManager( args[0], args[1]);  // Create the database manager.
    }
}
```

Save it. We have two ways to run it.

1: Providing Parameters In Code

The first way is to put your username and password right into your code. This is usually a bad practice, especially for a shared application, because whenever someone else wants to use the code, they need to edit and recompile. Still, for the sake of testing, sometimes it's convenient.

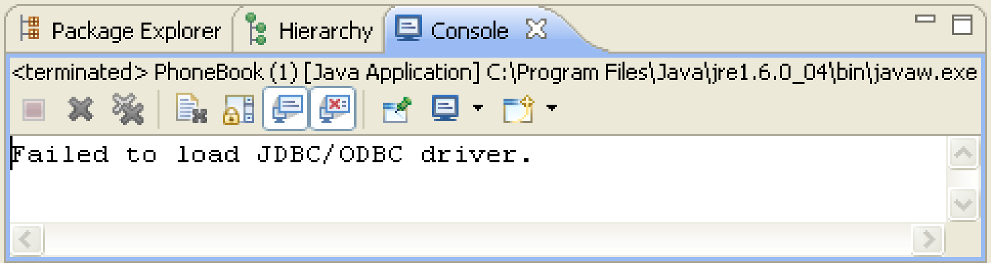
Edit **PhoneBook** as shown in **blue** below to reflect your username and password:

CODE TO EDIT: PhoneBook

```
package db;

public class PhoneBook {

    public static void main ( String[] args ) {
        // suppose your username is Iam and password is soCool
        DatabaseManager databaseManager = new DatabaseManager( "Iam", "soCool" )
;
    }
}
```

Note that "Iam" and "soCool" are *not* a valid username and password; replace them with your own.

Save and run it. You'll see this in the console:



```
<terminated> PhoneBook (1) [Java Application] C:\Program Files\Java\jre1.6.0_04\bin\javaw.exe
Failed to load JDBC/ODBC driver.
```

We'd better change the code back to the way it was before:

CODE TO EDIT: PhoneBook

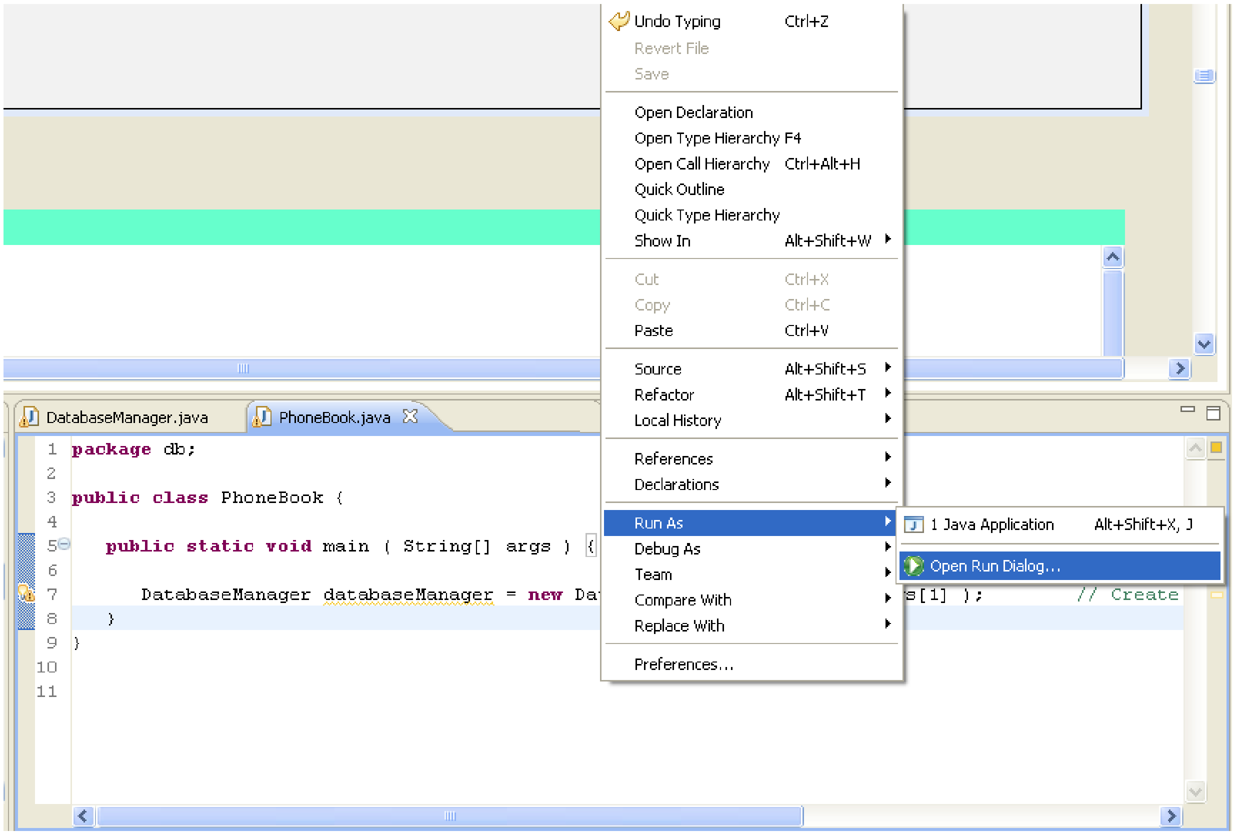
```
package db;

public class PhoneBook {

    public static void main ( String[] args ) {    // args[0] must be the usernam
e and args[1] must be the password to connect to the mysql database
        DatabaseManager databaseManager = new DatabaseManager( args[0], args[1]
);    // Create the database manager.
    }
}
```

Save it. We'll try to run it another way:

2: Giving Parameters in Eclipse

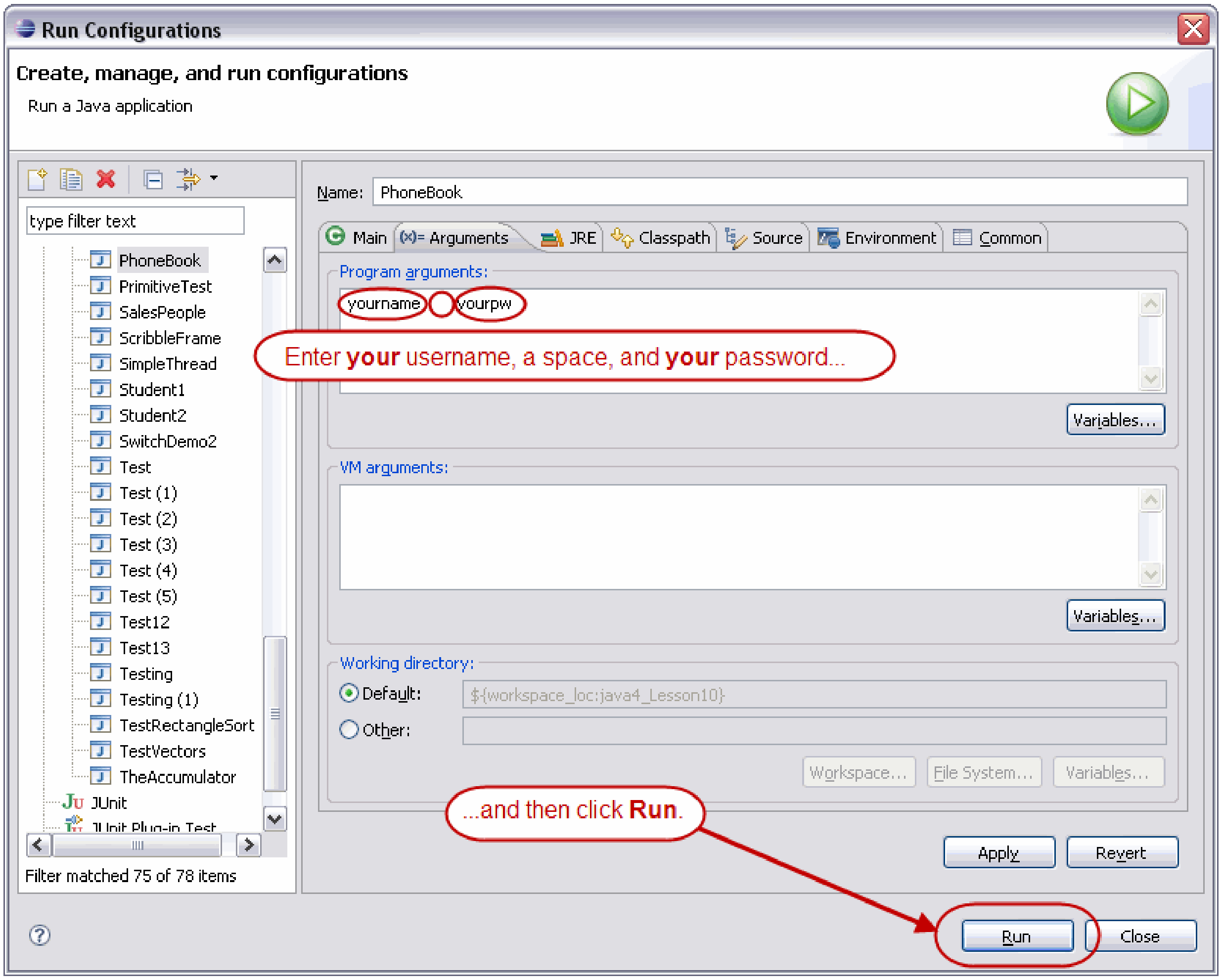
To run this program from the *command line* (outside of Eclipse), you would enter **java PhoneBook yourUserName yourPassword** (with your real username and password, of course). We use the Eclipse GUI to do this, and Eclipse enters our commands for us.

Right-click in the editor window for PhoneBook.java. Choose **Run As | Run Configurations...**:



```
package db;

public class PhoneBook {

    public static void main ( String[] args ) {

        DatabaseManager databaseManager = new Da                     s[1] );      // Create
    }
}
```

In the Run window that opens, choose the **Arguments** tab. Provide your username and password in the **Program arguments:** box. Again, use the username and password you use to access the course. Click **Run**:



Name: PhoneBook

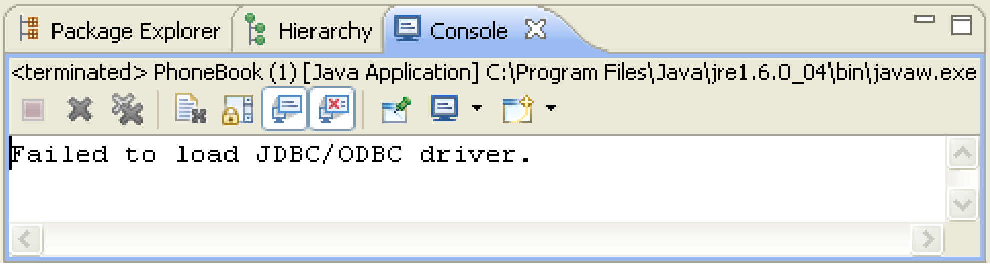
Program arguments: yourname yourpw

Working directory:
● Default: ${workspace_loc:java4_Lesson10}
○ Other:

Enter **your** username, a space, and **your** password...

...and then click **Run**.

You *still* see this:



```
<terminated> PhoneBook (1) [Java Application] C:\Program Files\Java\jre1.6.0_04\bin\javaw.exe
Failed to load JDBC/ODBC driver.
```

This problem is all too common. The Console says "**Failed to load JDBC/ODBC driver.**" That text is from the **DatabaseManager** class, in the **catch** block around line 16.

OBSERVE: DatabaseManager Constructor

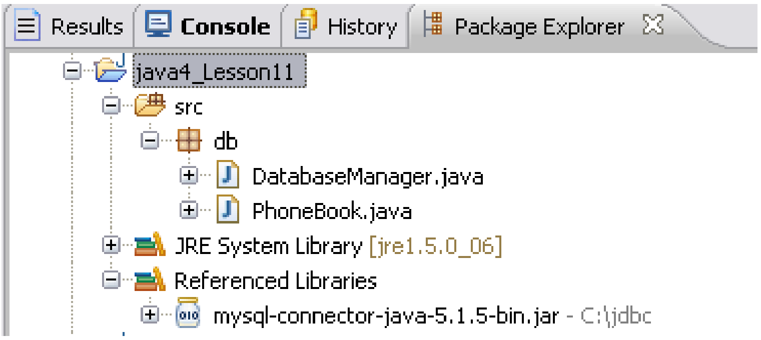
```
public DatabaseManager (String username, String password ) {              // the
constructor for the database manager
    String url = "jdbc:mysql://sql.useractive.com:3306/" + username;    // where
 username is your O'Reilly login username
    try {
   Class.forName ("com.mysql.jdbc.Driver");
    }
    catch (Exception e) {
        System.out.println("Failed to load JDBC/ODBC driver.");
        return;
    }
}
```

We thought that **Class.forName ("com.mysql.jdbc.Driver")** would go and get the driver, but apparently it didn't find that driver, because it threw an **Exception** that was caught by our code.

We put the driver's .jar into the **databaseDriver** Project, and we put these Java classes into the **java4_Lesson11** Project. Now we need to get the driver into the Project we're working on:

Right-click on the java4_Lesson11 Project, then choose **Build Path | Add External Archives**, which opens the file browser for you to get the driver. In the file dialog, start to type the path **C:\jdbc\mysql-connector-java-5.1.5-bin.jar**. The auto-complete should allow you to press Tab to fill in the file name **mysql-connector-java-5.1.5-bin.jar**.

Now your java4_Lesson11 should look like this:



Run PhoneBook again, giving it the arguments as shown above (Eclipse may remember them for us).

It's a good thing we included that **System.out.println**. If our program runs the way we want it to, **"Driver is set; ready to go!"** will appear in the Console.

Sending SQL Statements

Now that we know we have a connection, we can play with the database. We'll populate our database with some common SQL statements, specifically, methods that do the following:

- Create a table in the database named PhoneBook: **statement.execute("create table PhoneBook(Name varchar (32), PhoneNumber varchar (18))");**
- Add names and phone numbers to the *PhoneBook*: **statement.execute ("insert into PhoneBook values ('" + name + "', '" + phoneNumber + "');");**

Most of the method calls are to the instance of the interface **Statement** named **statement**. Even though only a few method calls are *to* the database, constructing them still requires *a lot* of code. Most of that code is contained within **try/catch** blocks and **System.out.printlns** and provides information about exceptions.

API Go to the interface **java.sql.Statement**. Scroll to the **Method Detail** section (or click here). Scroll down through the methods; every one of them throws an **SQLException**.

Tip When writing code with many methods that throw exceptions, localize your **try/catch** blocks. That is, place each statement (or small grouping) into its own **try** clause so you can identify the method that threw the exception.

In our example, each method we define in **DatabaseManager** contains relatively few method calls. And each method has its own try/catch clause for the set of method calls within the method block. There's no other way to do it.

Here's some additional information about exceptions.

Edit **DatabaseManager** as shown in **blue**:

CODE TO EDIT: DatabaseManager

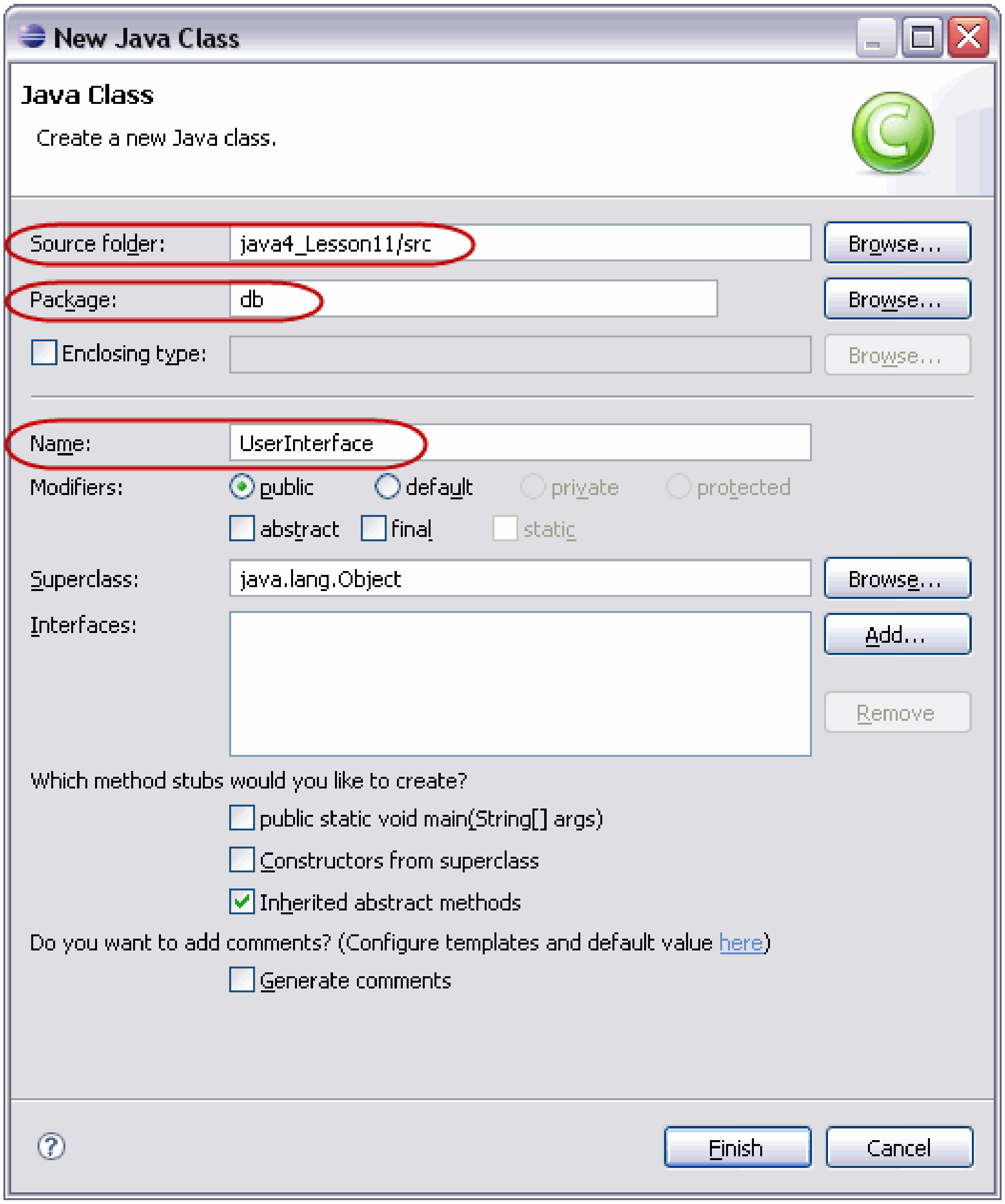
```
package db;

import java.sql.*;

public class DatabaseManager {

    private Connection connection;  // The database connection object.
    private Statement statement;    // the database statement object, used to execute S
QL commands.

    public DatabaseManager (String username, String password ) {          // the constr
uctor for the database manager
        String url = "jdbc:mysql://sql.useractive.com:3306/" + username;  // where user
name is your O'Reilly login username
        try {
            Class.forName ("com.mysql.jdbc.Driver");
        }
        catch (Exception e) {
            System.out.println("Failed to load JDBC/ODBC driver.");
            return;
        }

        try {                                                                        //
Establish the database connection, create a statement for execution of SQL commands.
            connection = DriverManager.getConnection (url, username, password );     //
 username and password are passed into this Constructor
            statement  = connection.createStatement();
            statement.execute("create table PhoneBook (Name varchar (32), PhoneNumber v
archar (18));"); // create a table in the database

        }
        catch (SQLException exception ) {
            System.out.println ("\n*** SQLException caught ***\n");
            while (exception != null)
            {                                                                          /
/ tell us the problem
                System.out.println ("SQLState:    " + exception.getSQLState()  );
                System.out.println ("Message:     " + exception.getMessage()   );
                System.out.println ("Error code:  " + exception.getErrorCode() );
                exception = exception.getNextException ();

                System.out.println ( "" );
            }
        }
        catch ( java.lang.Exception exception ) {
            exception.printStackTrace();
        }
    }

    public void addEntry (String name, String phoneNumber ){                      // a
dds an entry to the Phone Book
        try
        {
            statement.execute ( "insert into PhoneBook values ('" + name + "', '" + pho
neNumber + "');" );
        }
        catch ( SQLException exception )
        {
            System.out.println ("\n*** SQLException caught ***\n");

            while ( exception != null)
            {
                System.out.println ("SQLState:    " + exception.getSQLState()  );
                System.out.println ("Message:     " + exception.getMessage()   );
                System.out.println ("Error code:  " + exception.getErrorCode() );
                exception = exception.getNextException ();
                System.out.println ( "" );
            }
        }
        catch(java.lang.Exception exception )
        {
            exception.printStackTrace();
        }
    }
}
```

Save it.

User Access and Input

Next, we need to allow a user to give commands. Granted, we don't have many commands at our disposal at this time, but we have to start somewhere, right?

In the java4_Lesson11 project, create a **UserInterface** class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson11/src

Package: db

□ Enclosing type:

Name: UserInterface

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **UserInterface** as shown in **blue**:

CODE TO TYPE: UserInterface

```
package db;

import java.sql.*;
import java.util.*;

public class UserInterface {

    private DatabaseManager database;                                       // th
e reference to the DatabaseManager object,
                                                                           // han
dles all requests to access the database

    public UserInterface(DatabaseManager theDatabaseManager) {
     database = theDatabaseManager;
    }

    public void start() {
        Scanner in = new Scanner (System.in);
        while (true) {                                  // Continue until the u
ser quits
            System.out.println ("Click in the Console,"
             + "\n then enter a command:"
             + "\n A (then Enter) to Add a phone book entry, \n"
             + "Click red square to quit (terminate) for now.");

            String command = in.nextLine();

            if ( command.charAt(0) == 'A' )
            {
                System.out.println ("Enter name: ");
                String name = in.nextLine();
                System.out.println ("Enter phone number: ");
                String phoneNumber = in.nextLine();
                database.addEntry (name, phoneNumber);  // Add this entry to the
 database.
            }
        }
    }
}
```

Save it.

The **start()** method of the class has a loop that starts with **while (true)**. This allows the application to stay open for continuous input until the user finishes. After each user input, the loop performs the specified action and then returns to prompt again.

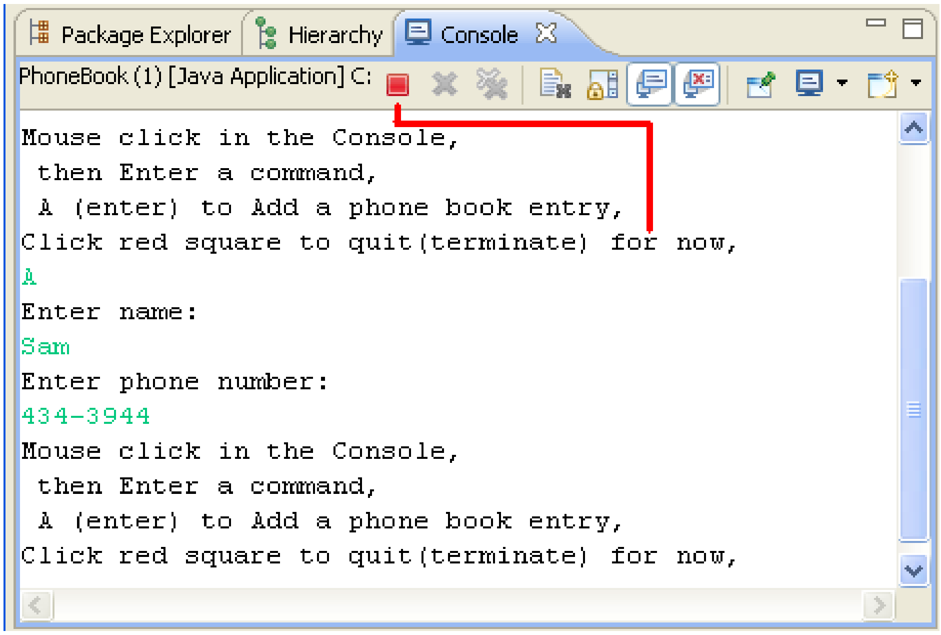
Edit the **PhoneBook** class to instantiate and start this interface as shown in **blue**:

CODE TO EDIT: PhoneBook

```
package db;

public class PhoneBook {

    public static void main ( String[] args ) {
   // args[0] must be the username and

   // args[1] must be the password to connect to the mysql database
        DatabaseManager databaseManager = new DatabaseManager(args[0], args[1] )
;  // Create the database manager.

        UserInterface userInterface = new UserInterface(databaseManager );
   // Create access for user input.
        userInterface.start();
    }
}
```

Save and run it. Make sure to click in the Console so your input goes there.

Except for the red line pointing to the Terminate button, you'll see this (the **green** text is our sample input):

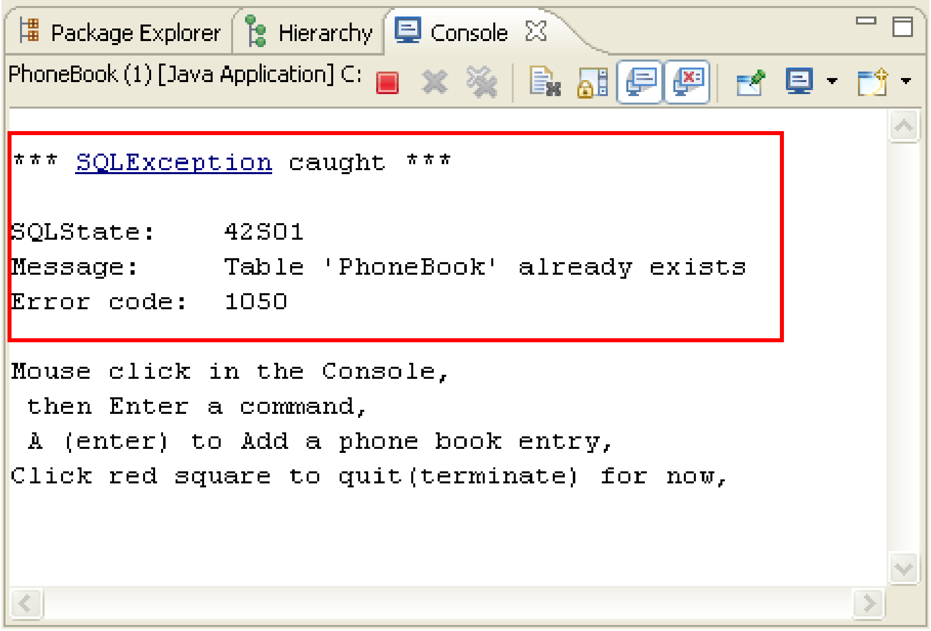


```
Mouse click in the Console,
 then Enter a command,
 A (enter) to Add a phone book entry,
Click red square to quit(terminate) for now,
A
Enter name:
Sam
Enter phone number:
434-3944
Mouse click in the Console,
 then Enter a command,
 A (enter) to Add a phone book entry,
Click red square to quit(terminate) for now,
```

Closing Our Connections

Never expect the user to use **Ctrl+C** or **Terminate** to end a program; that's poor design. The user may not know about these tools and more importantly, terminating an application this way might unexpectedly leave the database without a "clean-up" and information could be lost.

Run **PhoneBook** again to see another reason that an application must always close its open connections and processes:



```
*** SQLException caught ***

SQLState:     42S01
Message:      Table 'PhoneBook' already exists
Error code:   1050

Mouse click in the Console,
 then Enter a command,
 A (enter) to Add a phone book entry,
Click red square to quit(terminate) for now,
```

The first statement that we execute after getting our connection is:

statement.execute ("create table PhoneBook (Name varchar (32), PhoneNumber varchar (18));");

Specifically, we execute a statement to create a table named **PhoneBook** in the database. So, when we run it the second time, after terminating without proper procedure, we are trying to create a table that is already there. We have two ways around this problem: either avoid re-creating the table every time we access the database *or* remove the table when finished with the demonstration.

In the project for this lesson, you'll need to fix the problem so that we can stop our program with some dignity.

Additional Resources

Here are some additional items you can read to learn about Java and database connectivity.

- Trail: JDBC(TM) Database Access
- JavaWorld article about drivers.
- Gamelan's Using JDBC with MySQL, Getting Started

Databases and Java: Processing Information

Getting Results

In this lesson, we'll add methods to our code that will allow us to:

- Process the **ResultSet** we get from the JDBC.
- Write a **getEntries()** method to retrieve names and phone numbers and display the entries from the **PhoneBook** table.
- Write an **inspectTables()** method to determine whether a table already exists before creating it, and a **close()** method to *drop* (remove) the table and close the database connection.
- Create a better login using a **DialogBox**.

Databases and SQL

In the previous lesson, we connected to the database, created a table, and put information into it. If we use JDBC to *populate* the table or put the information into the database, we use an instance of the **Statement** interface to execute SQL statements like these:

- **CREATE TABLE** (makes whole tables)
- **INSERT** (adds a row)
- **DELETE** (removes a row)
- **UPDATE** (changes an existing value in a column or columns)

We used an instance **statement** of **Statement** to execute a **CREATE** and **INSERT**. This course uses the technique that uses JDBC, but information can be inserted into a table in any of these ways:

1. Using the database or IDE graphical interface.
2. Using ANT (and .xml files).
3. Using JDBC SQL written into an application.

Although we've only created one table in our example, we can add and manipulate any number of tables using the same techniques. For instance, we can **JOIN** related data from multiple tables. However, in this lesson, we'll continue to use our simple example and start the procedure for retrieving information.

The JDBC returns results from a **Statement**'s query in a **ResultSet** object, so we need an instance of the **ResultSet** interface to hold our results. The **ResultSet** interface provides methods for retrieving and manipulating the results of queries; particularly, it provides getter methods (such as **getBoolean()** and **getLong()**) for retrieving column values from the current row.

For specifics and examples, see the Oracle tutorial Relational Database Overview.

ResultSet

A **ResultSet** is the table of results from your query. This table can have one or more rows. You need to manipulate these results to get the information you want from the table. Although we usually look at a table as a two-dimensional array, the JDBC provides the **ResultSet** to manipulate this array to extract whatever information you need. We'll go over some examples in this lesson and the next, but the Oracle tutorial is always a good source for more information: Retrieving Values from Result Sets—and of course, the API. API Go to the **java.sql.ResultSet** interface and read the introduction.

Getting Information About Information

It's tough to anticipate the best ways to manipulate data when we're not even sure which data is present. We know how to make **for** loops to go through two-dimensional arrays, but in the database table, we don't know how many rows exist! We can't write code that goes to some **arrays.length** because we don't even know that we have an array--we only have information returned from a table. We need to know how many rows are in that information. In other words, we need information about the information we are getting! The JDBC helps us by providing *MetaData* classes.

Metadata

- *metaknowledge* is knowledge about knowledge.
- *metalanguage* is a language about languages.
- *metatheory* is the theory about theories.
- *metadata* is data about data.

API Go to the **java.sql** package. Scroll through the **Interface Summary**. We see some interesting names:

- DatabaseMetaData
- ParameterMetaData
- ResultSetMetaData

These interfaces can be instantiated by instances of the objects that help identify the data. Let's take a look at a method that illustrates this idea.

In the java4_Lesson11 Project, edit **DatabaseManager** as shown in **blue**:

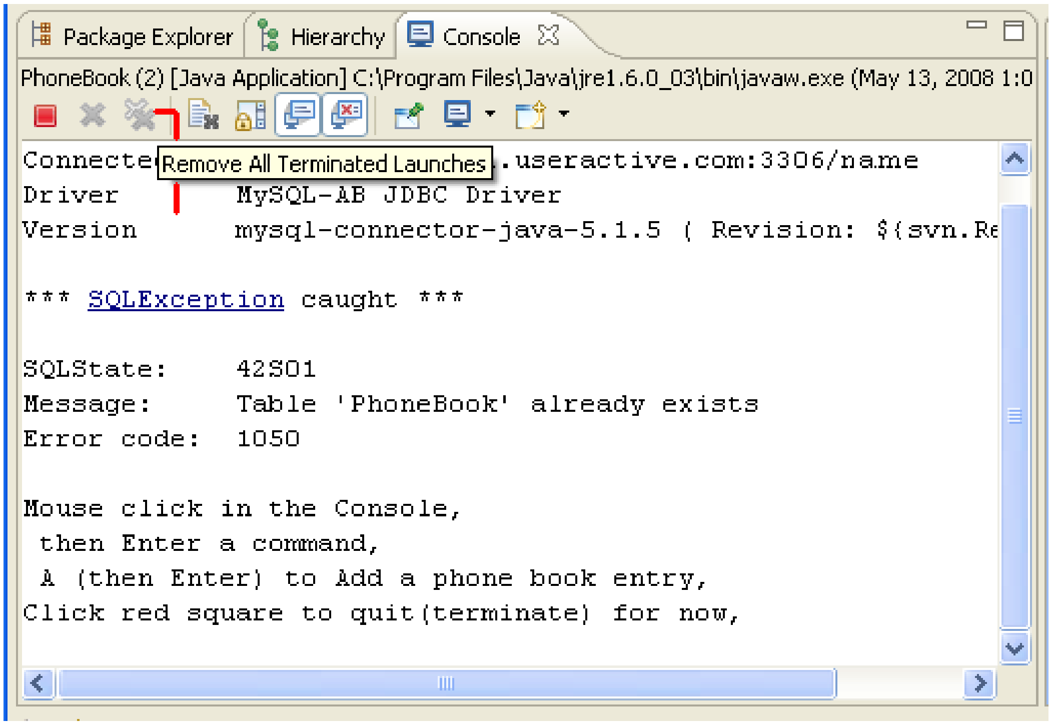
CODE TO EDIT: DatabaseManager

```
package db;

import java.sql.*;

public class DatabaseManager {

    private Connection connection;                                        // The
 database connection object.
    private Statement statement;                                          // the
 database statement object, used to execute SQL commands.

    public DatabaseManager (String username, String password ) {          // the
 constructor for the database manager
        String url = "jdbc:mysql://sql.useractive.com:3306/" + username;
        try {
            Class.forName ("com.mysql.jdbc.Driver");
        }
        catch (Exception e) {
            System.out.println("Failed to load JDBC/ODBC driver.");
            return;
        }
        try {                                                             // Est
ablish the database connection, create a statement for execution of SQL commands
.
            connection = DriverManager.getConnection (url, username, password );
   // username and password are passed into this Constructor
            // Get the DatabaseMetaData object and display
            // some information about the connection

            DatabaseMetaData aboutDB = connection.getMetaData();

            System.out.println("\nConnected to " + aboutDB.getURL());
            System.out.println("Driver       " + aboutDB.getDriverName());
            System.out.println("Version      " + aboutDB.getDriverVersion());

            statement  = connection.createStatement();
            statement.execute ("create table PhoneBook (Name varchar (32), Phone
Number varchar (18) );");    // create a table in the database

        }
        catch (SQLException exception ) {
            System.out.println ("\n*** SQLException caught ***\n");
            while (exception != null)
            {
      // tell us the problem
                System.out.println ("SQLState:    " + exception.getSQLState()  )
;
                System.out.println ("Message:     " + exception.getMessage()   )
;
                System.out.println ("Error code:  " + exception.getErrorCode() )
;
                exception = exception.getNextException ();

                System.out.println ( "" );
            }
        }
        catch ( java.lang.Exception exception ) {
            exception.printStackTrace();
        }
    }

    public void addEntry (String name, String phoneNumber ){
   // adds an entry to the Phone Book
       try
       {
           statement.execute ( "insert into PhoneBook values ('" + name + "', '"
 + phoneNumber + "');" );
       }
       catch ( SQLException exception )
       {
           System.out.println ("\n*** SQLException caught ***\n");

           while ( exception != null)
           {
               System.out.println ("SQLState:     " + exception.getSQLState()  );
               System.out.println ("Message:      " + exception.getMessage()   );
               System.out.println ("Error code:   " + exception.getErrorCode() );

               exception = exception.getNextException ();
               System.out.println ( "" );
           }
       }
       catch(java.lang.Exception exception )
       {
           exception.printStackTrace();
       }
    }
}
```

Save and run it (from **PhoneBook**). You'll see something like this:

```
Connected to jdbc:mysql://sql.useractive.com:3306/name
Driver       MySQL-AB JDBC Driver
Version      mysql-connector-java-5.1.5 ( Revision: ${svn.Revision} )
```

Even though you may exit the application by clicking the red **Terminate** square, you should also either "Remove Launch" (single X) or "Remove All Terminated Launches" by clicking the double X:


Package Explorer
Hierarchy
Console
PhoneBook (2) [Java Application] C:\Program Files\Java\jre1.6.0_03\bin\javaw.exe (May 13, 2008 1:0
Remove All Terminated Launches

```
Connecte                .useractive.com:3306/name
Driver       MySQL-AB JDBC Driver
Version      mysql-connector-java-5.1.5 ( Revision: ${svn.Re

*** SQLException caught ***

SQLState:    42S01
Message:     Table 'PhoneBook' already exists
Error code:  1050

Mouse click in the Console,
 then Enter a command,
 A (then Enter) to Add a phone book entry,
Click red square to quit(terminate) for now,
```

In databases, when a query returns a **ResultSet**, it is returning a table of data. **ResultSet**s are used to retrieve data so that it can be manipulated. Suppose you want to loop through the data to find something, but you don't know how many rows or columns of data exist, so you don't know how many times to loop. This would be a perfect time to use **ResultSetMetaData**.

Creating a Table

If one doesn't exist already, we'll create a new **PhoneBook** table. By doing this, we won't need to **drop** the database table; we can keep the information we have just submitted *in* it for the next time we access it. We'll use both the **ResultSet** and the **ResultSetMetaData** interfaces.

In the java4_Lesson11 Project, edit **DatabaseManager** as shown in **blue** below:

CODE TO EDIT: DatabaseManager

```
package db;

import java.sql.*;

public class DatabaseManager {

    private Connection connection;  // The database connection object.
    private Statement statement;    // the database statement object, used to ex
ecute SQL commands.

    public DatabaseManager (String username, String password ) {          // the
 constructor for the database manager
        String url = "jdbc:mysql://sql.useractive.com:3306/" + username;
        try {
            Class.forName ("com.mysql.jdbc.Driver");
        }
        catch (Exception e) {
            System.out.println("Failed to load JDBC/ODBC driver.");
            return;
        }

        try {
  // Establish the database connection, create a statement for execution of SQL
commands.
            connection = DriverManager.getConnection (url, username, password );
  // username and password are passed into this Constructor

            statement  = connection.createStatement();
            DatabaseMetaData aboutDB = connection.getMetaData ();
            // do more useful things with the meta class
            String [] tableType = {"TABLE"};
            ResultSet rs = aboutDB.getTables(null, null, "PhoneBook",  tableType
);  // for more info about this method, see the getTables method in DatabaseMeta
Data in the API

            if (!inspectForTable(rs))
   // use this method (written below) to see if we already have the table PhoneB
ook
             statement.execute ("create table PhoneBook (Name varchar (32), Phon
eNumber varchar (18) );");  // if we do NOT already have one, we want to do this

            rs.close();
    // in this example, the ResultSet is local, so close it here
        }
        catch (SQLException exception ) {
            System.out.println ("\n*** SQLException caught ***\n");
            while (exception != null)
            {
       // tell us the problem
                System.out.println ("SQLState:    " + exception.getSQLState()  )
;
                System.out.println ("Message:     " + exception.getMessage()   )
;
                System.out.println ("Error code:  " + exception.getErrorCode() )
;
                exception = exception.getNextException ();

                System.out.println ( "" );
            }
        }
        catch ( java.lang.Exception exception ) {
            exception.printStackTrace();
        }
    }

    public void addEntry (String name, String phoneNumber ){
```

```
   // adds an entry to the Phone Book
        try
        {
            statement.execute ( "insert into PhoneBook values ('" + name + "', '
" + phoneNumber + "');" );
        }
        catch ( SQLException exception )
        {
            System.out.println ("\n*** SQLException caught ***\n");

            while ( exception != null)
            {
                System.out.println ("SQLState:    " + exception.getSQLState()  )
;
                System.out.println ("Message:     " + exception.getMessage()   )
;
                System.out.println ("Error code:  " + exception.getErrorCode() )
;

                exception = exception.getNextException ();
                System.out.println ( "" );
            }
        }
        catch(java.lang.Exception exception )
        {
            exception.printStackTrace();
        }
    }

    private static boolean inspectForTable (ResultSet rs)  throws SQLException {
  // will be caught when used
        int i;
        ResultSetMetaData rsmd = rs.getMetaData ();
  // Get the ResultSetMetaData.  This will be used for information about the col
umns.
        int numCols = rsmd.getColumnCount ();
  // Get the number of columns in the result set

        for (i=1; i<=numCols; i++) {                                           /
/ Display column headings
            if (i > 1) System.out.print(", ");
  // just to show what is there for our curiosity
            System.out.print(rsmd.getColumnLabel(i));
        }
        System.out.println("");

        boolean more = rs.next ();
        while (more) {                     // Display data, fetching until end o
f the result set
                                           // Loop through each row, getting the
 column data and displaying
            for (i=1; i<=numCols; i++)
            {
                System.out.print(rs.getString(i)+"\n");
                if (rsmd.getColumnLabel(i) == "TABLE_NAME")
                    if (rs.getString(i).equals("PhoneBook"))
                    {
                        System.out.println("Found one that equals " + rs.getStri
ng(i));   // is PhoneBook there already or not?
                        return true;
          // it is, tell the method that inquired
                    }
            }
            System.out.println("");
            more = rs.next ();
      // Fetch the next result set row
        }
        return false;
        // went though all of the rows and it was not there
    }
}
```

Save and run it (from **PhoneBook**). Terminate it and run it again. The SQL Exception alerting you to an existing table should no longer appear. We've added a few extra **printlns** in our code so we can observe what's been returned. We can remove them later.

Closing Properly

In the objective for the last lesson, you added a method to **close()**. We'll include it here too, but we'll also give the user the option to either keep the information or drop the table.

Edit **DatabaseManager** as shown in **blue**:

CODE TO EDIT: DatabaseManager

```
package db;

import java.sql.*;

public class DatabaseManager {

    private Connection connection;                                        // The databa
se connection object.
    private Statement statement;                                          // the databa
se statement object, used to execute SQL commands.

    public DatabaseManager (String username, String password ) {          // the constr
uctor for the database manager
        String url = "jdbc:mysql://sql.useractive.com:3306/" + username;
        try {
            Class.forName ("com.mysql.jdbc.Driver");
        }
        catch (Exception e) {
            System.out.println("Failed to load JDBC/ODBC driver.");
            return;
        }

        try {                                                             // Establish
the database connection, create a statement for execution of SQL commands.
            connection = DriverManager.getConnection (url, username, password );  // us
ername and password are passed into this Constructor
            statement  = connection.createStatement();

            DatabaseMetaData aboutDB = connection.getMetaData ();
            // do more useful things with the meta class
            String [] tableType = {"TABLE"};
            ResultSet rs = aboutDB.getTables(null, null, "PhoneBook",  tableType);  //
for more info about this method, see the getTables method in DatabaseMetaData in the AP
I

            if (!inspectForTable (rs))                                    // use this m
ethod to see if we already have the table PhoneBook
             statement.execute ("create table PhoneBook (Name varchar (32), PhoneNumber
 varchar (18) );");     // if we do NOT already have one, we want to do this
            rs.close();                                                          //
in this example, the ResultSet is local - so close it here

        }
        catch (SQLException exception ) {
            System.out.println ("\n*** SQLException caught ***\n");
            while (exception != null)
            {                                                                     /
/ tell us the problem
                System.out.println ("SQLState:    " + exception.getSQLState()  );
                System.out.println ("Message:     " + exception.getMessage()   );
                System.out.println ("Error code:  " + exception.getErrorCode() );
                exception = exception.getNextException ();
                System.out.println ( "" );
            }
        }
        catch ( java.lang.Exception exception ) {
            exception.printStackTrace();
        }
    }

    public void addEntry (String name, String phoneNumber ){                       // a
dds an entry to the Phone Book
        try
        {
            statement.execute ( "insert into PhoneBook values ('" + name + "', '" + pho
neNumber + "');" );
```

```
        }
        catch ( SQLException exception )
        {
            System.out.println ("\n*** SQLException caught ***\n");
            while ( exception != null)
            {
                System.out.println ("SQLState:    " + exception.getSQLState()  );
                System.out.println ("Message:     " + exception.getMessage()   );
                System.out.println ("Error code:  " + exception.getErrorCode() );
                exception = exception.getNextException ();
                System.out.println ( "" );
            }
        }
        catch(java.lang.Exception exception )
        {
            exception.printStackTrace();
        }
    }

    private static boolean inspectForTable (ResultSet rs)  throws SQLException {  // wi
ll be caught when used
        int i;
        ResultSetMetaData rsmd = rs.getMetaData ();                                // Ge
t the ResultSetMetaData. This will be used for information about the columns.
        int numCols = rsmd.getColumnCount ();                                      // Ge
t the number of columns in the result set
        for (i=1; i<=numCols; i++) {                                            // Displ
ay column headings
            if (i > 1) System.out.print(", ");                                     // ju
st to show what is there for our curiosity
                System.out.print(rsmd.getColumnLabel(i));
        }
        System.out.println("");

        boolean more = rs.next ();
        while (more) {                                                              // D
isplay data, fetching until end of the result set
            // Loop through each row, getting the column data and displaying
            for (i=1; i<=numCols; i++)
            {
                System.out.print(rs.getString(i)+"\n");
                if (rsmd.getColumnLabel(i) == "TABLE_NAME")
                    if (rs.getString(i).equals("PhoneBook"))
                    {
                        System.out.println("Found one that equals " + rs.getString(i));
   // is PhoneBook there already or not?
                        return true;
   // it is, tell the method that inquired
                    }
            }
            System.out.println("");
            more = rs.next ();                                                         /
/ Fetch the next result set row
        }
        return false;
// went though all of the rows and it was not there
    }

    public void close(boolean remove) {
       // drops the table and properly closes the database
        try
        {
            if (remove)
            {
                statement.execute("drop table PhoneBook;");
            }
            statement.close();
            connection.close();
        }
        catch (SQLException exception)
        {
            System.out.println("\n*** SQLException caught ***\n");
            while (exception != null)
            {
                System.out.println("SQLState:  " + exception.getSQLState());
                System.out.println("Message:   " + exception.getMessage());
                System.out.println("ErrorCode: " + exception.getErrorCode());
                exception = exception.getNextException ();
                System.out.println("");
            }
        }
        catch(java.lang.Exception exception )
        {
            exception.printStackTrace();
        }
    }
}
```

Save it.

Now we need to accommodate the option in the user interface. In the java4_Lesson11 project, edit **UserInterface** as shown in **blue**:

CODE TO EDIT: UserInterface

```
package db;

import java.sql.*;
import java.util.*;

public class UserInterface {

    private DatabaseManager database;  // the reference to the DatabaseManager object,
                                       // handles all requests to access the database

    public UserInterface(DatabaseManager theDatabaseManager) {
     database = theDatabaseManager;
    }

    public void start() {
        Scanner in = new Scanner (System.in);
        while (true) {                             // Continue until the user enters a
quit command
            System.out.println ("Click in the Console,"
             + "\n then Enter a command: (choose)"
             + "\n A (then Enter) to Add a phone book entry,"
             + "\n K (then Enter) to Exit and Keep the entries,"
             + "\n or Q (then Enter) to Quit and Remove the entries: " );

            String command = in.nextLine();

            if (command.charAt(0) == 'A')
            {
                System.out.println ("Enter name: ");
                String name = in.nextLine();
                System.out.println ("Enter phone number: ");
                String phoneNumber = in.nextLine();
                database.addEntry (name, phoneNumber);  // Add this entry to the databa
se.
            }
            else if (command.charAt(0) == 'K')
            {
                System.out.println("Bye");
                database.close(false);      // The user entered the quit command, but d
oes not want to delete info.
                return;
            }
            else if (command.charAt(0) != 'Q')
            {
                System.out.println ("Invalid command. Please enter either A, K, or Q.")
;
            }
            else                                                                 // com
mand is Q
            {
                System.out.println("Bye");
                database.close(true);                                             // Th
e user entered the quit command, so shutdown the database and return.
                return;
            }
        }
    }
}
```

Save and run it (from **PhoneBook**). Exit with **K** to **K**eep the table. Run it again.

Seeing Table Contents

Is there even anything in there? Let's find out.

Edit **DatabaseManager** as shown in **blue**:

CODE TO EDIT: DatabaseManager

```
package db;

import java.sql.*;

public class DatabaseManager {

    private Connection connection;                                         // The databa
se connection object.
    private Statement statement;                                           // the databa
se statement object, used to execute SQL commands.
    private ResultSet resultSet;                                           // results fr
om a database query

    public DatabaseManager (String username, String password ) {           // the constr
uctor for the database manager
        String url = "jdbc:mysql://sql.useractive.com:3306/" + username;
        try {
            Class.forName ("com.mysql.jdbc.Driver");
        }
        catch (Exception e) {
            System.out.println("Failed to load JDBC/ODBC driver.");
            return;
        }

        try {                                                                           //
Establish the database connection, create a statement for execution of SQL commands.
            connection = DriverManager.getConnection (url, username, password );     //
 username and password are passed into this Constructor
            statement  = connection.createStatement();

            DatabaseMetaData aboutDB = connection.getMetaData ();
                                                                                   // do
 more useful things with the meta class
            String [] tableType = {"TABLE"};
            ResultSet rs = aboutDB.getTables(null, null, "PhoneBook",  tableType);   //
 check out the getTables method in DatabaseMetaData to see more about this method

            if (!inspectForTable (rs))                                                  //
use this method to see if we already have the table PhoneBook
                statement.execute ("create table PhoneBook (Name varchar (32), PhoneNum
ber varchar (18) );");     // if we do NOT already have one, we want to do this
            rs.close();                                                                 //
in this example, the ResultSet is local - so close it here

        }
        catch (SQLException exception ) {
            System.out.println ("\n*** SQLException caught ***\n");
            while (exception != null)
            {                                                                             /
/ tell us the problem
                System.out.println ("SQLState:    " + exception.getSQLState()  );
                System.out.println ("Message:     " + exception.getMessage()   );
                System.out.println ("Error code:  " + exception.getErrorCode() );
                exception = exception.getNextException ();

                System.out.println ( "" );
            }
        }
        catch ( java.lang.Exception exception ) {
            exception.printStackTrace();
        }
    }

    public void addEntry (String name, String phoneNumber ){                       // a
dds an entry to the Phone Book
        try
        {
            statement.execute ( "insert into PhoneBook values ('" + name + "', '" + pho
neNumber + "');" );
        }
        catch ( SQLException exception )
        {
            System.out.println ("\n*** SQLException caught ***\n");

            while ( exception != null)
            {
                System.out.println ("SQLState:    " + exception.getSQLState()  );
                System.out.println ("Message:     " + exception.getMessage()   );
                System.out.println ("Error code:  " + exception.getErrorCode() );

                exception = exception.getNextException ();
                System.out.println ( "" );
            }
        }
        catch(java.lang.Exception exception )
        {
            exception.printStackTrace();
        }
    }

    private static boolean inspectForTable (ResultSet rs)  throws SQLException {  // wi
ll be caught when used
        int i;
        ResultSetMetaData rsmd = rs.getMetaData ();                                // G
et the ResultSetMetaData.  This will be used for information about the columns.
        int numCols = rsmd.getColumnCount ();                                      // G
et the number of columns in the result set

        // for (i=1; i<=numCols; i++) {                                               /
/ Display column headings
            //if (i > 1) System.out.print(", ");                                      /
/ just to show what is there for our curiosity
                // System.out.print(rsmd.getColumnLabel(i));
        //}
        //System.out.println("");

        boolean more = rs.next ();
        while (more) {                                                             // D
isplay data, fetching until end of the result set
            // Loop through each row, getting the column data and displaying
            for (i=1; i<=numCols; i++)
            {
                //System.out.print(rs.getString(i)+"\n");
                if (rsmd.getColumnLabel(i) == "TABLE_NAME")
                    if (rs.getString(i).equals("PhoneBook"))
                    {
                        System.out.println("Found one that equals " + rs.getString(i));
   // is PhoneBook there already or not?
                        return true;
   // it is, tell the method that inquired
                    }
            }
            System.out.println("");
            more = rs.next ();                                                        /
/ Fetch the next result set row
       }
       return false;                                                                  /
/ went though all of the rows and it was not there
    }

    public void getEntries(){                                                   // returns
a ResultSet containing all entries in the phone book
        try
        {
            resultSet = statement.executeQuery("SELECT * FROM PhoneBook"); // call the
query and get a ResultSet

            ResultSetMetaData metaData = resultSet.getMetaData();            // Get the
 ResultSetMetaData.
            int numCols = metaData.getColumnCount();                         // Get the
 number of columns in the result set
            int i;
            System.out.println("");
            for (i=1;  i <= numCols;  i++)
            {
                if (i > 1) System.out.print("\t\t\t\t");
                System.out.print ( metaData.getColumnLabel(i) );
            }
            System.out.println("");
            System.out.println("");

            boolean more = resultSet.next();                           // Display data,
 fetching until end of the result set
            while (more)                                                // Loop through
 each column, getting the column data and displaying
            {
                for (i = 1;  i <= numCols;  i++)
                {
                    System.out.print  (resultSet.getString(i) + "\t\t\t\t" );
                }
                System.out.println("");
                more = resultSet.next();                                 // Go to the n
ext result set row
            }
            resultSet.close();
            System.out.println("");

        }
        catch (SQLException exception)
        {
            System.out.println ("\n*** SQLException caught ***\n");

            while ( exception != null)
            {
                System.out.println("SQLState:   " + exception.getSQLState());
                System.out.println("Message:    " + exception.getMessage());
                System.out.println("Error code: " + exception.getErrorCode());

                exception = exception.getNextException();
                System.out.println("");
            }
        }
        catch (java.lang.Exception exception )
        {
            exception.printStackTrace();
        }
    }

    public void close(boolean remove){
      // drops the table and properly closes the database
        try
        {
            if (remove)
                statement.execute("drop table PhoneBook;");
            statement.close();
            connection.close();
        }
        catch (SQLException exception)
        {
            System.out.println("\n*** SQLException caught ***\n");

            while (exception != null)
            {
                System.out.println("SQLState:   " + exception.getSQLState());
                System.out.println("Message:    " + exception.getMessage());
                System.out.println("Error code: " + exception.getErrorCode());

                exception = exception.getNextException();
                System.out.println("");
            }
        }
        catch(java.lang.Exception exception)
        {
            exception.printStackTrace();
        }
    }
}
```

Save it.

We have lots of new additions to the DatabaseManager; let's give our users more capabilities to keep up with those additions.

Edit **UserInterface** as shown in **blue**:

CODE TO EDIT: UserInterface

```
package db;

import java.util.*;

public class UserInterface {

    private DatabaseManager database;                           // the reference to the
 DatabaseManager object,
                                                                // handles all requests
 to access the database
    public UserInterface(DatabaseManager theDatabaseManager) {
     database = theDatabaseManager;
    }

    public void start() {
        Scanner in = new Scanner (System.in);
        while (true) {                                      // Continue until the user
enters a quit command
            System.out.println ("Click in the Console,"
             + "\n then Enter a command: (choose)"
             + "\n A (then Enter) to Add a phone book entry, "
             + "\n D (then Enter) to Display all phone book entries,"
             + "\n K (then Enter) to exit and Keep the entries,"
             + "\n or Q (then Enter) to Quit and remove the entries: " );

            String command = in.nextLine();

            if ( command.charAt(0) == 'A' )
            {
                System.out.println ("Enter name: ");
                String name = in.nextLine();
                System.out.println ("Enter phone number: ");
                String phoneNumber = in.nextLine();
                database.addEntry (name, phoneNumber);  // Add this entry to the databa
se.
            }
            else if (command.charAt(0) == 'D')
            {
                database.getEntries();  // Query the database for the resultSet
            }
            else if (command.charAt(0) == 'K' )
            {
                System.out.println("Bye");
                database.close(false);               // The user entered the quit comma
nd, but does not want to delete info.
                return;
            }
            else if ( command.charAt(0) != 'Q' )
            {
                System.out.println ("Invalid command, please enter either A, D, K, or Q
.");
            }
            else                                     // command is Q
            {
                System.out.println("Bye");
                database.close(true);                // The user entered the quit comma
nd, so shutdown the database and return.
                return;
            }
      }
   }
}
```

Note Did you notice the modularity here? After adding each method, we made a minor edit to the user interface.

Save and run it (from **PhoneBook.java**). Use the **D** option to see all the entries that may be there already. Type **Q** to **Q**uit--this calls the **close()** method of **DatabaseManager**, which drops the table and closes all open connections to the database.

Run it again. Since it was closed properly, there shouldn't be any errors. Use the **D** option to see that all of the entries were removed. Add a few new entries and try the **D** command again. Exit with the **K** option. In fact, try both the **Q** and **K** options until you feel confident that they behave as expected.

SQL Commands

Look over the code in this Observe box:

OBSERVE: DatabaseManager Stripped

```
package db;

import java.sql.*;

public class DatabaseManager {

    private Connection connection;
    private Statement statement;
    private ResultSet resultSet;

    public DatabaseManager (String username, String password ) {
                        // connect
        String url = "jdbc:mysql://sql.useractive.com:3306/" + username;
        Class.forName ("com.mysql.jdbc.Driver");

        connection = DriverManager.getConnection (url, username, password );
        statement  = connection.createStatement();
        statement.execute ("create table PhoneBook (Name varchar (32), PhoneNumber varc
har (18) );");
    }

    public void addEntry (String name, String phoneNumber ){
                        // add entries
        statement.execute ( "insert into PhoneBook values ('" + name + "', '" + phoneNu
mber + "');" );
    }

    public ResultSet getEntries(){
                        // retrieve
        return statement.executeQuery ( "SELECT * FROM PhoneBook");
    }

    public void close(){
                         // close
        statement.execute ("drop table PhoneBook;");
        statement.close();
        connection.close();
    }
}
```

We've removed all of the **try/catch** clauses and **System.out.println** statements to show what's left of the **DatabaseManager** (other than the Meta questions). You might be tempted to try to program database applications without all of those try/catch clauses, but that would make your application unstable and, for all intents and purposes, unusable. In fact, Java won't even let you do it. **SQLExceptions** are not **RuntimeExceptions** and therefore must be handled.

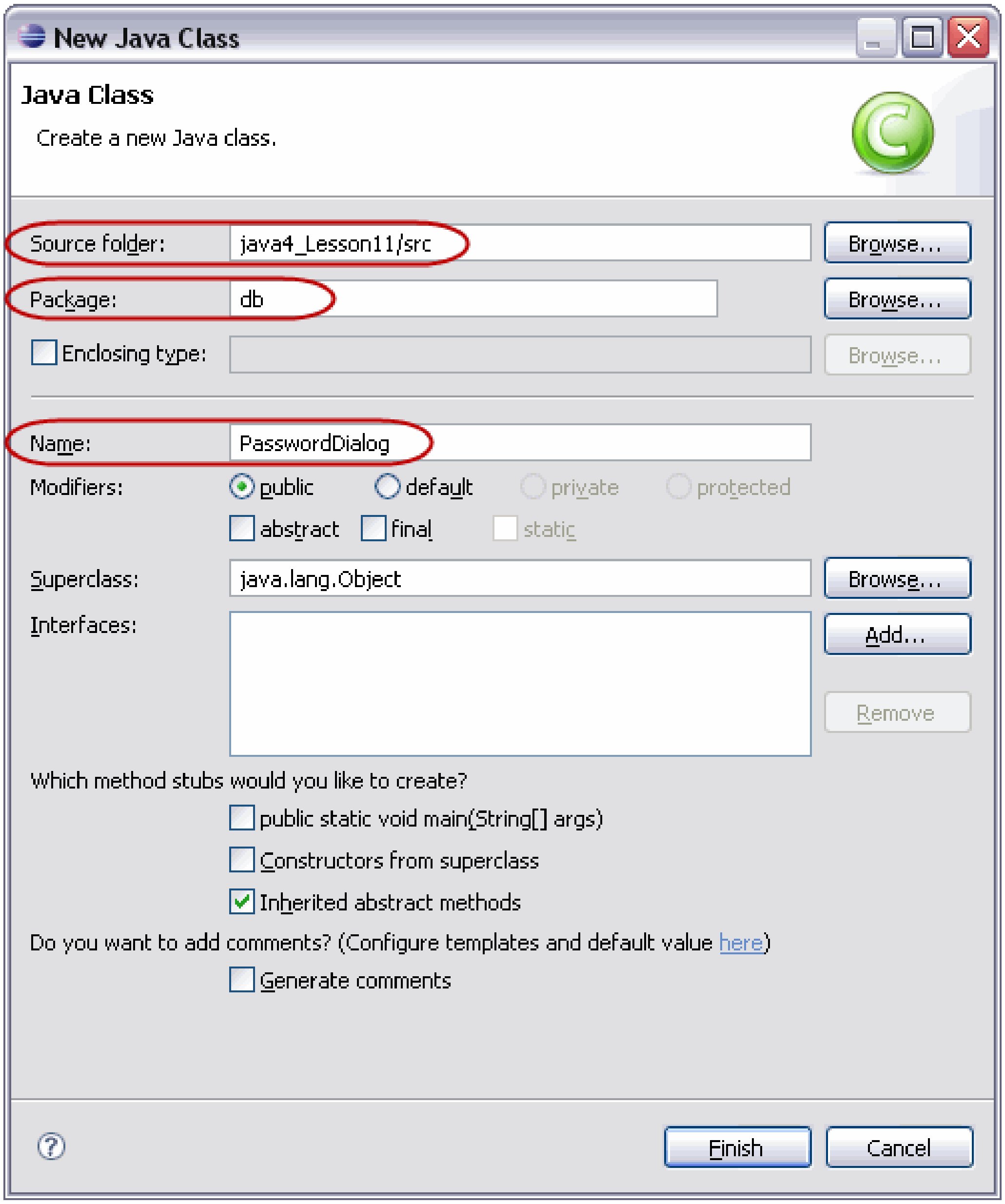
We used a wild card (*) in the **SELECT** query to retrieve all of the entries from our data. Explore more options using **SELECT** with these resources:

- Oracle's JDBC Introduction.
- More from Oracle on the Statement class methods.
- The SQL SELECT link from Key Data.
- One of many books available on SQL.

Logging In

So now we have a running application, but what user wants to use the command line to enter commands? We'll begin to fix this problem by creating a tool to log in. In the next lesson, we'll retool the whole application to take advantage of the information we now know about Swing.

In the java4_Lesson11 project, create a **PasswordDialog** class:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson11/src

Package: db

□ Enclosing type:

Name: PasswordDialog

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **PasswordDialog** as shown in **blue**:

CODE TO TYPE: PasswordDialog

```
package db;

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

class PasswordDialog extends JDialog implements ActionListener {
    private JTextField user;
    private JPasswordField password;
    private String username, passwd;
    private static String [] info;
    private static boolean set = false;

    public PasswordDialog(final JFrame owner) {
        // set the dialog title and size
        super(owner, "Login", true);
        setSize(280, 150);
        user = new JTextField(10);
        user.addActionListener(this);
        password = new JPasswordField(10);
        password.addActionListener(this);
        // Create the center panel which contains the fields for entering information
        JPanel center = new JPanel();
        center.setLayout(new GridLayout(3, 2));     // 3 rows leaves a nice space betwe
en
        center.add(new JLabel(" Enter UserName:"));
        center.add(user);
        center.add(new JLabel(" Enter Password:"));
        center.add(password);
        // Create the south panel which contains the buttons
        JPanel south = new JPanel();
        JButton submitButton = new JButton("Submit");
        submitButton.setActionCommand("SUBMIT");
        submitButton.addActionListener(this);

        JButton helpButton = new JButton("Help");
        south.add(submitButton);
        south.add(helpButton);
        // Add listeners to the buttons
        helpButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent aEvent) {  // The user has asked fo
r help
                JOptionPane.showMessageDialog(owner,
                "Your username and password are the same as those\n" +
                "you use to access your O'Reilly School of Technology courses.\n");
            }
        });
        // Add the panels to the dialog window
        Container contentPane = getContentPane();
        contentPane.add(center, BorderLayout.CENTER);
        contentPane.add(south,  BorderLayout.SOUTH);
    }

    public void actionPerformed(ActionEvent e) {
        String cmd = e.getActionCommand();
        if ("SUBMIT".equals(cmd))
        {                               // Process the inputs.
            username = user.getText();
            char[] input = password.getPassword();
            passwd = new String(input);
            // to verify it is working, print the name and password--remove this line l
ater!
            System.out.println("User is " + username + ", password is " + passwd);
            info = new String[2];
            info[0] = username;
            info[1] = passwd;
            set = true;       // now can send info back
            dispose();
        }
    }

    public static void main(String [] args) {  // create the frame first and then give
it that frame as owner
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        final PasswordDialog addPassword = new PasswordDialog(frame);
        addPassword.setVisible(true);
        System.exit(0);  // terminates
    }
}
```

Save and run it. That's much more efficient than going into Eclipse frames to set variables. Of course, normally we wouldn't just run a Login Dialog and stop there, so the **main()** method might seem a little weird as it is.

Let's add a method to access the **PasswordDialog** from other classes, and change it so it doesn't print the username and password.

Note This example does not address security issues--it simply passes the information as elements in an array.

Edit **PasswordDialog** as shown in **blue**:

CODE TO EDIT: PasswordDialog

```
package db;

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

class PasswordDialog extends JDialog implements ActionListener{
    private JTextField user;
    private JPasswordField password;
    private String username, passwd;
    private static String [] info;
    private static boolean set = false;

    public PasswordDialog(final JFrame owner) {
        // set the dialog title and size
        super(owner, "Login", true);
        setSize(280, 150);
        user = new JTextField(10);
        user.addActionListener(this);
        password = new JPasswordField(10);
        password.addActionListener(this);
        // Create the center panel which contains the fields for entering information
        JPanel center = new JPanel();
        center.setLayout(new GridLayout(3, 2));      // 3 rows leaves a nice space betw
een
        center.add(new JLabel(" Enter UserName:"));
        center.add(user);
        center.add(new JLabel(" Enter Password:"));
        center.add(password);
        // Create the south panel which contains the buttons
        JPanel south = new JPanel();
        JButton submitButton = new JButton("Submit");
        submitButton.setActionCommand("SUBMIT");
        submitButton.addActionListener(this);

        JButton helpButton = new JButton("Help");
        south.add(submitButton);
        south.add(helpButton);
        // Add listeners to the buttons
        helpButton.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent aEvent) {   // The user has asked f
or help.
                JOptionPane.showMessageDialog(owner,
                "Your username and password are the same as those\n" +
                "you use to access your O'Reilly School of Technology courses.\n");
            }
        });
        // Add the panels to the dialog window
        Container contentPane = getContentPane();
        contentPane.add(center, BorderLayout.CENTER);
        contentPane.add(south,  BorderLayout.SOUTH);
    }

    public void actionPerformed(ActionEvent e) {
        String cmd = e.getActionCommand();
        if ("SUBMIT".equals(cmd))
        {                                     // Process the inputs.
            username = user.getText();
            char[] input = password.getPassword();
            passwd = new String(input);
            // to verify it is working, uncomment this line
            //System.out.println("User is " + username + " password is " + passwd);
            info = new String[2];
            info[0] =username;
            info[1] = passwd;
            set = true;                       // now can send info back
            dispose();
        }
    }

    public static void main(String [] args){  // create the frame first and then give i
t that frame as owner
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        final PasswordDialog addPassword = new PasswordDialog(frame);
        addPassword.setVisible(true);
    }

    public static String [] login(Object sender) {                              //  ob
ject who requested login is the  sender;
        JFrame frame = new JFrame();                                             // at
tempt is to make as reusable as possible
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        final PasswordDialog addPassword = new PasswordDialog(frame);
        addPassword.setVisible(true);
        while (!set)                                                              // w
ait until user has put information in before returning values
            try {
                Thread.sleep(5000);
            }
            catch (InterruptedException e) {};
        return info;
    }
}
```

Save it. Now, let's link it to our database. Edit **PhoneBook** as shown in **blue** below:

CODE TO EDIT: PhoneBook

```
package db;

public class PhoneBook {

    public PhoneBook(){
        String [] info = PasswordDialog.login(this);                             // st
atic login so can call from class
        DatabaseManager databaseManager = new DatabaseManager(info[0], info[1]);  // Cr
eate the database manager and pass login info.
        UserInterface userInterface = new UserInterface(databaseManager );        // Cr
eate access for user input
        userInterface.start();
    }

    public static void main ( String[] args ) {
 // instantiate to start
                        // args[1] must be the password to connect to the mysql databas
e
        PhoneBook myApp = new PhoneBook();
    }
}
```

Save and run it.

For other examples of improved login dialogs, see Swing Components and Oracle's Java Look and Feel Design Guidelines book (both are copyright-protected so we can't provide the code for them here).

You're doing great! We've unearthed a lot of valuable stuff so far!

Database Application With GUI

Refining the Application

Our last example demonstrated the general concepts used to implement JDBC, and some of the fundamental elements you encounter when writing database applications:

- **Creating a database**: create the database outside of Java, with tools supplied by the database vendor, or with SQL statements fed to the database from a Java program.
- **Connecting to a data source**: use a bridge to connect to the data source. (you can learn how to connect to databases on Windows machines using the JDBC/ODBC Bridge.)
- **Inserting information into a database**: either enter data outside of Java, using database-specific tools, or with SQL statements sent by a Java program.
- **Selectively retrieving information**: use SQL commands from Java to get results and then use Java to display or manipulate that data.

In this lesson, we'll enable the phonebook application to:

1. create the database table using SQL that was written into the application initially to populate a table.
2. provide the user with a graphical user interface that:
    - has a graphical display.
    - allows users to search the table for desired entries.
    - allows users to edit the database table (add and delete).

Improving the Appearance

Graphical user interfaces make an application more visually appealing, but they come at a programming cost. You must program every aspect you want to allow the user to experience. But even though you'll write lots of code, the end result may not look particularly impressive to the untrained eye. Such is the thankless life of the Java programmer. Your reward will be found in the satisfaction of knowing you've written amazing, clean code.

Copying an Existing Class

Create a new **java4_Lesson13** project. If you're given the option to **Open Associated Perspective**, click **No**. Right-click your new project and select **New | Package**. For the Name, enter **greenDB** (this code was provided courtesy of David Green).

Copy the **PasswordDialog** class from the previous Project (java4_Lesson11) and paste it into the java4_Lesson13 project, greenDB package.

Open the new copy of **PasswordDialog** in the editor to verify that it contains package **greenDB**.

▶ Save and run it. We aren't running it from another application, we're just checking the Login dialog itself.

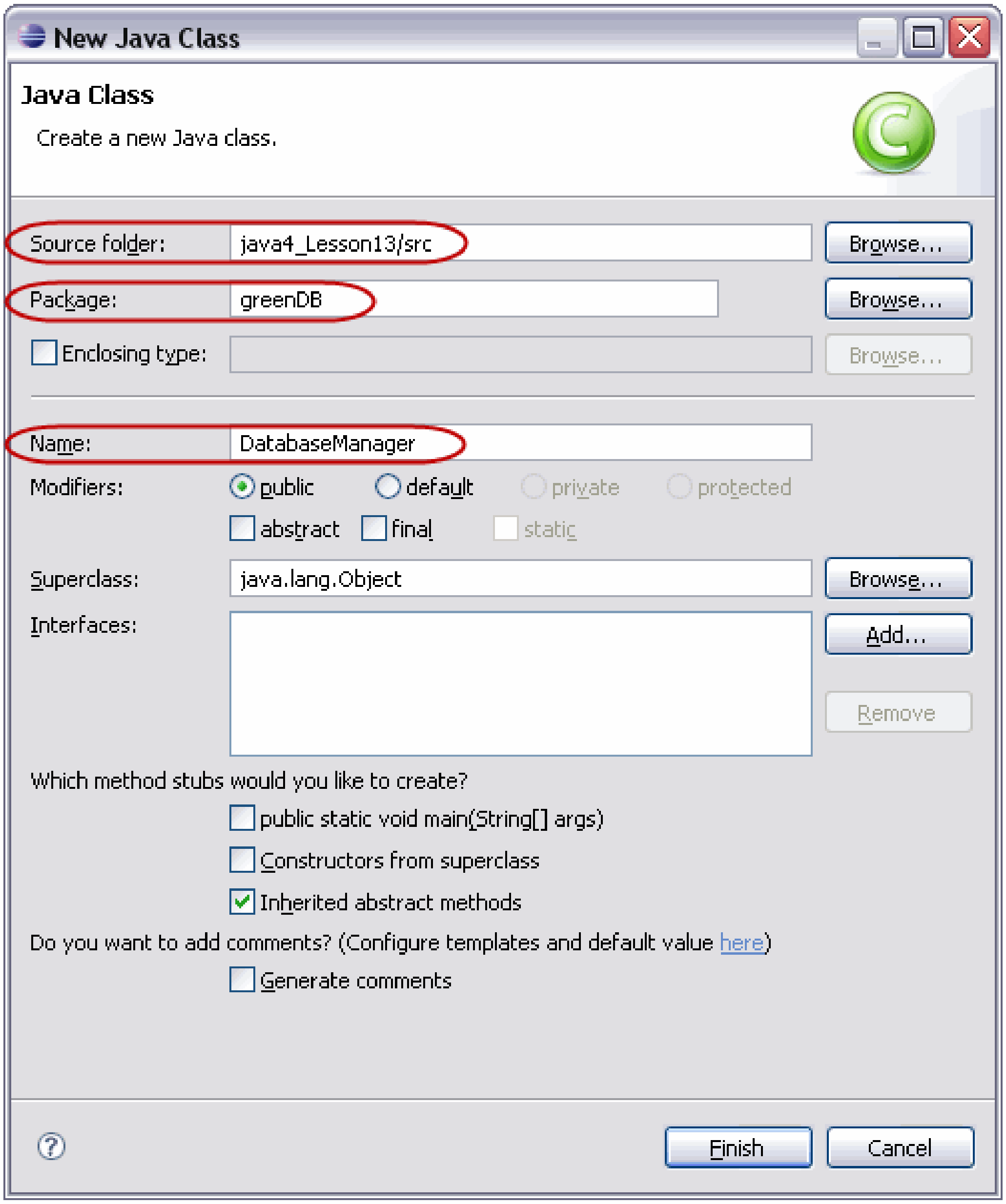
Creating New Classes

The first new class for this application will connect it to the database. Although the JDBC provides more advanced features, you'll notice that the basic elements used when working with database are consistent. Most JDBC code that "talks" to a database looks similar.

As you type this class, you may notice some minor changes. For example, some variable and method names have been changed.

The **inspectForTable()** method has been made more general; it passes the name of the table you are looking for rather than having it hard-coded. This is always a better choice for reusable code.

In the java4_Lesson13 project, create a **DatabaseManager** class as shown. (Because it is in a different package, it can have the same name as the class used in previous lessons):



Source folder: java4_Lesson13/src

Package: greenDB

□ Enclosing type:

Name: DatabaseManager

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **DatabaseManager** as shown in **blue**:

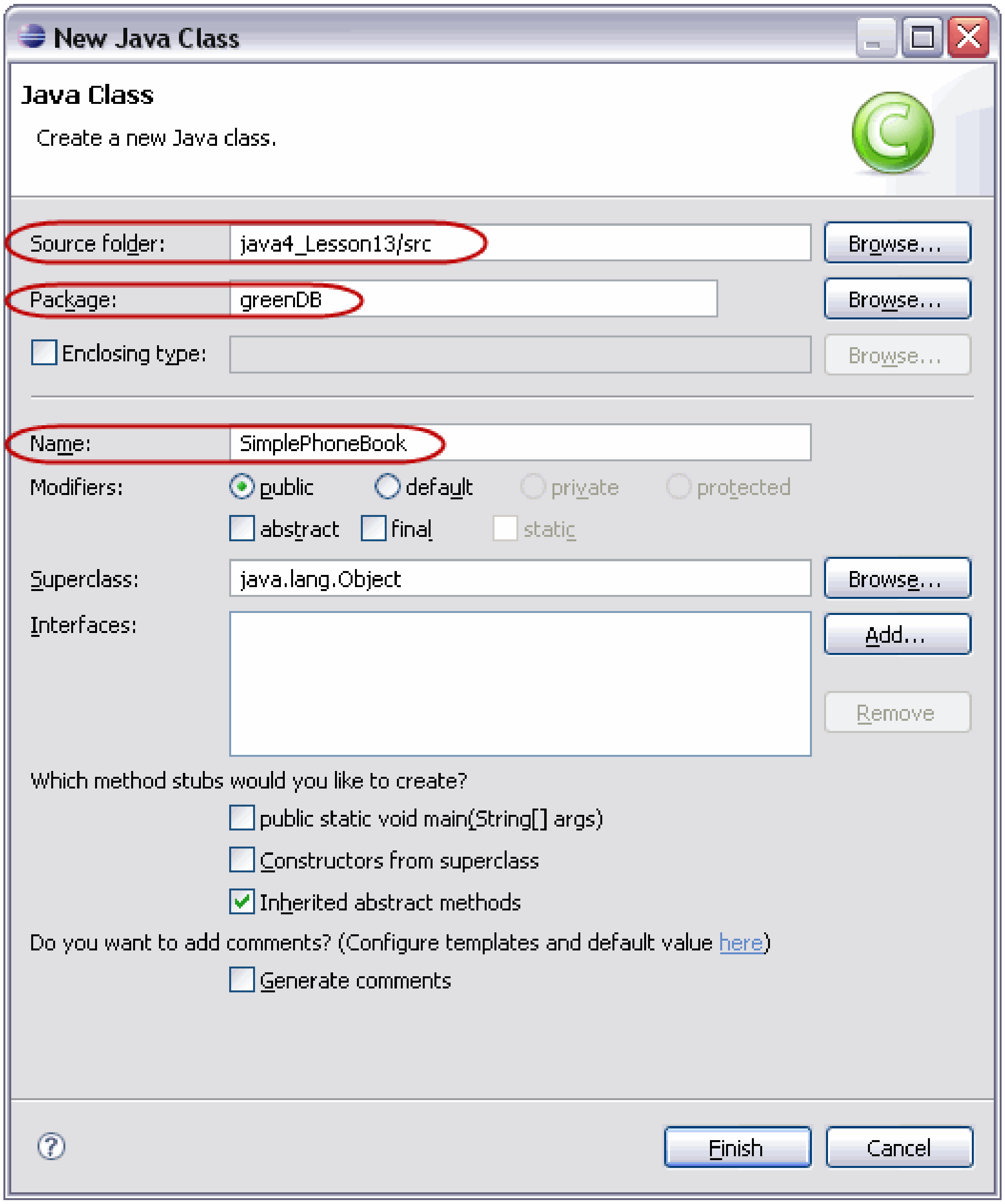
CODE TO TYPE: DatabaseManager

```
package greenDB;

import java.sql.*;

public class DatabaseManager {
    private Connection conn;
    private Statement stmt;
    private ResultSet rset;

    public DatabaseManager (String username, String password) {  // the construc
tor for the database manager
        // Connect to database and execute the SQL commands for creating and ini
tializing the Listings table.
        try {
            Class.forName ("com.mysql.jdbc.Driver");  // Load the MySQL JDBC dri
ver
        }
        catch (ClassNotFoundException e) {
            System.out.println("Failed to load JDBC/ODBC driver.");
            e.printStackTrace();
            return;
        }

        try {
            // Connect to the database.
            // Give the whole URL as a parameter rather than using a variable
            conn = DriverManager.getConnection("jdbc:mysql://sql.useractive.com:
3306/" + username, username, password);
            stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, Resul
tSet.CONCUR_UPDATABLE);   // Create a Statement
            // Execute the creation and initialization of table query
            DatabaseMetaData aboutDB = conn.getMetaData();
            String [] tableType = {"TABLE"};
            ResultSet rs = aboutDB.getTables(null, null, "Listings",  tableType)
;
            if (!inspectForTable (rs, "Listings")) {     // Find out if the tabl
e is already there
                // there is no table--make it from the initialization listing
                String [] SQL = initListingsTable();     // code for this method
 is below
                for (int i=0; i < SQL.length; i++)
                {
                    stmt.execute(SQL[i]);
                }
            }
        }catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private String [] initListingsTable() {
        // Executable SQL commands for creating Listings table
        // inserting initial names and phone numbers.
        String[]  SQL = {
            "create table Listings (" +
            "LAST_NAME  varchar (16)," +
            "FIRST_NAME varchar (16)," +
            "AREA_CODE  varchar(3)," +
            "PREFIX     varchar(3)," +
            "SUFFIX     varchar(4))",
            "insert into Listings values ('ANDERSON', 'JOHN',  '314', '825', '16
95')",
            "insert into Listings values ('CABLES',   'WALLY', '212', '434', '96
85')",
            "insert into Listings values ('FRY',      'EDGAR', '415', '542', '58
85')",
            "insert into Listings values ('MARTIN',   'EDGAR', '665', '662', '90
01')",
            "insert into Listings values ('TUCKER',   'JOHN',  '707', '696', '85
41')",
        };
        return SQL;
    }

    private boolean inspectForTable (ResultSet rs, String tableName)  throws SQL
Exception {  // exception will be caught when method is used
        int i;
        ResultSetMetaData rsmd = rs.getMetaData ();  // Get the ResultSetMetaDat
a to use for the column headings
        int numCols = rsmd.getColumnCount ();        // Get the number of column
s in the result set

        boolean more = rs.next ();
        while (more) {                               // Get each row, fetching u
ntil end of the result set
            for (i=1; i<=numCols; i++) {
                if (rsmd.getColumnLabel(i) == "TABLE_NAME")   // Loop through ea
ch row, getting the column data looking for Tables
                    if  (rs.getString(i).equals(tableName))   // If the column i
s the TABLE_NAME, is it the one we are looking for?
                    {
                        System.out.println("Found one that equals " + rs.getStri
ng(i));
                        return true;
                    }
            }
            System.out.println("");
            more = rs.next ();              // Fetch the next result set row
        }
        return false;
    }

    public void doGetQuery(String query) {  // rather than the "getEntries" of t
he previous example
        try {
            rset = stmt.executeQuery(query);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public void doInsertQuery(String query) {   // rather than the hard-coded "a
ddEntry" of the previous example
        try {
            stmt.executeUpdate(query);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public ResultSet getResultSet() {  // a new method that will let the GUI get
 the resultSet to manipulate it
        return rset;
    }

    public void close(boolean remove){  // closes all open connections

        try {
            if (remove)
                stmt.execute ("drop table Listings;");

            stmt.close();
            conn.close();
        }
        catch ( SQLException e ) {
            System.out.println ("\n*** SQLException caught ***\n");
            e.printStackTrace();
        }
    }
}
```

Save it (there's nothing to run yet).

We need a class to instantiate and start this application. We'll create the necessary class, but it won't be ready for consumption until we make the GUI. Please be patient--we need all the ingredients before we can cook!

In the java4_Lesson13 project, create **SimplePhoneBook** as shown:



Java Class

Create a new Java class.

Source folder: java4_Lesson13/src

Package: greenDB

□ Enclosing type:

Name: SimplePhoneBook

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **SimplePhoneBook** as shown in **blue**:

CODE TO TYPE: SimplePhoneBook

```
package greenDB;

import javax.swing.JFrame;

public class SimplePhoneBook {
    public static void main(String args[]) {   // Instantiate the phone book fra
me window and display it.
        PhoneBookFrame pbFrame = new PhoneBookFrame();
        pbFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pbFrame.setVisible(true);
    }
}  // End SimplePhoneBook class
```

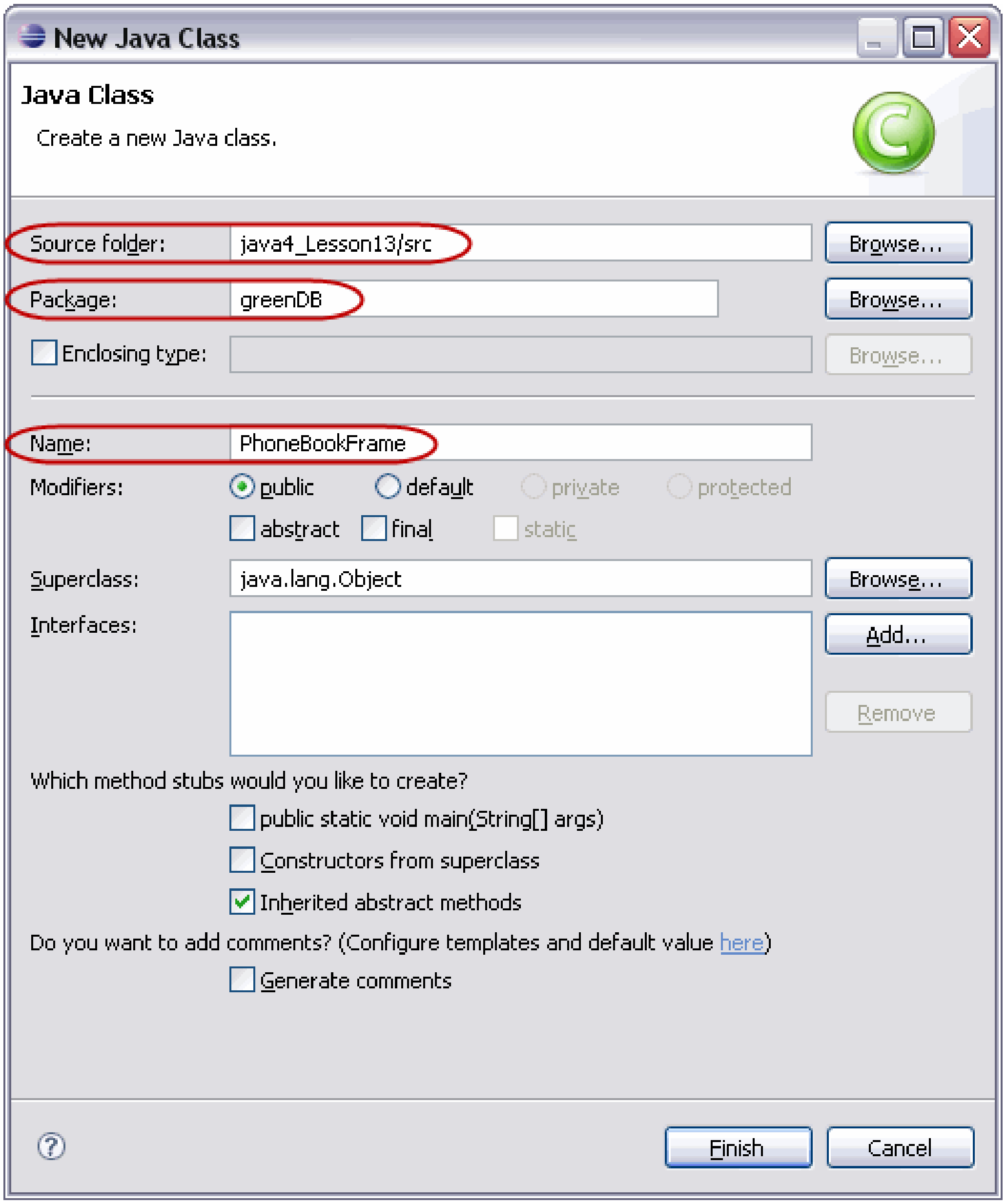
💾 Even though there are errors, save it. The errors are there because we haven't defined the GUI and its components yet.

Creating the View

Now we need to create the **JFrame** for our application.

Click here to see what the PhoneBookFrame will look like, so you can compare the code and the GUI as you write the code.

In the java4_Lesson13 project, create a **PhoneBookFrame** class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson13/src

Package: greenDB

□ Enclosing type:

Name: PhoneBookFrame

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **PhoneBookFrame** as shown in **blue**:

CODE TO TYPE: PhoneBookFrame

```
package greenDB;

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

class PhoneBookFrame extends JFrame {
    /** The initial user interface width, in pixels */
    private static final int WIDTH  = 577;
    /** The initial user interface height, in pixels */
    private static final int HEIGHT = 466;
    /** Provides methods for displaying a SQL result set in a JTable */
    // Commented out for now so the program can run without it.
    // private ListingsTableModel tblModel;
    /** Used to display the SQL result set in a cell format */
    private JTable table;
    /** A scrollable view for the SQL result set */
    private JScrollPane scrollPane;
    /** A text field for entering the phone listing's last name */
    private JTextField lNameField    = new JTextField(10);
    /** A text field for entering the phone listing's first name */
    private JTextField fNameField    = new JTextField(10);
    /** A text field for entering the phone listing's area code. The value in parenthes
es
    is the number of columns (NOT necessarily characters) to allow for the field. */
    private JTextField areaCodeField = new JTextField(2);
    /** A text field for entering the phone listing's prefix */
    private JTextField prefixField   = new JTextField(2);
    /** A text field for entering the phone listing's extension */
    private JTextField suffixField   = new JTextField(3);
    /** Database Operations */
    private DatabaseManager myDB;

    public PhoneBookFrame() {
        String [] info = PasswordDialog.login(this);  // static login so can call from
class
        // create and initialize the listings table
        myDB = new DatabaseManager(info[0], info[1]);
        // should have access so make GUI
        JButton getButton = new JButton("Get");  // get the listing
        JButton add       = new JButton("+");    // add a listing
        JButton rem       = new JButton("-");    // remove a listing
        JLabel  space     = new JLabel(" ");
        // set the window size and title
        setTitle("Simple Phone Book");
        setSize(WIDTH, HEIGHT);
        // if user presses Enter, get button pressed
        getRootPane().setDefaultButton(getButton);
        // create the panel for looking up listing
        JPanel south = new JPanel();
        south.setLayout(new FlowLayout(FlowLayout.LEFT));

        south.add(new JLabel("Last:"));
        south.add(lNameField);
        south.add(new JLabel(" First:"));
        south.add(fNameField);
        south.add(new JLabel("  Phone:  ("));
        south.add(areaCodeField);
        south.add(new JLabel(") "));
        south.add(prefixField);
        south.add(new JLabel("-"));
        south.add(suffixField);
        south.add(new JLabel("   "));
        south.add(getButton);
        // create the panel for adding and deleting listings
        JPanel  east           = new JPanel();
        GridBagLayout gb       = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        east.setLayout(gb);
        add.setFont(new Font("SansSerif", Font.BOLD, 12));
        rem.setFont(new Font("SansSerif", Font.BOLD, 12));

        gbc.fill = GridBagConstraints.BOTH;
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gb.setConstraints(add, gbc);
        gb.setConstraints(space, gbc);
        gb.setConstraints(rem, gbc);
        east.setLayout(gb);
        east.add(add);
        east.add(space);
        east.add(rem);

        // add the panels
        Container contentPane = getContentPane();
        contentPane.add(south, BorderLayout.SOUTH);
        contentPane.add(east, BorderLayout.EAST);
        // Add listeners
        // When the application closes, drop the Listings table and close the connection to MySQL
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent wEvent) {
             myDB.close(false);  // We will want to save our additions to the PhoneBook, so don't drop table
            }
        });

        // when the UI first displays, do an empty lookup so the center panel doesn't look funny
        getButton.doClick();
        lNameField.requestFocus();  // set focus to last name field (most common lookup)
    }

    public DatabaseManager getDBManager() {
     return myDB;
    }
}  // End PhoneBookFrame class
```

Save and run it (from **SimplePhoneBook**).

Whoops! We haven't added the Driver to this package yet--do it now:

Terminate the current running process from the console. Right-click the **java4_Lesson13** Project and select **Build Path | Add External Archives** to open the file browser so you can get the driver. choose **Build Path | Add External Archives**, which opens the file browser for you to get the driver. Again, in the file dialog, start to type the path **C:\jdbc\mysql-connector-java-5.1.5-bin.jar**. The auto-complete feature should allow you to press Tab to fill in the file name **mysql-connector-java-5.1.5-bin.jar**. Then, click **Open**.

Run the **SimplePhoneBook** class.

We've made a nice little user interface here, but it may not seem that impressive yet because we have not added listeners. So far, the only component on the **PhoneBookFrame** that listens at all is the Window. At least we can close it.

Close the Window by clicking the X in the upper right corner of the application window.

Creating Controllers for the View

We have quite a few components here. Let's arrange them so we can see the table **Listing** we created through the **DatabaseManager**.

Edit **PhoneBookFrame** as shown in **blue**:

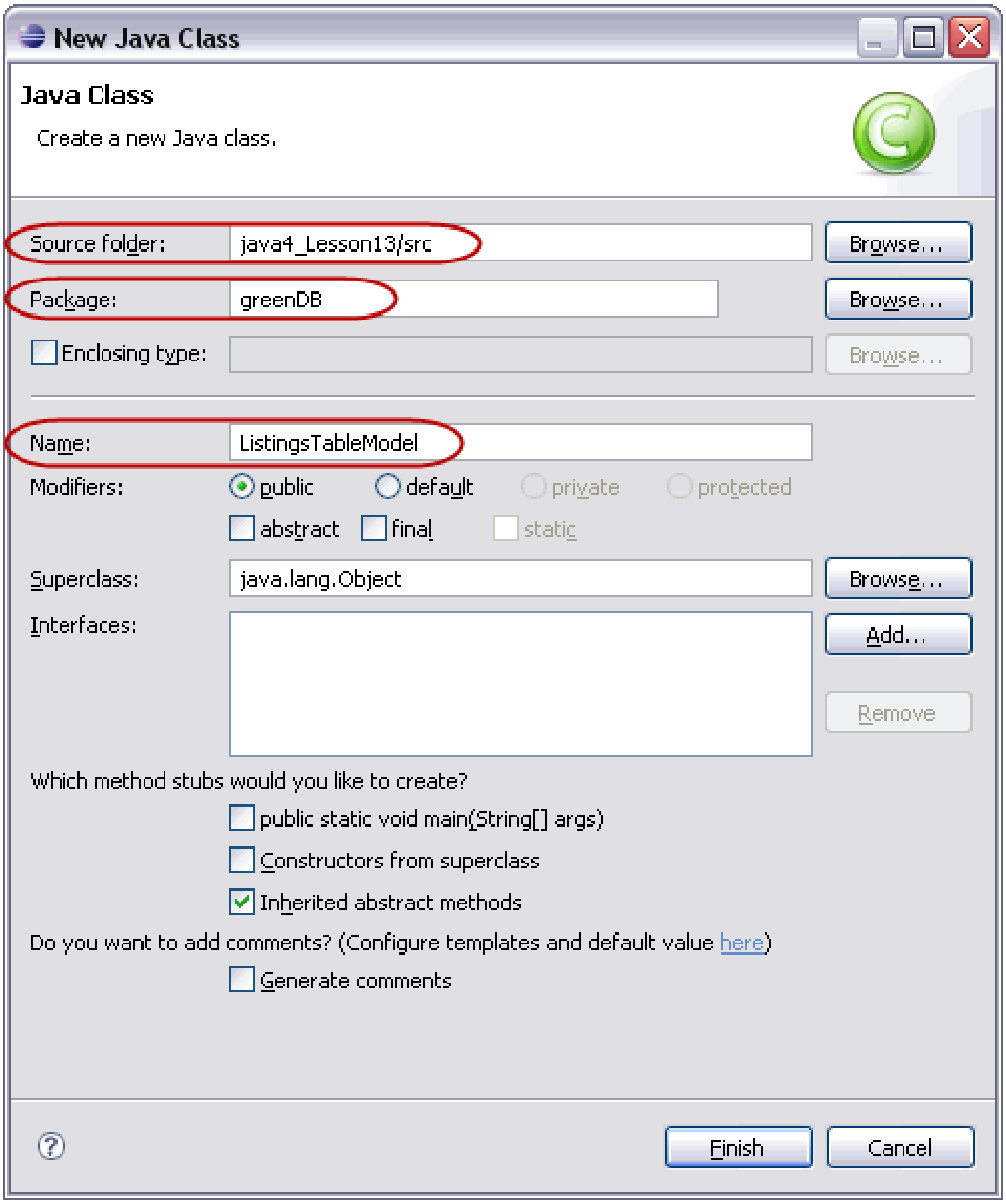
CODE TO EDIT: PhoneBookFrame

```
package greenDB;

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

class PhoneBookFrame extends JFrame {
    /** The initial user interface width, in pixels */
    private static final int WIDTH  = 577;
    /** The initial user interface height, in pixels */
    private static final int HEIGHT = 466;
    /** Provides methods for displaying a SQL result set in a JTable */
    // Commented out for now so the program can run without it.
    // private ListingsTableModel tblModel;
    /** Used to display the SQL result set in a cell format */
    private JTable table;
    /** A scrollable view for the SQL result set */
    private JScrollPane scrollPane;
    /** A text field for entering the phone listing's last name */
    private JTextField lNameField    = new JTextField(10);
    /** A text field for entering the phone listing's first name */
    private JTextField fNameField    = new JTextField(10);
    /** A text field for entering the phone listing's area code */
    private JTextField areaCodeField = new JTextField(2);
    /** A text field for entering the phone listing's prefix */
    private JTextField prefixField   = new JTextField(2);
    /** A text field for entering the phone listing's extension */
    private JTextField suffixField   = new JTextField(3);
    /** Database Operations */
    private DatabaseManager myDB;

    public PhoneBookFrame() {
        String [] info = PasswordDialog.login(this);  // static login so can call from
class
        // create and initialize the listings table
        myDB = new DatabaseManager(info[0], info[1]);
        // Should have access so make GUI
        JButton getButton = new JButton("Get");  // get the listing
        JButton add       = new JButton("+");    // add a listing
        JButton rem       = new JButton("-");    // remove a listing
        JLabel  space     = new JLabel(" ");
        // set the window size and title
        setTitle("Simple Phone Book");
        setSize(WIDTH, HEIGHT);
        // if user presses enter, get button pressed
        getRootPane().setDefaultButton(getButton);
        // create the panel for looking up listing
        JPanel south = new JPanel();
        south.setLayout(new FlowLayout(FlowLayout.LEFT));

        south.add(new JLabel("Last:"));
        south.add(lNameField);
        south.add(new JLabel(" First:"));
        south.add(fNameField);
        south.add(new JLabel("  Phone:  ("));
        south.add(areaCodeField);
        south.add(new JLabel(") "));
        south.add(prefixField);
        south.add(new JLabel("-"));
        south.add(suffixField);
        south.add(new JLabel("   "));
        south.add(getButton);

        // create the panel for adding and deleting listings
        JPanel  east           = new JPanel();
        GridBagLayout gb       = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        east.setLayout(gb);
        add.setFont(new Font("SansSerif", Font.BOLD, 12));
        rem.setFont(new Font("SansSerif", Font.BOLD, 12));

        gbc.fill = GridBagConstraints.BOTH;
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gb.setConstraints(add, gbc);
        gb.setConstraints(space, gbc);
        gb.setConstraints(rem, gbc);
        east.setLayout(gb);
        east.add(add);
        east.add(space);
        east.add(rem);

        // add the panels
        Container contentPane = getContentPane();
        contentPane.add(south, BorderLayout.SOUTH);
        contentPane.add(east, BorderLayout.EAST);

        // Add listeners

        // When the application closes, drop the Listings table and close the connection to MySQL
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent wEvent) {
             myDB.close(false);
            }
        });

        getButton.addActionListener(new GetListener());  // Add the listener for the getButton (GetListener inner class defined below)
        // when the UI first displays, do an empty lookup so the center panel doesn't look funny
        getButton.doClick();
        lNameField.requestFocus();    // set focus to last name field (most common lookup)
    }

    public DatabaseManager getDBManager(){
        return myDB;
    }

    /* inner class GetListener */
    class GetListener implements ActionListener {  // Gets the entries from the text fields

        public void actionPerformed(ActionEvent aEvent) {
            // Get whatever the user entered, trim any white space and change to upper case
            String last  = lNameField.getText().trim().toUpperCase();
            String first = fNameField.getText().trim().toUpperCase();
            String ac    = areaCodeField.getText().trim().toUpperCase();
            String pre   = prefixField.getText().trim().toUpperCase();
            String sfx   = suffixField.getText().trim().toUpperCase();

            // Replace any single quote chars w/ space char or SQL will think the ' is the end of the string
            last  = last.replace('\'', ' ');
            first = first.replace('\'', ' ');
            ac    = ac.replace('\'', ' ');
            pre   = pre.replace('\'', ' ');
            sfx   = sfx.replace('\'', ' ');
            // Get rid of the last result displayed if there is one
            if(scrollPane != null)
                getContentPane().remove(scrollPane);
            // Only execute the query if one or more fields have data, else just displa
```

```
y an empty table
            if(last.length()  > 0 ||
             first.length() > 0 ||
             ac.length()    > 0 ||
             pre.length()   > 0 ||
             sfx.length()   > 0) {
                // build the query and execute it. Provide the results to the table mod
el
                myDB.doGetQuery(buildQuery(last, first, ac, pre, sfx));
                ResultSet rset = myDB.getResultSet();
                tblModel = new ListingsTableModel(rset);
                table = new JTable(tblModel);
            } else {
                table = new JTable();
            }
            // Allows the user to only delete one record at a time
            table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
            // Add the table with the results to the contentPane and display it.
            scrollPane = new JScrollPane(table);
            getContentPane().add(scrollPane, BorderLayout.CENTER);
            pack();
            doLayout();
        }

        public String buildQuery(String last, String first, String ac, String pre, Stri
ng sfx) {
            String whereClause = " where";
            // Build the where clause
            if(last.length() > 0)
                whereClause += (" LAST_NAME = '" + last + "'");

            if(first.length() > 0) {
                if(whereClause.length() > 6)
                    whereClause += " AND";
                whereClause += (" FIRST_NAME = '" + first + "'");
            }

            if(ac.length() > 0) {
                if(whereClause.length() > 6)
                    whereClause += " AND";
                whereClause += (" AREA_CODE = '" + ac + "'");
            }

            if(pre.length() > 0) {
                if(whereClause.length() > 6)
                    whereClause += " AND";
                whereClause += (" PREFIX = '" + pre + "'");
            }

            if(sfx.length() > 0) {
                if(whereClause.length() > 6)
                    whereClause += " AND";
                whereClause += (" SUFFIX = '" + sfx + "'");
            }

            return "select LAST_NAME, FIRST_NAME, AREA_CODE, PREFIX, SUFFIX from Listin
gs" + whereClause;
        }
    }   // End GetListener inner class
}
```

We've got a few errors, all referring to a **ListingTablesModel**. We need to define this **ListingTablesModel** so that our GUI can display our entries in a table format.

In the java4_Lesson13 project, create a **ListingsTableModel** class as shown:



Java Class

Create a new Java class.

Source folder: java4_Lesson13/src

Package: greenDB

□ Enclosing type:

Name: ListingsTableModel

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **ListingsTableModel** as shown in **blue**:

CODE TO TYPE: ListingsTableModel

```
package greenDB;

import java.sql.ResultSet;
import java.sql.SQLException;
import javax.swing.table.AbstractTableModel;

class ListingsTableModel extends AbstractTableModel {
    /** The result set from the Listings table to be displayed */
    private ResultSet rs;

    public ListingsTableModel(ResultSet rs) {
        this.rs = rs;
    }

    public int getRowCount() {
        try {
            rs.last();
            return rs.getRow();
        } catch (SQLException e) {
            e.printStackTrace();
            return 0;
        }
    }

    public int getColumnCount() {
        return 3;
    }

    public String getColumnName(int column) {
        try {
            String colName = rs.getMetaData().getColumnName(column + 1);
            // Return column names that look better than the database column names.
            // Since getColumnCount always returns 3, we only look for first 3 columns
in
            // the result set.
            if(colName.equals("LAST_NAME"))
                return "Last Name";
            else if(colName.equals("FIRST_NAME"))
                return "First Name";
            else if(colName.equals("AREA_CODE"))
                return "Phone Number";
            else return colName;      // Should never get here.

        } catch (SQLException e) {
            e.printStackTrace();
            return "";
        }
    }

    public Object getValueAt(int row, int column) {
        try {
            rs.absolute(row + 1);
            // for the 3rd column in the results, combine all of the phone number field
s for output
            if(column == 2)
                return "(" + rs.getObject(column + 1) + ") " + rs.getObject(column + 2)
 + "-" + rs.getObject(column + 3);
            else
                return rs.getObject(column + 1);
        } catch (SQLException e) {
            e.printStackTrace();
            return null;
        }
    }
}  // End ListingsTableModel class
```

API Go to **java.sql.ResultSet** and read over the methods invoked in this class, and all of the **get*x*()** methods therein:

- **last()**
- **getRow()**
- **getMetaData()**
- **absolute(int)**
- **getObject(int)**

💾 Save ListingsTableModel.

Go back to the **PhoneBookFrame** class and its Instance Variables and uncomment the code that declares the ListingsTableModel, as shown in **blue**:

CODE TO EDIT: PhoneBookFrame

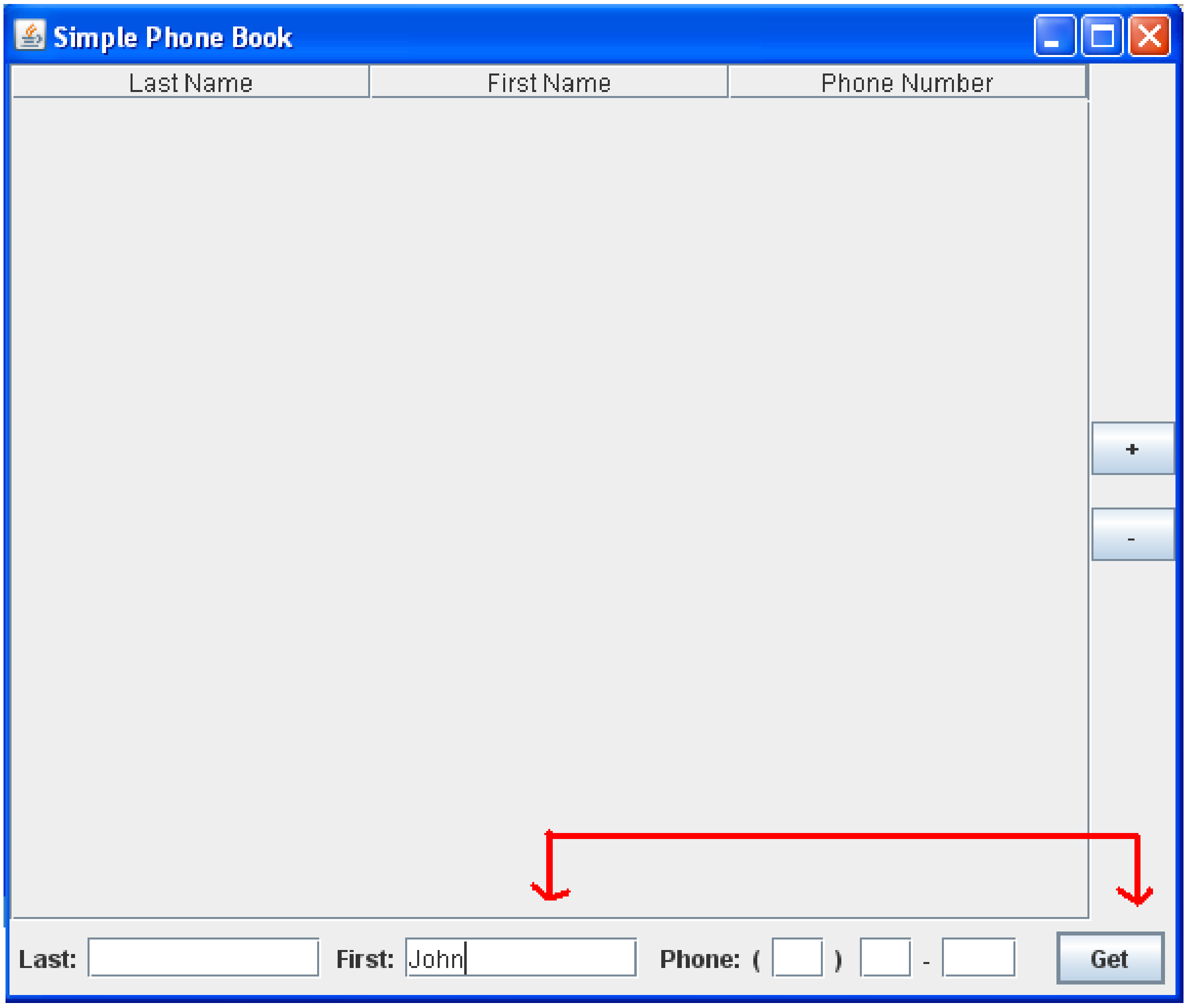
```
package greenDB;

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

public class PhoneBookFrame extends JFrame {
    /** The initial user interface width, in pixels */
    private static final int WIDTH  = 577;
    /** The initial user interface height, in pixels */
    private static final int HEIGHT = 466;
    /** Provides methods for displaying a SQL result set in a JTable */
    private ListingsTableModel tblModel;
    /** Used to display the SQL result set in a cell format */
    private JTable table;
    /** A scrollable view for the SQL result set */
    private JScrollPane scrollPane;
    /** A text field for entering the phone listing's last name */
    private JTextField lNameField    = new JTextField(10);
    /** A text field for entering the phone listing's first name */
    private JTextField fNameField    = new JTextField(10);
    /** A text field for entering the phone listing's area code */
    private JTextField areaCodeField = new JTextField(2);
    /** A text field for entering the phone listing's prefix */
    private JTextField prefixField   = new JTextField(2);
    /** A text field for entering the phone listing's extension */
    private JTextField suffixField   = new JTextField(3);
    /** Database Operations */
    private DatabaseManager myDB;

    public PhoneBookFrame() {
        String [] info = PasswordDialog.login(this);  // static login so can call from
class
        // create and initialize the listings table
        myDB = new DatabaseManager(info[0], info[1]);
        // should have access so make GUI
        JButton getButton = new JButton("Get");  // get the listing
        JButton add       = new JButton("+");    // add a listing
        JButton rem       = new JButton("-");    // remove a listing
        JLabel  space     = new JLabel(" ");
        // set the window size and title
        setTitle("Simple Phone Book");
        setSize(WIDTH, HEIGHT);
        // if user presses Enter, get button pressed
        getRootPane().setDefaultButton(getButton);
        // create the panel for looking up listing
        JPanel south = new JPanel();
        south.setLayout(new FlowLayout(FlowLayout.LEFT));

        south.add(new JLabel("Last:"));
        south.add(lNameField);
        south.add(new JLabel(" First:"));
        south.add(fNameField);
        south.add(new JLabel("  Phone:  ("));
        south.add(areaCodeField);
        south.add(new JLabel(") "));
        south.add(prefixField);
        south.add(new JLabel("-"));
        south.add(suffixField);
        south.add(new JLabel("   "));
        south.add(getButton);
        // create the panel for adding and deleting listings
        JPanel  east           = new JPanel();
        GridBagLayout gb       = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        east.setLayout(gb);
        add.setFont(new Font("SansSerif", Font.BOLD, 12));
        rem.setFont(new Font("SansSerif", Font.BOLD, 12));

        gbc.fill = GridBagConstraints.BOTH;
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gb.setConstraints(add, gbc);
        gb.setConstraints(space, gbc);
        gb.setConstraints(rem, gbc);
        east.setLayout(gb);
        east.add(add);
        east.add(space);
        east.add(rem);

        // add the panels
        Container contentPane = getContentPane();
        contentPane.add(south, BorderLayout.SOUTH);
        contentPane.add(east, BorderLayout.EAST);
        // Add listeners
        // When the application closes, drop the Listings table and close the connectio
n to MySQL
        addWindowListener(
            new WindowAdapter() {
                public void windowClosing(WindowEvent wEvent) {
                    myDB.close(false);  // We will want to save our additions to the Ph
oneBook, so don't drop table
                }
        });

        getButton.addActionListener(new GetListener());  // Add the listener for the ge
tButton (GetListener inner class defined below)

        // when the UI first displays, do an empty lookup so the center panel doesn't l
ook funny
        getButton.doClick();
        lNameField.requestFocus();                                      // set focus to
 last name field (most common lookup)
    }

    public DatabaseManager getDBManager(){
        return myDB;
    }

    /* inner class GetListener */
    class GetListener implements ActionListener {  // Gets the entries from the text fi
elds

        public void actionPerformed(ActionEvent aEvent) {
            // Get whatever the user entered, trim any white space and change to upper
case
            String last  = lNameField.getText().trim().toUpperCase();
            String first = fNameField.getText().trim().toUpperCase();
            String ac    = areaCodeField.getText().trim().toUpperCase();
            String pre   = prefixField.getText().trim().toUpperCase();
            String sfx   = suffixField.getText().trim().toUpperCase();

            // Replace any single quote chars w/ space char or SQL will think the ' is
the end of the string
            last  = last.replace('\'', ' ');
            first = first.replace('\'', ' ');
            ac    = ac.replace('\'', ' ');
            pre   = pre.replace('\'', ' ');
            sfx   = sfx.replace('\'', ' ');
            // Get rid of the last result displayed if there is one
            if(scrollPane != null)
                getContentPane().remove(scrollPane);
            // Only execute the query if one or more fields have data, else just displa
y an empty table
```

```
            if(last.length()  > 0 ||
             first.length() > 0 ||
             ac.length()    > 0 ||
             pre.length()   > 0 ||
             sfx.length()   > 0) {
                // build the query and execute it. Provide the results to the table mod
el
                myDB.doGetQuery(buildQuery(last, first, ac, pre ,sfx));
                ResultSet rset = myDB.getResultSet();
                tblModel = new ListingsTableModel(rset);
                table = new JTable(tblModel);
            } else {
                table = new JTable();
            }
            // Allows the user to only delete one record at a time
            table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
            // Add the table with the results to the contentPane and display it.
            scrollPane = new JScrollPane(table);
            getContentPane().add(scrollPane, BorderLayout.CENTER);
            pack();
            doLayout();
        }

        public String buildQuery(String last, String first, String ac, String pre, Stri
ng sfx) {
            String whereClause = " where";
            // Build the where clause
            if(last.length() > 0)
                whereClause += (" LAST_NAME = '" + last + "'");

            if(first.length() > 0) {
                if(whereClause.length() > 6)
                    whereClause += " AND";
                whereClause += (" FIRST_NAME = '" + first + "'");
            }

            if(ac.length() > 0) {
                if(whereClause.length() > 6)
                    whereClause += " AND";
                whereClause += (" AREA_CODE = '" + ac + "'");
            }

            if(pre.length() > 0) {
                if(whereClause.length() > 6)
                    whereClause += " AND";
                whereClause += (" PREFIX = '" + pre + "'");
            }

            if(sfx.length() > 0) {
                if(whereClause.length() > 6)
                    whereClause += " AND";
                whereClause += (" SUFFIX = '" + sfx + "'");
            }

            return "select LAST_NAME, FIRST_NAME, AREA_CODE, PREFIX, SUFFIX from Listin
gs" + whereClause;
        }
    }   // End GetListener inner class
}
```

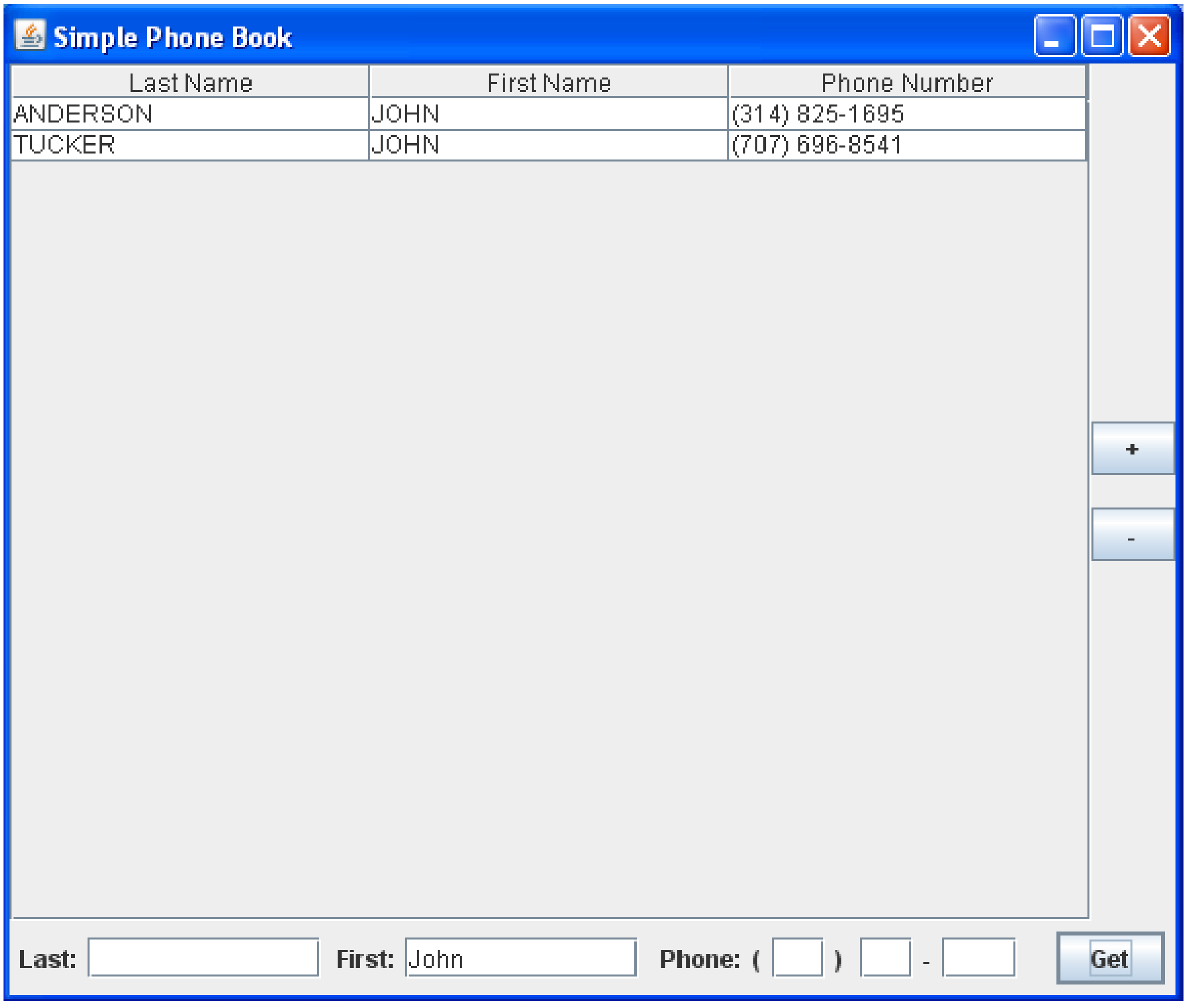
Save **PhoneBookFrame**.

Once **ListingsTableModel** is saved and its variable declared (uncommented) in **PhoneBookFrame**, your programs should be free of errors, at least for the moment.

Run it (from **SimplePhoneBook**). Type **John** in the First Name text field.



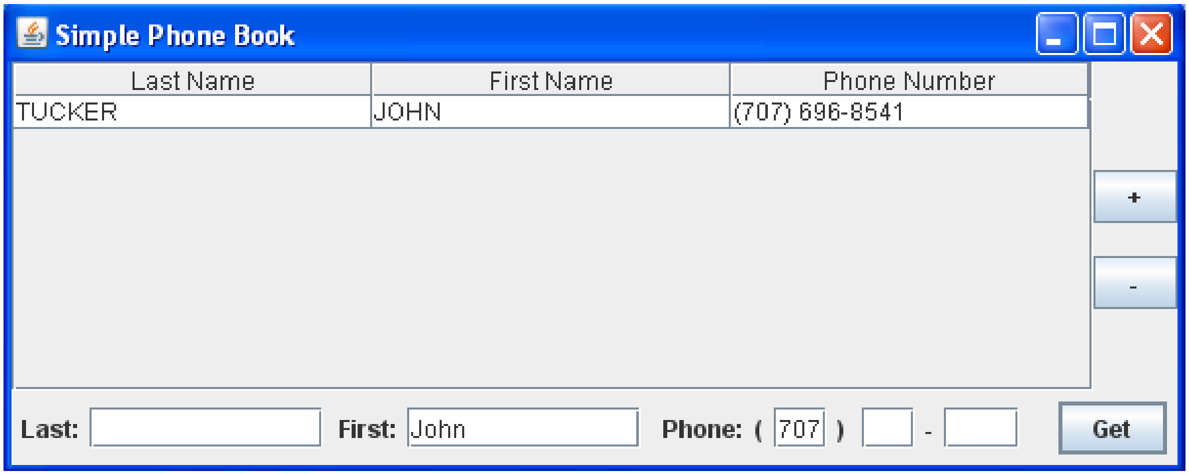
Click **Get** to display our database table entries that have a first name of **John**:



Last Name	First Name	Phone Number
ANDERSON	JOHN	(314) 825-1695
TUCKER	JOHN	(707) 696-8541

Last: First: John Phone: ( ) -

While you're there, notice the area codes in the phone numbers. The other text fields are *listening* as well. Keep **John** in the **First** Name text field, and type **707** in the **Phone ( )** area code field. Press **Enter** to display our database table entries that have a first name of **John** *and* a phone area code of **707**:

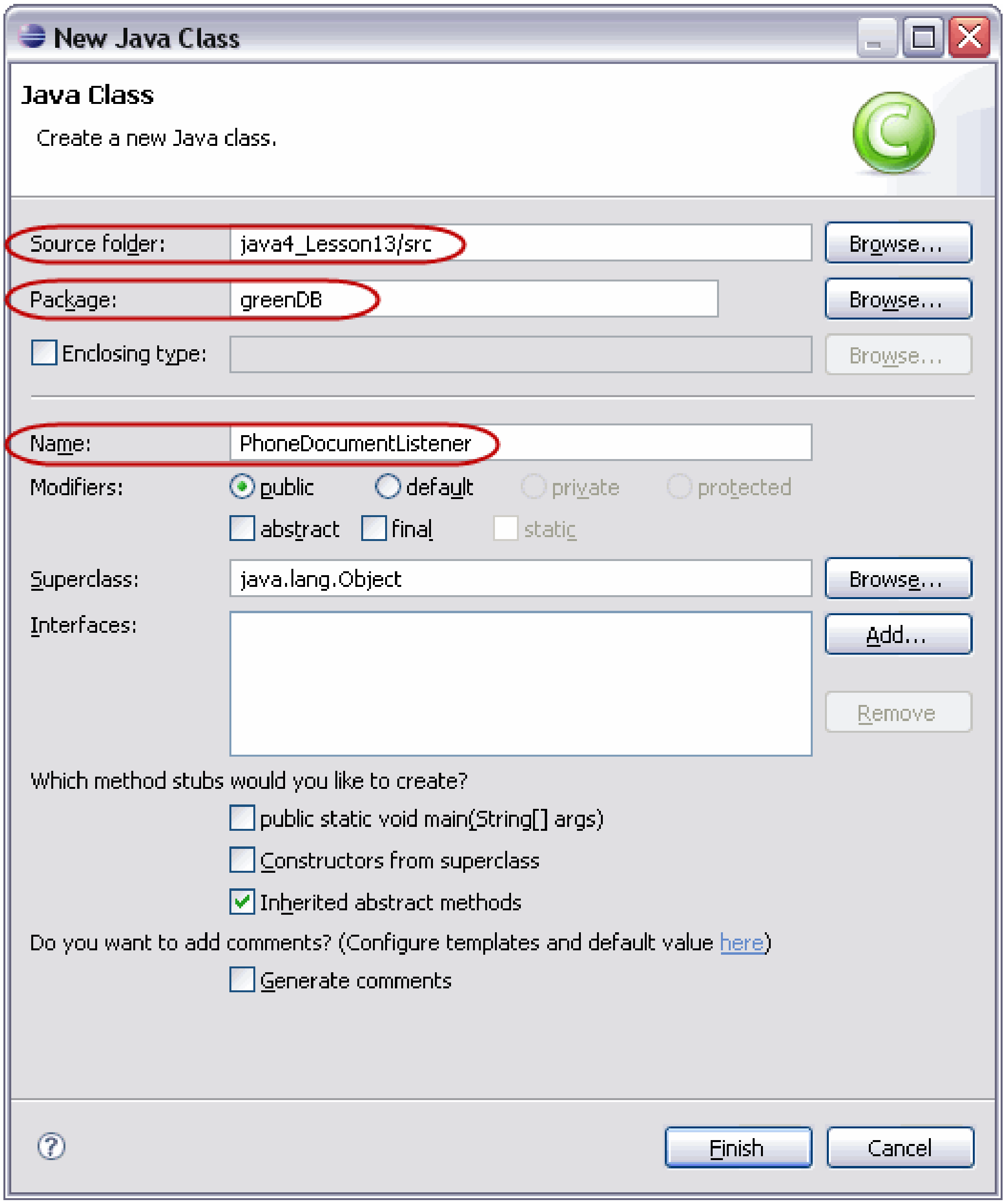


Last Name	First Name	Phone Number
TUCKER	JOHN	(707) 696-8541

Last: First: John Phone: (707) -

Isn't that cool? We're on a roll! Now let's incorporate the other listeners.

In the java4_Lesson13 project, add a **PhoneDocumentListener** class as shown:



Source folder: java4_Lesson13/src

Package: greenDB

□ Enclosing type:

Name: PhoneDocumentListener

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

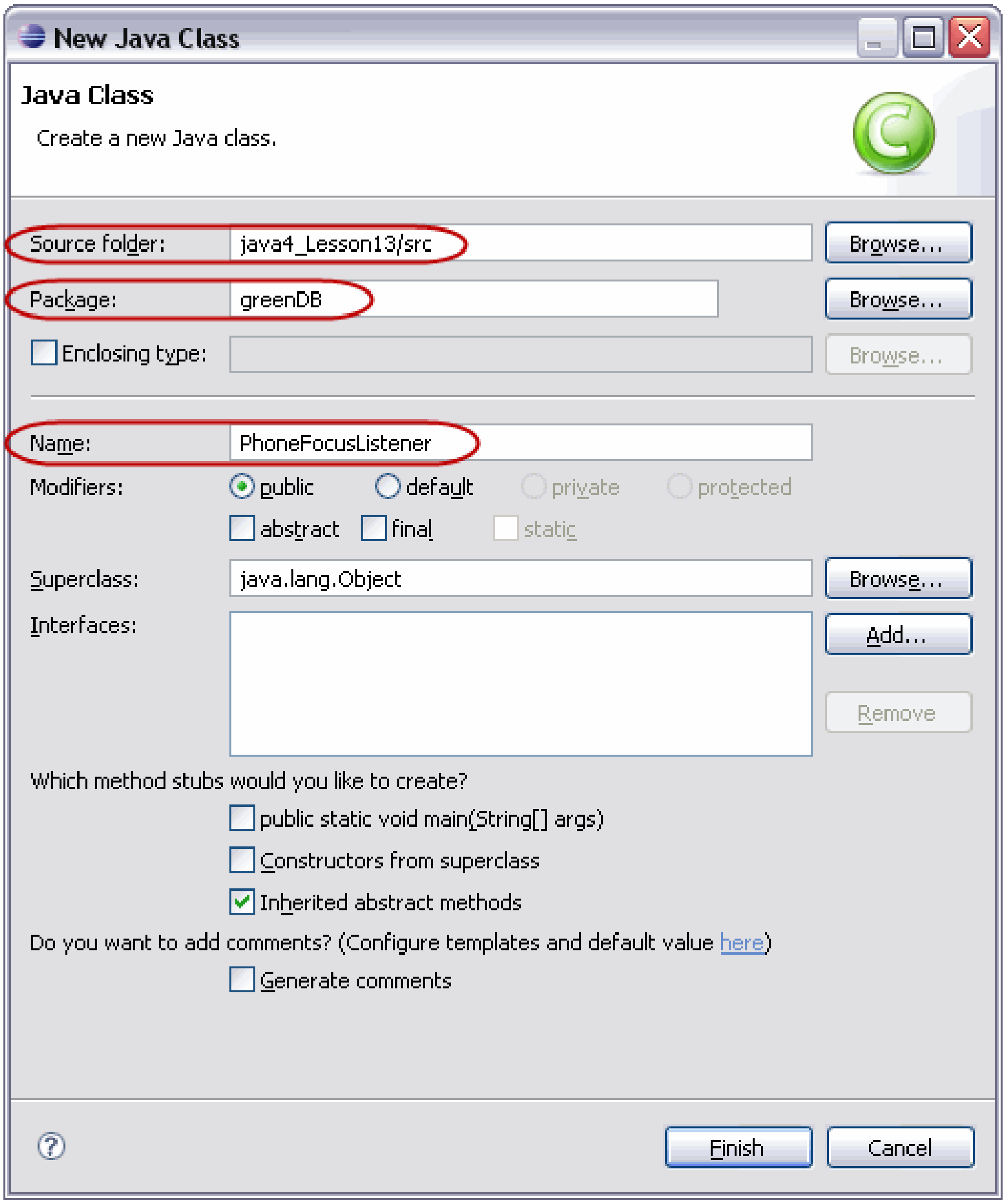
Type **PhoneDocumentListener** as shown in **blue**:

CODE TO TYPE: PhoneDocumentListener

```
package greenDB;

import javax.swing.JTextField;
import javax.swing.event.DocumentEvent;
import javax.swing.event.DocumentListener;

class PhoneDocumentListener implements DocumentListener {
    /** The phone number text field to which this listener applies */
    private JTextField txtField;
    /** The number of characters that will cause focus to be transferred */
    private int numsAllowed;

    public PhoneDocumentListener(JTextField tf, int numsAllowed) {
        txtField = tf;
        this.numsAllowed = numsAllowed;
    }

    public void insertUpdate(DocumentEvent dEvent) {
        if(dEvent.getDocument().getLength() == numsAllowed)
            txtField.transferFocus();
    }

    /** Empty implementation. Method necessary for implementation of DocumentListener *
/
    public void removeUpdate(DocumentEvent dEvent) {}
    /** Empty implementation. Method necessary for implementation of DocumentListener *
/
    public void changedUpdate(DocumentEvent dEvent) {}
} // End PhoneDocumentListener class
```

Save it.

In the java4_Lesson13 project, create **PhoneFocusListener** as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson13/src

Package: greenDB

□ Enclosing type:

Name: PhoneFocusListener

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **PhoneFocusListener** as shown in **blue**:

CODE TO TYPE: PhoneFocusListener

```
package greenDB;

import java.awt.event.FocusEvent;
import java.awt.event.FocusListener;
import javax.swing.JTextField;

class PhoneFocusListener implements FocusListener {

    /** an event generated as a result of focus being gained on this telephone number f
ield.  */
    public void focusGained(FocusEvent fEvent) {
        JTextField tf = (JTextField)fEvent.getSource();
        tf.setText("");
    }

    /** Not implemented */
    public void focusLost(FocusEvent fEvent){}

} // End PhoneFocusListener class
```

💾 Save it.

We need to add these listeners to our **PhoneBookFrame**. While we're there, we'll also add the listeners for the add (+) and remove (-) buttons to the JFrame.

Edit **PhoneBookFrame** as shown in **blue**:

CODE TO EDIT: PhoneBookFrame

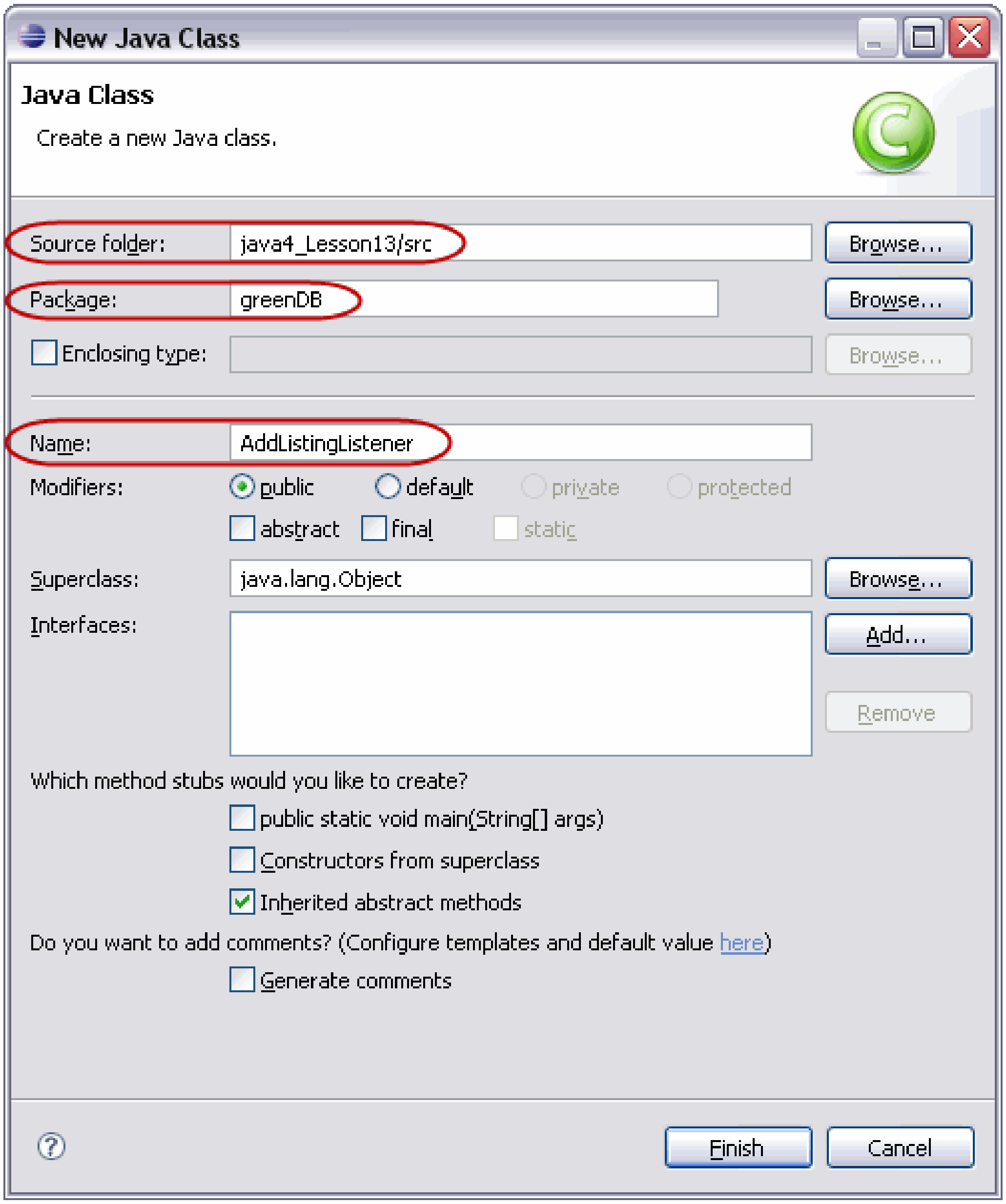
```
package greenDB;

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

class PhoneBookFrame extends JFrame {
    /** The initial user interface width, in pixels */
    private static final int WIDTH  = 577;
    /** The initial user interface height, in pixels */
    private static final int HEIGHT = 466;
    /** Provides methods for displaying a SQL result set in a JTable */
    private ListingsTableModel tblModel;
    /** Used to display the SQL result set in a cell format */
    private JTable table;
    /** A scrollable view for the SQL result set */
    private JScrollPane scrollPane;
    /** A text field for entering the phone listing's last name */
    private JTextField lNameField    = new JTextField(10);
    /** A text field for entering the phone listing's first name */
    private JTextField fNameField    = new JTextField(10);
    /** A text field for entering the phone listing's area code */
    private JTextField areaCodeField = new JTextField(2);
    /** A text field for entering the phone listing's prefix */
    private JTextField prefixField   = new JTextField(2);
    /** A text field for entering the phone listing's extension */
    private JTextField suffixField   = new JTextField(3);
    /** Database Operations */
    private DatabaseManager myDB;

    public PhoneBookFrame() {
        String [] info = PasswordDialog.login(this);  // static login so can call from
class
        // create and initialize the listings table
        myDB = new DatabaseManager(info[0], info[1]);
        // Should have access so make GUI

        JButton getButton = new JButton("Get");  // get the listing
        JButton add       = new JButton("+");    // add a listing
        JButton rem       = new JButton("-");    // remove a listing
        JLabel  space     = new JLabel(" ");
        // set the window size and title
        setTitle("Simple Phone Book");
        setSize(WIDTH, HEIGHT);
        // if user presses Enter, get button pressed
        getRootPane().setDefaultButton(getButton);
        // create the panel for looking up listing
        JPanel south = new JPanel();
        south.setLayout(new FlowLayout(FlowLayout.LEFT));

        south.add(new JLabel("Last:"));
        south.add(lNameField);
        south.add(new JLabel(" First:"));
        south.add(fNameField);
        south.add(new JLabel("  Phone:  ("));
        south.add(areaCodeField);
        south.add(new JLabel(") "));
        south.add(prefixField);
        south.add(new JLabel("-"));
        south.add(suffixField);
        south.add(new JLabel("   "));
        south.add(getButton);

        // create the panel for adding and deleting listings
        JPanel  east           = new JPanel();
        GridBagLayout gb       = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        east.setLayout(gb);
        add.setFont(new Font("SansSerif", Font.BOLD, 12));
        rem.setFont(new Font("SansSerif", Font.BOLD, 12));

        gbc.fill = GridBagConstraints.BOTH;
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gb.setConstraints(add, gbc);
        gb.setConstraints(space, gbc);
        gb.setConstraints(rem, gbc);
        east.setLayout(gb);
        east.add(add);
        east.add(space);
        east.add(rem);

        // add the panels
        Container contentPane = getContentPane();
        contentPane.add(south, BorderLayout.SOUTH);
        contentPane.add(east, BorderLayout.EAST);

        // Add listeners
        // When the application closes, drop the Listings table and close the connectio
n to MySQL
        addWindowListener(
                new WindowAdapter() {
                    public void windowClosing(WindowEvent wEvent) {
                     myDB.close(false);
                    }
                });

        areaCodeField.addFocusListener(new PhoneFocusListener());
        areaCodeField.getDocument().addDocumentListener(new PhoneDocumentListener(areaC
odeField, 3));

        prefixField.addFocusListener(new PhoneFocusListener());
        prefixField.getDocument().addDocumentListener(new PhoneDocumentListener(prefixF
ield, 3));

        suffixField.addFocusListener(new PhoneFocusListener());
        suffixField.getDocument().addDocumentListener(new PhoneDocumentListener(suffixF
ield, 4));

        add.addActionListener(new AddListingListener(this));  // add (+) listener--defi
ne in own class

        // remove (-) listener--delete the highlighted listing from the result set and
database
        rem.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent aEvent) {
                    try {
                        int selected = table.getSelectedRow();
                        ResultSet rset  = myDB.getResultSet();
                        if(selected != -1 && selected < tblModel.getRowCount()) {
                            rset.absolute(table.getSelectedRow() + 1);
                            rset.deleteRow();
                            table.repaint();
                            table.clearSelection();
                        }
                    } catch (SQLException e) {
                        e.printStackTrace();
                    }
                }
        });

        getButton.addActionListener(new GetListener());  // Add the listener for the ge
tButton (GetListener inner class defined below)
```

```
        // when the ui first displays do an empty lookup so the center panel doesn't lo
ok funny
        getButton.doClick();
        lNameField.requestFocus();    // set focus to last name field (most common look
up)
    }

    public DatabaseManager getDBManager(){
     return myDB;
    }
    /* inner class GetListener */
    class GetListener implements ActionListener {  // Gets the entries from the text fi
elds

        public void actionPerformed(ActionEvent aEvent) {
            // Get whatever the user entered, trim any white space and change to upper
case
            String last  = lNameField.getText().trim().toUpperCase();
            String first = fNameField.getText().trim().toUpperCase();
            String ac    = areaCodeField.getText().trim().toUpperCase();
            String pre   = prefixField.getText().trim().toUpperCase();
            String sfx   = suffixField.getText().trim().toUpperCase();

            // Replace any single quote chars w/ space char or SQL will think the ' is
the end of the string
            last  = last.replace('\'', ' ');
            first = first.replace('\'', ' ');
            ac    = ac.replace('\'', ' ');
            pre   = pre.replace('\'', ' ');
            sfx   = sfx.replace('\'', ' ');
            // Get rid of the last result displayed if there is one
            if(scrollPane != null)
                getContentPane().remove(scrollPane);
            // Only execute the query if one or more fields have data, else just displa
y an empty table
            if(last.length()  > 0 ||
             first.length() > 0 ||
             ac.length()    > 0 ||
             pre.length()   > 0 ||
             sfx.length()   > 0) {
                // build the query and execute it. Provide the results to the table mod
el
                myDB.doGetQuery(buildQuery(last, first, ac, pre ,sfx));
                ResultSet rset = myDB.getResultSet();
                tblModel = new ListingsTableModel(rset);
                table = new JTable(tblModel);
            } else {
                table = new JTable();
            }
            // Allows the user to only delete one record at a time
            table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
            // Add the table with the results to the contentPane and display it.
            scrollPane = new JScrollPane(table);
            getContentPane().add(scrollPane, BorderLayout.CENTER);
            pack();
            doLayout();
        }

        public String buildQuery(String last, String first, String ac, String pre, Stri
ng sfx) {
            String whereClause = " where";
            // Build the where clause
            if(last.length() > 0)
                whereClause += (" LAST_NAME = '" + last + "'");

            if(first.length() > 0) {
                if(whereClause.length() > 6)
                    whereClause += " AND";
                whereClause += (" FIRST_NAME = '" + first + "'");
            }

            if(ac.length() > 0) {
                if(whereClause.length() > 6)
                    whereClause += " AND";
                whereClause += (" AREA_CODE = '" + ac + "'");
            }

            if(pre.length() > 0) {
                if(whereClause.length() > 6)
                    whereClause += " AND";
                whereClause += (" PREFIX = '" + pre + "'");
            }

            if(sfx.length() > 0) {
                if(whereClause.length() > 6)
                    whereClause += " AND";
                whereClause += (" SUFFIX = '" + sfx + "'");
            }

            return "select LAST_NAME, FIRST_NAME, AREA_CODE, PREFIX, SUFFIX from Listin
gs" + whereClause;
        }
    } // End GetListener inner class
}
```

There's only one error; we might have expected it--we didn't define the **AddListingListener**.

Save it. Now, we'll allow the user to add entries.

In the java4_Lesson13 project, add the **AddListingListener** class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson13/src

Package: greenDB

□ Enclosing type:

Name: AddListingListener

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

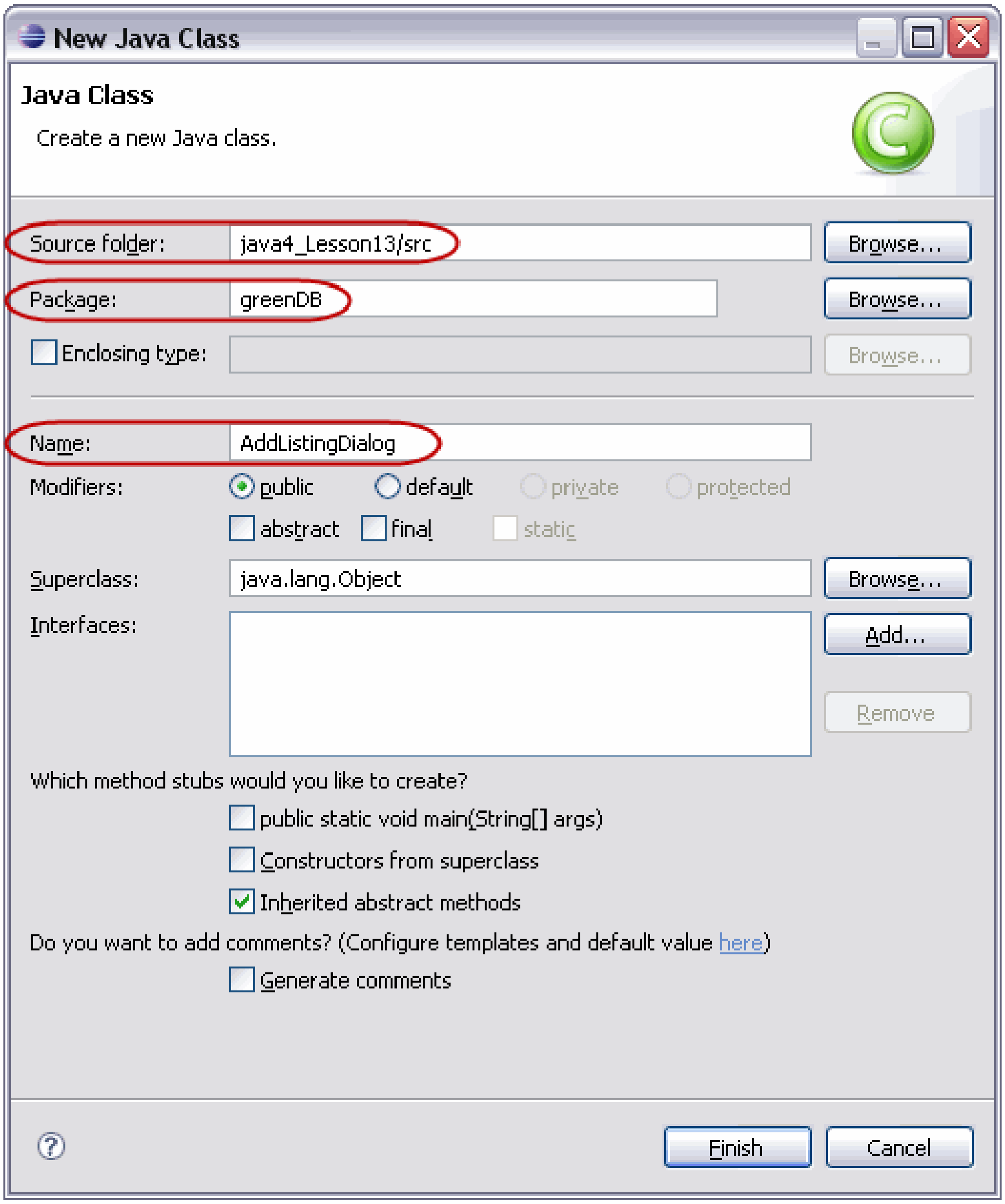
Type **AddListingListener** as shown in **blue**:

CODE TO TYPE: AddListingListener

```
package greenDB;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

class AddListingListener implements ActionListener {
    /** The SimplePhoneBook application frame */
    PhoneBookFrame pbf;

    public AddListingListener(PhoneBookFrame pbFrame) {
        pbf = pbFrame;
    }

    public void actionPerformed(ActionEvent aEvent) {
        AddListingDialog addDialog = new AddListingDialog(pbf);
        addDialog.setVisible(true);
    }
}  // End AddListingListener class
```

Save it. Can you tell what our next class will be?

In the java4_Lesson13 project, add an **AddListingDialog** class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java4_Lesson13/src

Package: greenDB

□ Enclosing type:

Name: AddListingDialog

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Type **AddListingDialog** as shown in **blue** below:

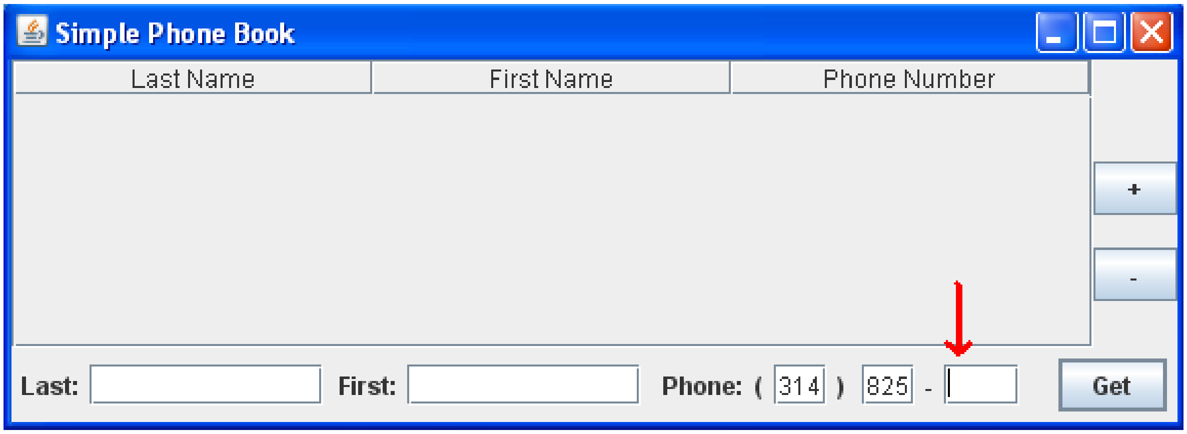
CODE TO TYPE: AddListingDialog

```
package greenDB;

import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.event.DocumentEvent;
import javax.swing.event.DocumentListener;

class AddListingDialog extends JDialog {
    /** A text field for entering the new phone listing's last name */
    private JTextField lNameField    = new JTextField(16);
    /** A text field for entering the new phone listing's first name */
    private JTextField fNameField    = new JTextField(16);
    /** A text field for entering the new phone listing's area code */
    private JTextField areaCodeField = new JTextField(2);
    /** A text field for entering the new phone listing's prefix */
    private JTextField prefixField   = new JTextField(2);
    /** A text field for entering the new phone listing's extension */
    private JTextField suffixField   = new JTextField(3);
    /** A button which, when clicked, will add the new listing to the Listings table */
    private JButton addButton;

    public AddListingDialog(final JFrame owner) {
        // set the dialog title and size
        super(owner, "Add Listing", true);
        setSize(280, 150);

        // Create the center panel which contains the fields for entering the new listi
ng
        JPanel center = new JPanel();
        center.setLayout(new GridLayout(3, 2));
        center.add(new JLabel(" Last Name:"));
        center.add(lNameField);
        center.add(new JLabel(" First Name:"));
        center.add(fNameField);

        // Here we create a panel for the phone number fields and add it to the center
panel.
        JPanel pnPanel = new JPanel();
        pnPanel.add(new JLabel("("));
        pnPanel.add(areaCodeField);
        pnPanel.add(new JLabel(") "));
        pnPanel.add(prefixField);
        pnPanel.add(new JLabel("-"));
        pnPanel.add(suffixField);
        center.add(new JLabel(" Phone Number:"));
        center.add(pnPanel);

        // Create the south panel, which contains the buttons
        JPanel south = new JPanel();
        addButton     = new JButton("Add");
        JButton cancelButton = new JButton("Cancel");
        addButton.setEnabled(false);
        south.add(addButton);
        south.add(cancelButton);

        // Add listeners to the fields and buttons
        lNameField.getDocument().addDocumentListener(new InputListener());
        fNameField.getDocument().addDocumentListener(new InputListener());
        areaCodeField.getDocument().addDocumentListener(new InputListener());
        prefixField.getDocument().addDocumentListener(new InputListener());
        suffixField.getDocument().addDocumentListener(new InputListener());

        areaCodeField.getDocument().addDocumentListener(new PhoneDocumentListener(areaC
odeField, 3));
        prefixField.getDocument().addDocumentListener(new PhoneDocumentListener(prefixF
ield, 3));
        suffixField.getDocument().addDocumentListener(new PhoneDocumentListener(suffixF
ield, 4));

        areaCodeField.addFocusListener(new PhoneFocusListener());
        prefixField.addFocusListener(new PhoneFocusListener());
        suffixField.addFocusListener(new PhoneFocusListener());

        // listeners to close the window
        addButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent aEvent) {
                    // ((PhoneBookFrame)owner).doInsertQuery(buildQuery());
                    DatabaseManager ownersDB = ((PhoneBookFrame)owner).getDBManager();
                    ownersDB.doInsertQuery(buildQuery());
                    dispose();
                }
        });

        cancelButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent aEvent) {
                    dispose();
                }
        });

        // Add the panels to the dialog window
        Container contentPane = getContentPane();
        contentPane.add(center, BorderLayout.CENTER);
        contentPane.add(south,  BorderLayout.SOUTH);
    }

    public String buildQuery() {
        // Get the data entered by the user, trim the white space and change to upper c
ase
        String query = "";
        String last  = lNameField.getText().trim().toUpperCase();
        String first = fNameField.getText().trim().toUpperCase();
        String ac    = areaCodeField.getText().trim().toUpperCase();
        String pre   = prefixField.getText().trim().toUpperCase();
        String sfx   = suffixField.getText().trim().toUpperCase();

        // Replace any single quote chars with a space char so the string will not get
truncated by SQL
        last  = last.replace('\'', ' ');
        first = first.replace('\'', ' ');
        ac    = ac.replace('\'', ' ');
        pre   = pre.replace('\'', ' ');
        sfx   = sfx.replace('\'', ' ');

        // build  and return the insert statement
        return new String("insert into Listings values ('" + last + "', '" +
         first + "', '" +
         ac + "', '" +
         pre + "', '" +
         sfx + "')");
    }

    /* inner class InputListener */
    class InputListener implements DocumentListener {
        public void insertUpdate(DocumentEvent dEvent) {
            // If first name and last name have data and phone number is complete
            // enable the add button, give it focus and make it clickable if
            // user presses Enter.
            if(lNameField.getDocument().getLength()     > 0 &&
             fNameField.getDocument().getLength()     > 0 &&
             areaCodeField.getDocument().getLength() == 3 &&
             prefixField.getDocument().getLength()    == 3 &&
             suffixField.getDocument().getLength()    == 4) {

                addButton.setEnabled(true);
                if(dEvent.getDocument() == suffixField.getDocument()) {
                    addButton.requestFocus();
                    getRootPane().setDefaultButton(addButton);
                }
            }
        }

        public void removeUpdate(DocumentEvent dEvent) {
            // If last name or first name don't have data or phone number
            // is  not complete, disable the Add button.
            if(lNameField.getDocument().getLength()   == 0 ||
             fNameField.getDocument().getLength()   == 0 ||
             areaCodeField.getDocument().getLength() < 3 ||
             prefixField.getDocument().getLength()    < 3 ||
             suffixField.getDocument().getLength()    < 4 )

                addButton.setEnabled(false);
        }

        /** Empty implementation. Method necessary for implementation of DocumentListen
er */
        public void changedUpdate(DocumentEvent dEvent) {}

    } // End InputListener inner class

} // End AddListingDialog class
```

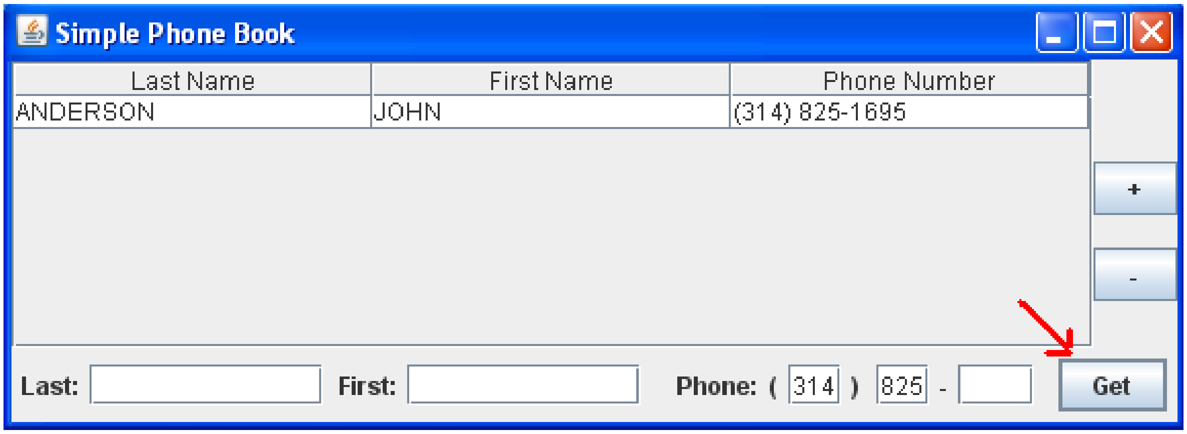
Save it. We should be all set now!

Run it from SimplePhoneBook.

In the Phone field, type **314825** without moving the mouse or using the Tab key--see how the "focus" moves to the other Phone field area? That's really convenient for the user.



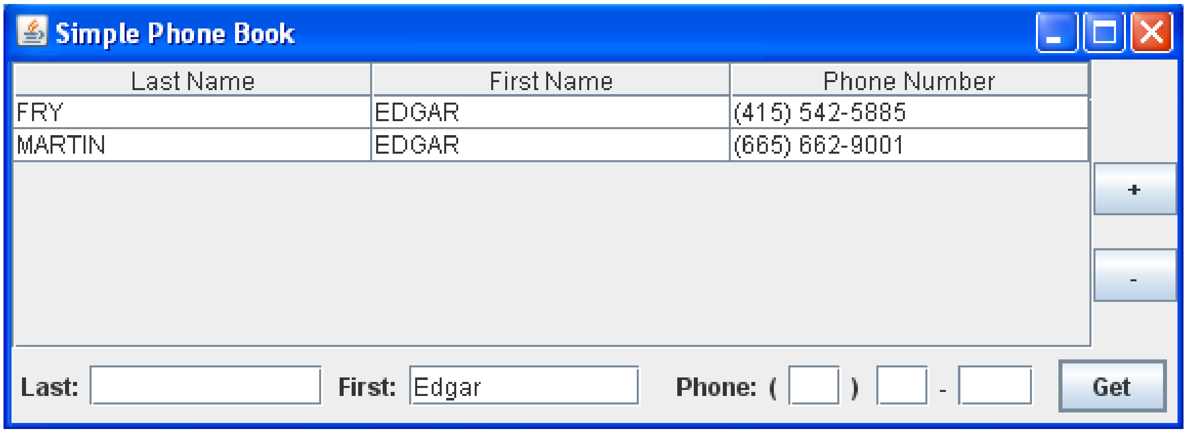
Now, click **Get**. The entry that has those first 6 digits in the phone number is retrieved.



Last Name	First Name	Phone Number
ANDERSON	JOHN	(314) 825-1695

Last: First: Phone: (314) 825 -

Clear the Phone number fields, and in the First name field, type **Edgar** and press **Enter**.

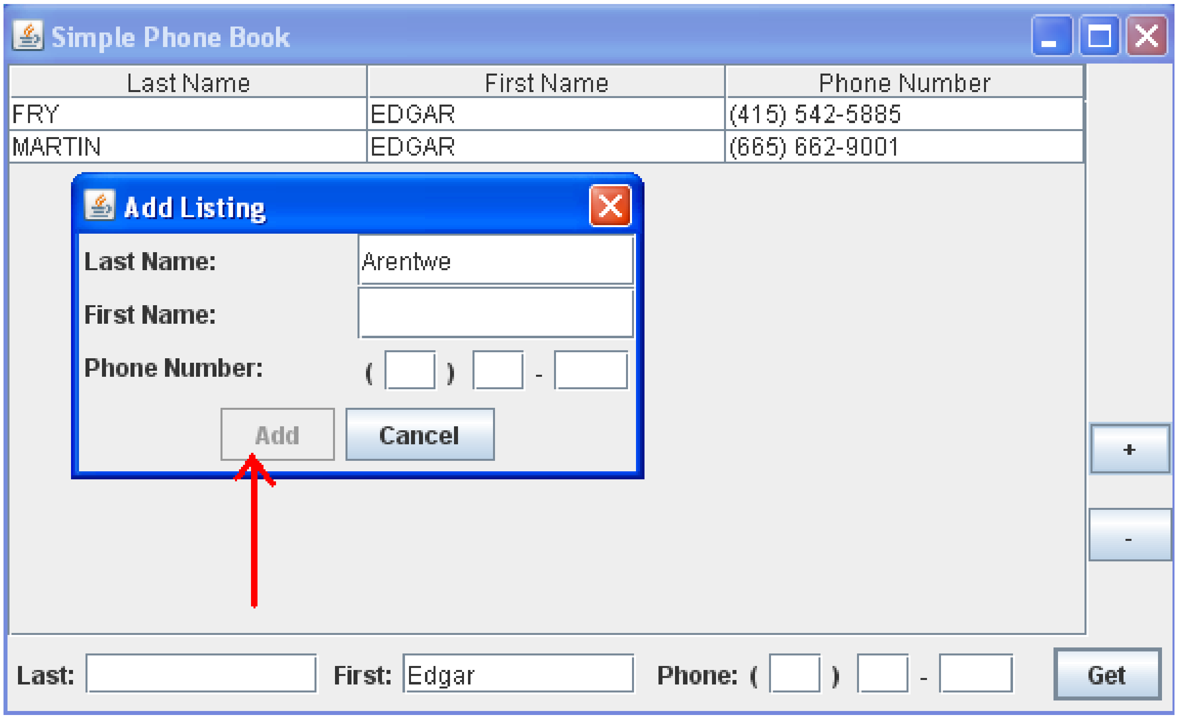


Last Name	First Name	Phone Number
FRY	EDGAR	(415) 542-5885
MARTIN	EDGAR	(665) 662-9001

Last: First: Edgar Phone: ( ) -

So there ARE names in our table other than John! Now let's add some entries of our own.

Click the AddButton **(+)**. Type a name in the **Last Name** field. The **Add** button is not an option--can you see where this was set in your code?



Last Name	First Name	Phone Number
FRY	EDGAR	(415) 542-5885
MARTIN	EDGAR	(665) 662-9001

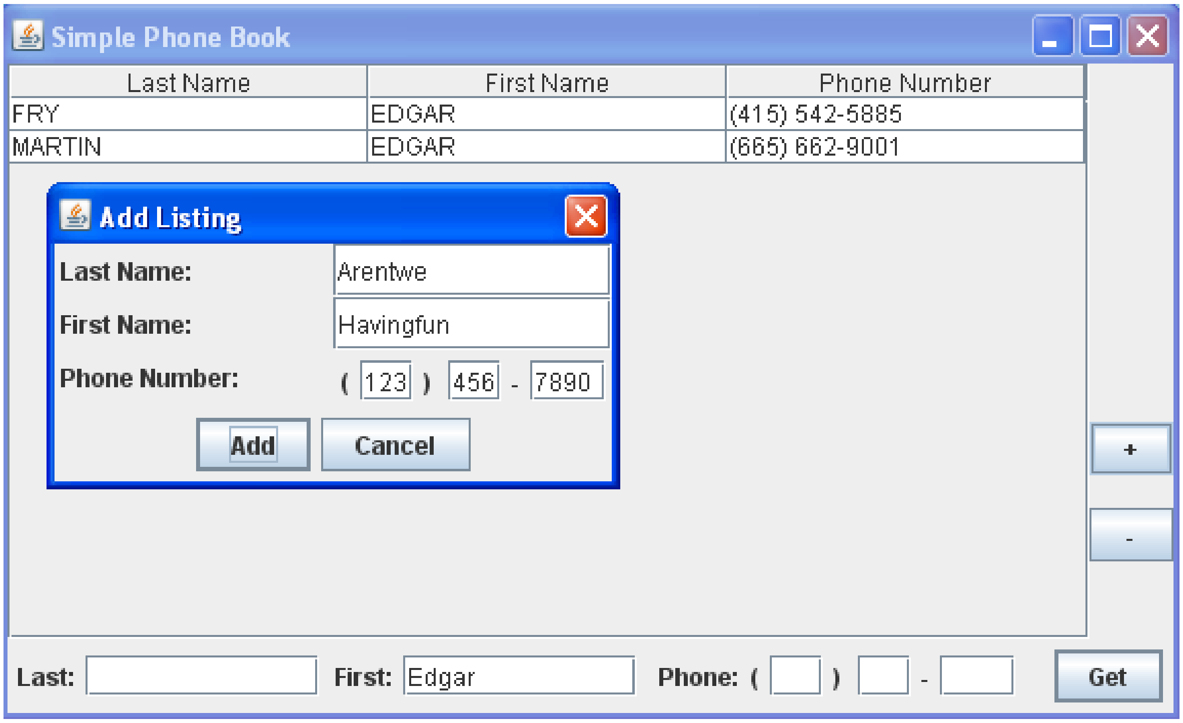
Last Name: Arentwe

First Name:

Phone Number: ( ) -

Last: First: Edgar Phone: ( ) -

Add the remaining information into the Add Listing Dialog Box. Note that when *all* of the fields have values, **Add** is enabled. Click **Add** now.



Simple Phone Book

Last Name	First Name	Phone Number
FRY	EDGAR	(415) 542-5885
MARTIN	EDGAR	(665) 662-9001

Add Listing

Last Name: Arentwe

First Name: Havingfun

Phone Number: (123) 456 - 7890

Last: First: Edgar Phone: ( ) -

Note One SQL statement and/or query does not implicitly invoke others. It is up to the programmer to tell the application what to do.

To see your new entry, query for it, using the appropriate text field(s).

Play with the GUI. You can cut and paste from one text field to another and there's cool ways to of tabs, etc. And, with each capability you notice, look at the code and see how it was done--or, if it was inherited, how *someone* (Java? Swing?) did all the work for you!

Additional Resources

We've created a good application and learned some of the basics of JDBC. But there's still a lot more out there. Here are a few more useful resources:

Oracle's JDBC page has links to:

We've illustrated many facets of JDBC, but lots of additional techniques are available. Among other tasks, you may want to:

- Oracle's JDBC Technology Guide links to Getting Started with the JDBC API.
- JDBC Introduction.
- Three nice tutorials:
  - Basic Tutorial: includes JDBC Basics.
  - Advanced Tutorial: shows deleting and adding rows and much more.
  - RowSet Tutorial: useful for enterprise applications.

But wait--there's more:

- Online Tutorial, from the Java Developer Connection.
- Another Online Tutorial—this one illustrates connecting to a database on a Windows machine using the JDBC/ODBC bridge.

And of course, there are books:

Database Programming with JDBC & Java

The next lesson has additional resource links for SQL and documentation capabilities, because we want *you* to write great Java code!

Documentation: Javadoc, Doc Comments, and Annotation

Documenting Your Work

After all of your hard programmming work, your long hours of planning and toil, most people will generally see just the application or running applet. However, when someone considers purchasing your appplications, or offering you a job writing code, they'll probably want you to *show your work*. In these cases especially, you want your code and documentation to be clean and readable.

Programmers estimate that about 70% of all programming effort goes toward maintenance. Given this percentage, along with the likelihood that someday *your* code wil be maintained by others, you want to be sure to document your code adequately. Fortunately, Java makes it easy.

Javadoc and API Pages

Oracle provides good-looking API pages for Java. We can create similar pages for our own code. Java provides a tool called *Javadoc*, which allows us to create API pages that have the professional and clean look.

But before we create those pages, let's add documentation to the code that we created in earlier lessons. The edits we make first will not affect the running of our application because they'll consist of comments, not new code. The comments will contain the format and documentation tags that will enable us to make beautiful--and *accessible*--documentation.

Note Some of our code already contains Javadoc comments we added while creating it.

DatabaseManager

In the java4_Lesson13 project, greenDB package, edit **DatabaseManager** as shown in **blue**:

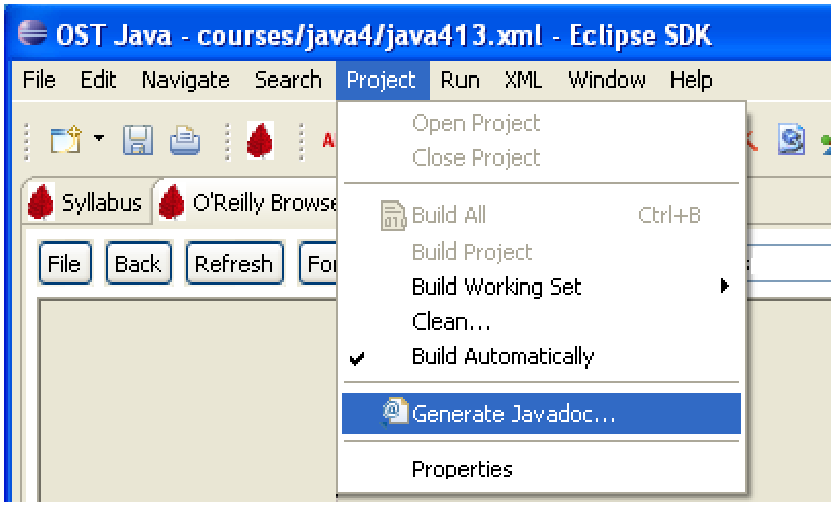
CODE TO EDIT: DatabaseManager

```java
package greenDB;

import java.sql.*;

public class DatabaseManager {
    /** A connection to a database */
    private Connection conn;

    /** An executable SQL statement */
    private Statement stmt;

    /** The result of an executed SQL statement */
    private ResultSet rset;


    /* DatabaseManager Constructor */
    /**
     * This constructor connects to the MySQL database at jdbc:mysql://sql.useractive.com:3306.
     * It creates instances of Statement and ResultSet that will be used by the other methods
     * in the class. It also checks to see if the Listings table already exists. If it does, it
     * simply returns to the caller. Otherwise, it instantiates the method to create a table
     * called Listings, and then populates the table.
     * <pre>
     * PRE:  MySQL server is available and account for user has been established.
     *       The MySQL driver is installed on the client workstation and its location
     *       has been defined in CLASSPATH (or for Eclipse, in its Referenced Libraries).
     *       Username and password are not null.
     * POST: A connection is made and the Listings table is either created and initialized on
     *       jdbc:mysql://sql.useractive.com:3306, or the already existing Listings table is
     *       identified.
     * </pre>
     */
    public DatabaseManager (String username, String password ) {  // the constructor for the database manager

        // Connect to database and execute the SQL commands for creating and initializing the Listings table.
        try {
            Class.forName ("com.mysql.jdbc.Driver");  // Load the MySQL JDBC driver
        }
        catch (ClassNotFoundException e) {
            System.out.println("Failed to load JDBC/ODBC driver.");
            e.printStackTrace();
            return;
        }

        try {        // Connect to the database--
                     // give the whole URL as a parameter rather than using a variable
            conn = DriverManager.getConnection ("jdbc:mysql://sql.useractive.com:3306/" + username, username, password);
            stmt = conn.createStatement (ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);   // Create a Statement
            DatabaseMetaData aboutDB = conn.getMetaData ();  // Execute the creation and initialization of table query
            String [] tableType = {"TABLE"};
            ResultSet rs = aboutDB.getTables(null, null, "Listings",  tableType)
;
            if (!inspectForTable (rs, "Listings")) {         // First find out i
f the table is already there
                                                             // there is no tabl
e, make it from the initialization listing
                String [] SQL = initListingsTable();         // code for this me
thod is below
                for (int i=0; i < SQL.length; i++)
                {
                    stmt.execute(SQL[i]);
                }
            }
        }catch (SQLException e) {
            e.printStackTrace();
        }
    }

    /* initListingsTable */
    /**
     * Creates the Listings table and initializes it with some records. This met
hod connects
     * to the MySQL database at jdbc:mysql://sql.useractive.com:3306. It then cr
eates a table
     * called Listings and initializes the table with some arbitrary records.
     * <pre>
     * PRE: True
     * POST: SQL String is created for the initial population of a table named L
istings.
     * </pre>
     */
    private String [] initListingsTable() {
        // Executable SQL commands for creating Listings table and inserting ini
tial names and phone numbers.
        String[]  SQL = {
            "create table Listings ("+
            "LAST_NAME  varchar (16)," +
            "FIRST_NAME varchar (16)," +
            "AREA_CODE  varchar(3)," +
            "PREFIX     varchar(3)," +
            "SUFFIX     varchar(4))",
            "insert into Listings values ('ANDERSON', 'JOHN',      '314', '825',
 '1695')",
            "insert into Listings values ('CABLES',   'WALLY',     '212', '434',
 '9685')",
            "insert into Listings values ('FRY',      'EDGAR',     '415', '542',
 '5885')",
            "insert into Listings values ('MARTIN',   'EDGAR',     '665', '662',
 '9001')",
            "insert into Listings values ('TUCKER',   'JOHN',      '707', '696',
 '8541')",
        };
        return SQL;
    }

    /* inspectForTable */
    /**
     * Determines if a table exists in the db.
     * <pre>
     * PRE:  Connection to database has been established. rs is not null.
     * POST: Table has not been changed, but its presence is verified (or not).
     * </pre>
     * @param rs ResultSet from DatabaseMetaData query about existing Tables
     * @param tableName String identifying the table in question
     */
    private boolean inspectForTable (ResultSet rs, String tableName)  throws SQL
Exception {  // exception will be caught when method is used
        int i;
        ResultSetMetaData rsmd = rs.getMetaData ();
                    // Get the ResultSetMetaData.  This will be used for the col
umn headings
        int numCols = rsmd.getColumnCount ();
                    // Get the number of columns in the result set

        boolean more = rs.next ();
        while (more) {
                    // Get each row, fetching until end of the result set
            for (i=1; i<=numCols; i++) {
                if (rsmd.getColumnLabel(i) == "TABLE_NAME")
                      // Loop through each row, getting the column data looking
for Tables
                    if  (rs.getString(i).equals(tableName))
                       // If the column is the TABLE_NAME, is the the one we are
 looking for?
                    {
                        System.out.println("Found one that equals " + rs.getStri
ng(i));

                        return true;
                    }
            }
            System.out.println("");
            more = rs.next ();
                      // Fetch the next result set row
        }
        return false;
    }

    /* doGetQuery */
    /**
     * Executes the select query specified.
     * <pre>
     * PRE:  Connection to database has been established. Query is assigned and
is a simple
     *       select statement against the Listings table.
     * POST: The query is executed.
     * </pre>
     * @param query a simple select query against the Listings table
     */
    public void doGetQuery(String query) {
                    // rather than the "getEntries" of the previous example
        try {
            rset = stmt.executeQuery(query);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    /* doInsertQuery */
    /**
     * Executes an insert statement, specified by query.
     * <pre>
     * PRE:  Connection to database has been established. Query is assigned and
is a simple
     *       insert statement into the Listings table.
     * POST: The query is executed.
     * </pre>
     * @param query a simple insert query into the Listings table
     */
    public void doInsertQuery(String query) {   // rather than the hard-coded "a
ddEntry" of the previous example
        try {
            stmt.executeUpdate(query);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    /* getResultSet */
    /**
     * Returns the current value of the ResultSet instance
     * <pre>
     * PRE:  True
     * POST: ResultSet instance value is returned, its value remains the same as
 upon entry.
     * </pre>
     */
    public ResultSet getResultSet() {  // a new method that will let the GUI get
 the resultSet to manipulate it
        return rset;
    }

    /* close */
    /**
     * Closes opened Statements and the Connection.
     * <pre>
     * PRE:  Connection to database has been established. Statement has been cre
ated. Listings is a table in the db
     * POST: If remove is true, table Listings is dropped, otherwise it is prese
rved.  Open Connection and Statement are closed
     * </pre>
     * @param remove boolean to specify if the table Listings should be dropped
or not.
     */
    public void close(boolean remove){
                    // closes all open connections

        try {
            if (remove)
                stmt.execute ("drop table Listings;");

            stmt.close();
            conn.close();
        }
        catch ( SQLException e ) {
            System.out.println ("\n*** SQLException caught ***\n");
            e.printStackTrace();
        }
    }
}
```

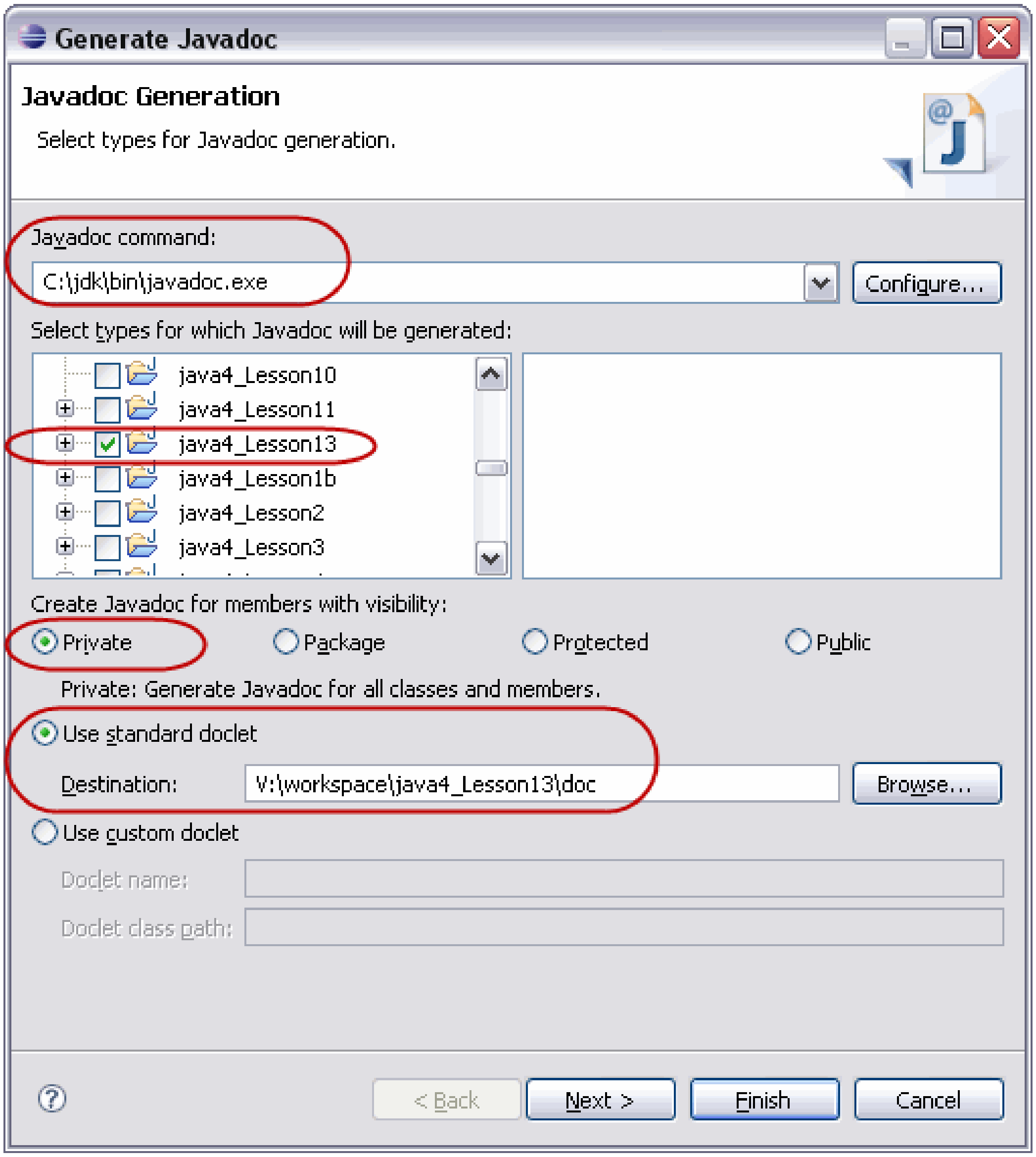
Save it.

Creating Javadocs

In the top Eclipse Menu Bar, select **Project | Generate Javadoc...**:



In the Generate Javadoc window, use the **Configure...** button to get to the **C:\jdk\bin\** directory. Choose **javadoc.exe** to retrieve the javadoc executable code. The default should remain **Private**; this allows us to view all members (private, package, protected and public). Keep the Destination as it is, so that the resulting Javadoc will be located with the Project. Click **Finish**:



Javadoc Generation

Select types for Javadoc generation.

Javadoc command:
C:\jdk\bin\javadoc.exe

Select types for which Javadoc will be generated:

- □ java4_Lesson10
- □ java4_Lesson11
- ☑ java4_Lesson13
- □ java4_Lesson1b
- □ java4_Lesson2
- □ java4_Lesson3

Create Javadoc for members with visibility:

● Private ○ Package ○ Protected ○ Public

Private: Generate Javadoc for all classes and members.

● Use standard doclet

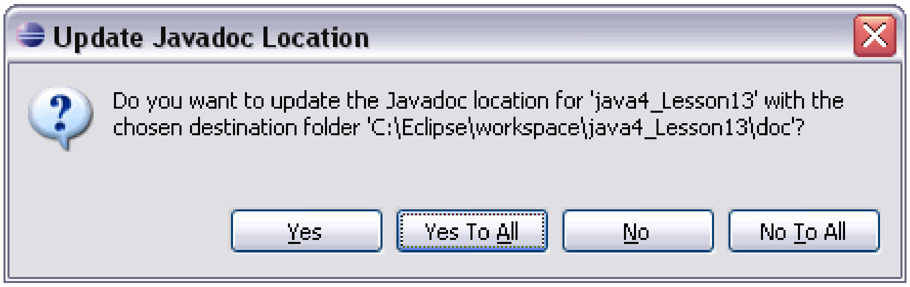
Destination: V:\workspace\java4_Lesson13\doc

○ Use custom doclet

Doclet name:

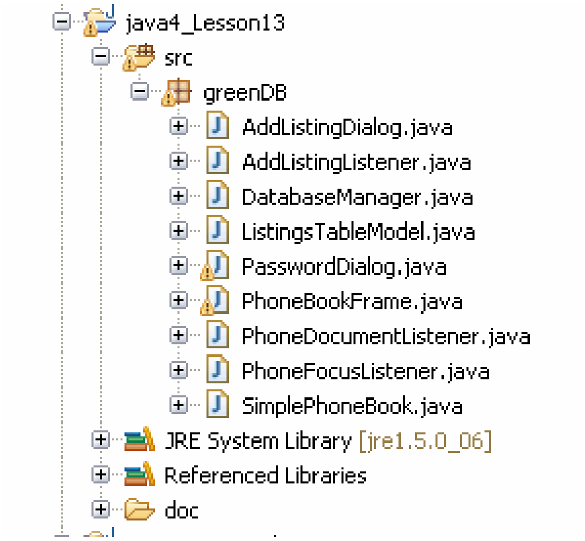
Doclet class path:

In the Update Javadoc Location dialog box that appears, you're given the option to choose Javadoc location as the destination folder. Click **Yes**:

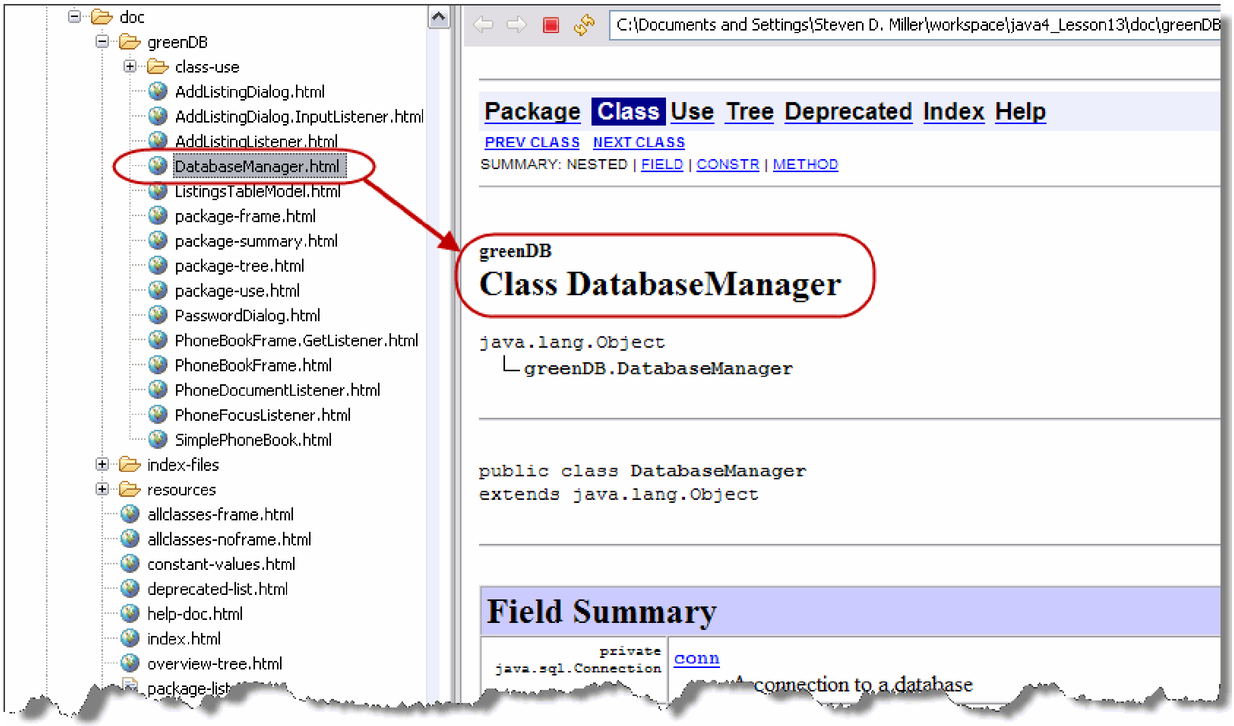


Do you want to update the Javadoc location for 'java4_Lesson13' with the chosen destination folder 'C:\Eclipse\workspace\java4_Lesson13\doc'?

The console displays the actions taking place and then displays a new **doc** folder in the java4_Lesson13 Project:



Open the **doc** folder and its **greenDB** subfolder, then double-click on **DatabaseManager.html**. Scroll through this API page of the application and observe the way the comments were printed. There is more documentation within the **Constructor Summary** and **Method Summary**, because we used more comments in those areas than in others:



greenDB

Class DatabaseManager

```
java.lang.Object
  └greenDB.DatabaseManager
```

```
public class DatabaseManager
extends java.lang.Object
```

Field Summary

private java.sql.Connection	conn A connection to a database

When the Javadoc **executable** runs, it looks for the special comments that begin with /** and end with */. When they're written for class members (instance and class methods and variables), these comments should be entered just *before* the field's definition or declaration in your code.

Documenting and Tagging the Application

Java provides some tags by default, but you can create your own as well. In this section, we go over each class with additional comments. The code is the same; the only differences are the added comments.

Note "@" is used to denote tags that will be created within the API pages.

Some methods include the words **PRE** and **POST**. These stand for preconditions and postconditions. Such specifications for methods are useful for the design, implementation, and use of code:

A **PRE**condition is an assertion that must be true in order for the operation to run properly. A preconditions of *true* (or empty) means that there are no stipulations on running the particular method.

- The implementer assumes it to be true and codes accordingly.
- The user must be sure in his code that the **PRE**condition is met before invoking the method.

A **POST** condition is an assertion that prevails upon completion of the operation. Postconditions describe the results of the actions performed by the operation. They must be accurately and precisely stated.

- The implementer must make the **POST** condition is true and codes accordingly.
- The user assumes it will be exactly and completely true after the code runs.

We will edit and save each class in our database application, and then run the Javadoc executable on the entire package when the classes all have Javadoc comments.

SimplePhoneBook

In java4_Lesson13, edit **SimplePhoneBook** as shown in **blue** below:

CODE TO EDIT: SimplePhoneBook

```
package greenDB;

import javax.swing.JFrame;

/* class SimplePhoneBook */
/**
 * This is the entry point of the SimplePhoneBook application. SimplePhoneBook i
s a Java application
 * that uses JDBC to store, retrieve, add, and delete phone number listings in a
 MySQL
 * database. The SimplePhoneBook class instantiates the application window frame
 and displays it
 * on screen.
 *
 * @author David Green
 * @version 1.0
 */
public class SimplePhoneBook {
    /* main */
    /**
     * The entry point for the SimplePhoneBook application. The main method inst
antiates the
     * application's frame window and displays it.
     * <pre>
     * PRE:
     * POST: SimplePhoneBook started.
     * </pre>
     * @param args    not used.
     */
    public static void main(String args[]) {
    // Instantiate the phone book frame window and display it.

        PhoneBookFrame pbFrame = new PhoneBookFrame();
        pbFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pbFrame.setVisible(true);
    }
}// End SimplePhoneBook class
```

Save it.

PhoneBookFrame

In java4_Lesson13, edit **PhoneBookFrame** as shown in **blue** below:

CODE TO EDIT: PhoneBookFrame

```
package greenDB;

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

//* class PhoneBookFrame */
/*
 * This class represents the SimplePhoneBook user interface. PhoneBookFrame incl
udes the application
 * window as well as the components for retrieving, adding, displaying, and dele
ting phone number listings
 * for the user.
 *
 * @author David Green
 * @version 1.0
 */
class PhoneBookFrame extends JFrame {
    /** The initial user interface width, in pixels */
    private static final int WIDTH  = 577;
    /** The initial user interface height, in pixels */
    private static final int HEIGHT = 466;
    /** Provides methods for displaying a SQL result set in a JTable */
    private ListingsTableModel tblModel;
    /** Used to display the SQL result set in a cell format */
    private JTable table;
    /** A scrollable view for the SQL result set */
    private JScrollPane scrollPane;
    /** A text field for entering the phone listing's last name */
    private JTextField lNameField    = new JTextField(10);
    /** A text field for entering the phone listing's first name */
    private JTextField fNameField    = new JTextField(10);
    /** A text field for entering the phone listing's area code */
    private JTextField areaCodeField = new JTextField(2);
    /** A text field for entering the phone listing's prefix */
    private JTextField prefixField   = new JTextField(2);
    /** A text field for entering the phone listing's extension */
    private JTextField suffixField   = new JTextField(3);
    /** Database Operations */
    private DatabaseManager myDB;

    /* PhoneBookFrame */
    /**
     * The PhoneBookFrame constructor.
     */
    public PhoneBookFrame() {
        String [] info = PasswordDialog.login(this);  // static login so can cal
l from class
        // create and initialize the listings table
        myDB = new DatabaseManager(info[0], info[1]);
        // Should have access so make GUI

        JButton getButton = new JButton("Get");     // get the listing
        JButton add       = new JButton("+");       // add a listing
        JButton rem       = new JButton("-");       // remove a listing
        JLabel  space     = new JLabel(" ");

        // set the window size and title
        setTitle("Simple Phone Book");
        setSize(WIDTH, HEIGHT);

        // if user presses Enter, get button pressed
        getRootPane().setDefaultButton(getButton);

        // create the panel for looking up listing
        JPanel south = new JPanel();
        south.setLayout(new FlowLayout(FlowLayout.LEFT));

        south.add(new JLabel("Last:"));
        south.add(lNameField);
        south.add(new JLabel(" First:"));
        south.add(fNameField);
        south.add(new JLabel("  Phone:  ("));
        south.add(areaCodeField);
        south.add(new JLabel(") "));
        south.add(prefixField);
        south.add(new JLabel("-"));
        south.add(suffixField);
        south.add(new JLabel("   "));
        south.add(getButton);

        // create the panel for adding and deleting listings
        JPanel  east           = new JPanel();
        GridBagLayout gb       = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        east.setLayout(gb);
        add.setFont(new Font("SansSerif", Font.BOLD, 12));
        rem.setFont(new Font("SansSerif", Font.BOLD, 12));

        gbc.fill = GridBagConstraints.BOTH;
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gb.setConstraints(add, gbc);
        gb.setConstraints(space, gbc);
        gb.setConstraints(rem, gbc);
        east.setLayout(gb);
        east.add(add);
        east.add(space);
        east.add(rem);

        // add the panels
        Container contentPane = getContentPane();
        contentPane.add(south, BorderLayout.SOUTH);
        contentPane.add(east, BorderLayout.EAST);

        // Add listeners
        getButton.addActionListener(new GetListener());

        areaCodeField.addFocusListener(new PhoneFocusListener());
        areaCodeField.getDocument().addDocumentListener(new PhoneDocumentListene
r(areaCodeField, 3));

        prefixField.addFocusListener(new PhoneFocusListener());
        prefixField.getDocument().addDocumentListener(new PhoneDocumentListener(
prefixField, 3));

        suffixField.addFocusListener(new PhoneFocusListener());
        suffixField.getDocument().addDocumentListener(new PhoneDocumentListener(
suffixField, 4));

        add.addActionListener(new AddListingListener(this));

        // delete the highlighted listing from the result set and database
        rem.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent aEvent) {
                    try {
                        int selected = table.getSelectedRow();
                        ResultSet rset  = myDB.getResultSet();
                        if(selected != -1 && selected < tblModel.getRowCount())
{
                            rset.absolute(table.getSelectedRow() + 1);
                            rset.deleteRow();
                            table.repaint();
                            table.clearSelection();
                        }
                    } catch (SQLException e) {
                    e.printStackTrace();
                    }
                }
        });

        // When the application closes, drop the Listings table and close the co
nnection to MySQL
        addWindowListener(
            new WindowAdapter() {
                public void windowClosing(WindowEvent wEvent) {
                 myDB.close(false);
                }
        });

        // when the ui first displays do an empty lookup so the center panel doe
sn't look funny
        getButton.doClick();
        lNameField.requestFocus();    // set focus to last name field (most comm
on lookup)
    }

    public DatabaseManager getDBManager(){
    return myDB;
    }

    /* inner class GetListener */
    /**
     * An inner class for handling the event when the user clicks the "Get" butt
on.
     *
     * @author David Green
     * @version 1.0
     */
    class GetListener implements ActionListener {

    /* actionPerformed */
    /**
     * This method builds a select Query and executes it against the Listings ta
ble to retrieve
     * records. This method creates a select string based on what the user has e
ntered in the
     * fields for Last Name, First Name, Area Code, Prefix, and Extension. The u
ser may look up a
     * record in the Listings table based on any combination of data entered in
the text fields.
     * The actionPerformed method builds the query string based on the user's in
put, executes the
     * query, and displays it in a scrollable cell format. All data entered in t
he text fields
     * is converted to upper case and any single quote character is replaced wit
h a space
     * character before the query is executed.
     * <pre>
     * PRE:  A connection to the database has been established. All text fields
can be empty.
     * POST: A select string is created based on what was entered, the query is
executed and the
     *      results are displayed.
     * </pre>
     * @param aEvent an event generated as a result of the "Get" button being cl
icked
     */
        public void actionPerformed(ActionEvent aEvent) {
            // Get whatever the user entered, trim any white space and change to
 upper case
```

```
            String last  = lNameField.getText().trim().toUpperCase();
            String first = fNameField.getText().trim().toUpperCase();
            String ac    = areaCodeField.getText().trim().toUpperCase();
            String pre   = prefixField.getText().trim().toUpperCase();
            String sfx   = suffixField.getText().trim().toUpperCase();

            // Replace any single quote chars w/ space char or SQL will think th
e ' is the end of the string
            last  = last.replace('\'', ' ');
            first = first.replace('\'', ' ');
            ac    = ac.replace('\'', ' ');
            pre   = pre.replace('\'', ' ');
            sfx   = sfx.replace('\'', ' ');

            // Get rid of the last result displayed if there is one
            if(scrollPane != null)
            getContentPane().remove(scrollPane);

            // Only execute the query if one or more fields have data, else just
 display an empty table
            if(last.length()  > 0 ||
             first.length() > 0 ||
             ac.length()    > 0 ||
             pre.length()   > 0 ||
             sfx.length()   > 0) {

                // build the query and execute it. Provide the results to the ta
ble model
                myDB.doGetQuery(buildQuery(last, first, ac, pre ,sfx));
                ResultSet rset = myDB.getResultSet();
                tblModel = new ListingsTableModel(rset);
                table = new JTable(tblModel);

            } else {
                table = new JTable();
            }

            // Allows the user to only delete on record at a time
            table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

            // Add the table with the results to the contentPane and display it.
            scrollPane = new JScrollPane(table);
            getContentPane().add(scrollPane, BorderLayout.CENTER);
            pack();
            doLayout();
        }

    /* buildQuery */
    /**
     * This method builds a simple select statement for retrieving records from
the Listings table.
     * The select statement is returned as a string. The select statement includ
es the last, first, ac,
     * pre, and sfx parameters as the search strings in the where clause of the
select statement.
     * If more than one parameter has data, buildQuery will combine them with an
 "AND" in the where
     * clause.
     * <pre>
     * PRE:  One or more parameters has length > 0.
     * POST: A SQL select statement is returned that selects records from the Li
stings table.
     * </pre>
     * @param last  create a SQL query that searches Listings where LAST_NAME =
last.
     * @param first create a SQL query that searches Listings where FIRST_NAME =
 first.
     * @param ac    create a SQL query that searches Listings where AREA_CODE =
ac.
     * @param pre   create a SQL query that searches Listings where PREFIX = pre
.
     * @param sfx   create a SQL query that searches Listings where SUFFIX = sfx
.
     * @return a SQL select statement that selects records from the Listings tab
le
     */
        public String buildQuery(String last, String first, String ac, String pr
e, String sfx) {
            String whereClause = " where";
            // Build the where clause
            if(last.length() > 0)
                whereClause += (" LAST_NAME = '" + last + "'");

            if(first.length() > 0) {
                if(whereClause.length() > 6)
                whereClause += " AND";
                whereClause += (" FIRST_NAME = '" + first + "'");
            }

            if(ac.length() > 0) {
                if(whereClause.length() > 6)
                    whereClause += " AND";
                whereClause += (" AREA_CODE = '" + ac + "'");
            }

            if(pre.length() > 0) {
                if(whereClause.length() > 6)
                    whereClause += " AND";
                whereClause += (" PREFIX = '" + pre + "'");
            }

            if(sfx.length() > 0) {
                if(whereClause.length() > 6)
                    whereClause += " AND";
                whereClause += (" SUFFIX = '" + sfx + "'");
            }

            return "select LAST_NAME, FIRST_NAME, AREA_CODE, PREFIX, SUFFIX from
 Listings" + whereClause;
        }
    }// End GetListener inner class
}// End PhoneBookFrame class
```

Save it.

Note We have an Inner class defined in this class. Remember to check it out in the Javadoc page!

ListingsTableModel

In java4_Lesson13, edit **ListingsTableModel** as shown in **blue**:

CODE TO EDIT: ListingsTableModel

```
package greenDB;

import java.sql.ResultSet;
import java.sql.SQLException;

import javax.swing.table.AbstractTableModel;

/* class ListingsTableModel */
/**
 * This class provides methods for displaying the results returned from the List
ings
 * table. The methods are used by a JTable object so the results may displayed i
n a cell format.
 *
 * @author David Green
 * @version 1.0
 */
public class ListingsTableModel extends AbstractTableModel {
    /** The result set from the Listings table to be displayed */
    private ResultSet rs;

    /* ListingsTableModel */
    /**
     * The ListingsTableModel constructor.
     * @param rs the result set from the Listings table to be displayed.
     */
    public ListingsTableModel(ResultSet rs) {
        this.rs = rs;
    }

    /* getRowCount */
    /**
     * Returns the number of rows in the result set.
     * <pre>
     * PRE: True
     * POST: The number of rows in the result set is returned.
     * </pre>
     * @return the number of rows in the result set.
     */
    public int getRowCount() {
        try {
            rs.last();
            return rs.getRow();
        } catch (SQLException e) {
            e.printStackTrace();
            return 0;
        }
    }

    /* getColumnCount */
    /**
     * Returns the number of columns to be displayed for the result set. Note th
at
     * this does not return the number of columns IN the result set. The three p
hone
     * number fields (area code, prefix, and extension) are combined together to
 form
     * a single column for output. This method always returns 3 for Last Name, F
irst
     * Name, and Phone Number.
     * <pre>
     * PRE: True
     * POST: The number 3 is returned.
     * </pre>
     * @return the number 3, for the three output columns Last Name, First Name,
 and Phone Number.
```

```
     */
    public int getColumnCount() {
        return 3;
    }

    /* getColumnName */
    /**
     * Returns the name of the column specified by the index.
     * <pre>
     * PRE:  Column is assigned and 0 >= column <= 2.
     * POST: A column name is returned.
     * </pre>
     * @param column the index of the column name to be returned.
     * @return the column name specified.
     */
    public String getColumnName(int column) {
        try {
            String colName = rs.getMetaData().getColumnName(column + 1);
            // Return column names that look better than the database column nam
es.
            // Since getColumnCount always returns 3 we only look for first 3 co
lumns in
            // the result set.
            if(colName.equals("LAST_NAME"))
                return "Last Name";
            else if(colName.equals("FIRST_NAME"))
                return "First Name";
            else if(colName.equals("AREA_CODE"))
                return "Phone Number";
            else return colName;                  // Should never get here.

        } catch (SQLException e) {
            e.printStackTrace();
            return "";
        }
    }

    /* getValueAt */
    /**
     * Returns the value in the result set at the location specified by row and
column. If column
     * is equal to 2 (the AREA_CODE), combine the AREA_CODE, PREFIX, and SUFFIX
for that row and
     * return the combined string.
     * <pre>
     * PRE:  row and column are assigned and 0 >= column <= 2 and row is within
range.
     * POST: The value in the result set at row and column is returned, or the c
ombined
     *       phone number is returned if column = 2.
     * </pre>
     * @param row the row of the result set whose value is to be returned.
     * @param column the column of the result set whose value is to be returned.
     * @return the value in the result set at row and column is returned, or the
 combined
     * phone number is returned if column = 2.
     */
    public Object getValueAt(int row, int column) {
        try {
            rs.absolute(row + 1);
            // for the 3rd column in the results, combine all of the phone numbe
r fields for output
            if(column == 2)
                return "(" + rs.getObject(column + 1) + ") " + rs.getObject(colu
mn + 2) + "-" + rs.getObject(column + 3);
            else
                return rs.getObject(column + 1);
        } catch (SQLException e) {
            e.printStackTrace();
            return null;
        }
    }

}// End ListingsTableModel class
```

Save it.

phew! This is a lot of work. It's worth doing though to make sure that everyone who encounters our code understands it, appreciates its elegance and efficiency, and can use and modify it as needed. *That*'s the reason we document. OK, now get back to work!

AddListingListener

In java4_Lesson13, edit **AddListingListener** as shown in **blue**:

CODE TO EDIT: AddListingListener

```
package greenDB;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/* class AddListingListener */
/**
 * A listener for when the add button is clicked. The add button looks like a pl
us ("+") sign. The
 * AddListingListener creates and displays an AddListingDialog box when actionPe
rformed is called.
 *
 * @author David Green
 * @version 1.0
 */
class AddListingListener implements ActionListener {
    /** The SimplePhoneBook application frame */
    PhoneBookFrame pbf;

    /* AddListingListener */
    /**
     * The AddListingListener constructor.
     * @param pbFrame the SimplePhoneBook application frame object.
     */
    public AddListingListener(PhoneBookFrame pbFrame) {
        pbf = pbFrame;
    }

    /* actionPerformed */
    /**
     * Instantiates and displays the Add Listings Dialog Box. This method is
     * called when the "+" button is clicked.
     * <pre>
     * PRE:
     * POST: The Add Listings Dialog Box is displayed on screen.
     * </pre>
     * @param aEvent an event generated as a result of the "+" button being clic
ked.
     */
    public void actionPerformed(ActionEvent aEvent) {
        AddListingDialog addDialog = new AddListingDialog(pbf);
        addDialog.setVisible(true);
    }
}// End AddListingListener class
```

Save it.

AddListingDialog

In java4_Lesson13, edit **AddListingDialog** as shown in **blue** below:

CODE TO EDIT: AddListingDialog

```java
package greenDB;

import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.event.DocumentEvent;
import javax.swing.event.DocumentListener;

/* class AddListingDialog */
/**
 * A dialog box for adding a new listing to the Listings table. The AddListingDialog has text
 * fields for gathering the new listing's last name, first name, and phone number. All of the text
 * fields are assigned an InputListener, which is responsible for enabling the "Add" button once
 * all fields contain data. This dialog box is displayed when the user clicks the "+" button on
 * the application frame window.
 *
 * @author David Green
 * @version 1.0
 */
public class AddListingDialog extends JDialog {
    /** A text field for entering the new phone listing's last name */
    private JTextField lNameField    = new JTextField(16);
    /** A text field for entering the new phone listing's first name */
    private JTextField fNameField    = new JTextField(16);
    /** A text field for entering the new phone listing's area code */
    private JTextField areaCodeField = new JTextField(2);
    /** A text field for entering the new phone listing's prefix */
    private JTextField prefixField   = new JTextField(2);
    /** A text field for entering the new phone listing's extension */
    private JTextField suffixField   = new JTextField(3);
    /** A button which, when clicked, will add the new listing to the Listings table */
    private JButton addButton;

    /* AddListingDialog */
    /**
     * The AddListingDialog constructor. Creates a dialog box for adding a new listing to the
     * Listings table.
     * @param owner the Frame from which the dialog is displayed.
     */
    public AddListingDialog(final JFrame owner) {
        // set the dialog title and size
        super(owner, "Add Listing", true);
        setSize(280, 150);

        // Create the center panel which contains the fields for entering the new listing
        JPanel center = new JPanel();
        center.setLayout(new GridLayout(3, 2));
        center.add(new JLabel(" Last Name:"));
        center.add(lNameField);
        center.add(new JLabel(" First Name:"));
        center.add(fNameField);

        // Here we create a panel for the phone number fields and add it to the
center panel.
        JPanel pnPanel = new JPanel();
        pnPanel.add(new JLabel("("));
        pnPanel.add(areaCodeField);
        pnPanel.add(new JLabel(") "));
        pnPanel.add(prefixField);
        pnPanel.add(new JLabel("-"));
        pnPanel.add(suffixField);
        center.add(new JLabel(" Phone Number:"));
        center.add(pnPanel);

        // Create the south panel which contains the buttons
        JPanel south = new JPanel();
        addButton    = new JButton("Add");
        JButton cancelButton = new JButton("Cancel");
        addButton.setEnabled(false);
        south.add(addButton);
        south.add(cancelButton);

        // Add listeners to the fields and buttons
        lNameField.getDocument().addDocumentListener(new InputListener());
        fNameField.getDocument().addDocumentListener(new InputListener());
        areaCodeField.getDocument().addDocumentListener(new InputListener());
        prefixField.getDocument().addDocumentListener(new InputListener());
        suffixField.getDocument().addDocumentListener(new InputListener());

        areaCodeField.getDocument().addDocumentListener(new PhoneDocumentListene
r(areaCodeField, 3));
        prefixField.getDocument().addDocumentListener(new PhoneDocumentListener(
prefixField, 3));
        suffixField.getDocument().addDocumentListener(new PhoneDocumentListener(
suffixField, 4));

        areaCodeField.addFocusListener(new PhoneFocusListener());
        prefixField.addFocusListener(new PhoneFocusListener());
        suffixField.addFocusListener(new PhoneFocusListener());

        // listeners to close the window
        addButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent aEvent) {
                    // ((PhoneBookFrame)owner).doInsertQuery(buildQuery());
                    DatabaseManager ownersDB = ((PhoneBookFrame)owner).getDBMana
ger();
                    ownersDB.doInsertQuery(buildQuery());
                    dispose();
                }
        });

        cancelButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent aEvent) {
                    dispose();
            }
        });

        // Add the panels to the dialog window
        Container contentPane = getContentPane();
        contentPane.add(center, BorderLayout.CENTER);
        contentPane.add(south,  BorderLayout.SOUTH);
    }

    /* buildQuery */
    /**
     * This method builds an insert statement for inserting a new record into th
```

```
e Listings table.
     * The insert statement is returned as a string. The insert statement will i
nclude the last name,
     * first name, area code, prefix, and extension that the user entered in the
 add listing dialog
     * box.
     * <pre>
     * PRE:  All of the fields in the Add Listing Dialog box contain data.
     * POST: A SQL insert statement is returned that inserts a new listing into
the Listings table.
     * </pre>
     * @return a SQL insert statement that will insert a new listing in the List
ings table.
     */
    public String buildQuery() {
        // Get the data entered by the user, trim the white space and change to
upper case
        String query = "";
        String last  = lNameField.getText().trim().toUpperCase();
        String first = fNameField.getText().trim().toUpperCase();
        String ac    = areaCodeField.getText().trim().toUpperCase();
        String pre   = prefixField.getText().trim().toUpperCase();
        String sfx   = suffixField.getText().trim().toUpperCase();

        // Replace any single quote chars with a space char so the string will n
ot get truncated by SQL
        last  = last.replace('\'', ' ');
        first = first.replace('\'', ' ');
        ac    = ac.replace('\'', ' ');
        pre   = pre.replace('\'', ' ');
        sfx   = sfx.replace('\'', ' ');

        // build  and return the insert statement
        return new String("insert into Listings values ('" + last + "', '" +
                                                         first + "', '" +
                                                         ac + "', '" +
                                                         pre + "', '" +
                                                         sfx + "')");
    }

    /* inner class InputListener */
    /**
     * This inner class is a Listener for the text fields of the Add Listing Dia
log Box.
     * The listener keeps track of whether all fields (last name, first name, ar
ea code,
     * prefix, and extension) have data entered in them. If all fields contain d
ata, the
     * "Add" button of the Add Listing Dialog box is enabled for the user. If an
y one of
     * the fields is empty or if the phone number fields contain fewer character
s than
     * required, the "Add" button is unavailable.
     *
     * @author David Green
     * @version 1.0
     */
    class InputListener implements DocumentListener {

    /* insertUpdate */
    /**
     * This method is called when data is put in the text field, either by typin
g or by a paste operation.
     * This method tracks the number of characters entered in the field. If all
fields, last name,
     * first name, area code, prefix, and extension have data and the phone numb
er fields contain the correct
     * number of characters (that is, 3 characters for area code and prefix and
4 characters for extension),
     * then the Add Listing Dialog box "Add" button is enabled.
     * <pre>
     * PRE:
     * POST: Add Listing Dialog Box "Add" button is enabled if all fields have t
he required number of characters
     * entered.
     * </pre>
     * @param dEvent the document event
     */

        public void insertUpdate(DocumentEvent dEvent) {
            // If first name, last name have data and phone number is complete
            // enable the add button, give it focus and make it clickable if
            // user hits <enter>.
            if(lNameField.getDocument().getLength()     > 0 &&
                fNameField.getDocument().getLength()     > 0 &&
                areaCodeField.getDocument().getLength() == 3 &&
                prefixField.getDocument().getLength()   == 3 &&
                suffixField.getDocument().getLength()   == 4) {

                addButton.setEnabled(true);
                if(dEvent.getDocument() == suffixField.getDocument()) {
                    addButton.requestFocus();
                    getRootPane().setDefaultButton(addButton);
                }
            }
        }


    /* removeUpdate */
    /**
     * This method is called when data is removed from the text field, either by
 backspacing or highlighting and
     * deleting. This method will track the number of characters removed from th
e field. If any of the fields
     * last name, first name, area code, prefix, and extension contain less than
 the required number of characters
     * the "Add" button of the Add Listing Dialog box is disabled.
     * <pre>
     * PRE:
     * POST: Add Listing Dialog Box "Add" button is disabled if any of the field
s contain less than the required
     *       number of characters.
     * </pre>
     * @param dEvent the document event
     */
        public void removeUpdate(DocumentEvent dEvent) {
            // If last name or first name don't have data or phone number
            // is  not complete, disable the Add button.
            if(lNameField.getDocument().getLength()   == 0 ||
             fNameField.getDocument().getLength()   == 0 ||
             areaCodeField.getDocument().getLength() < 3 ||
             prefixField.getDocument().getLength()   < 3 ||
             suffixField.getDocument().getLength()   < 4 )

            addButton.setEnabled(false);
        }

    /** Not implemented */
        public void changedUpdate(DocumentEvent dEvent) {}

    }// End InputListener inner class
}// End AddListingDialog class
```

Save it.

Note Another Inner class--remember to look for this one on the Javadoc page too.

PhoneDocumentListener

In java4_Lesson13, edit **PhoneDocumentListener** as shown in **blue** below:

CODE TO EDIT: PhoneDocumentListener

```
package greenDB;

import javax.swing.JTextField;
import javax.swing.event.DocumentEvent;
import javax.swing.event.DocumentListener;

/* class PhoneDocumentListener */
/**
 * A listener that is applied to any of the telephone number fields (area code, prefix,
 * and extension). The purpose of this listener is to prevent more than the expected number
 * of characters from being entered in the telephone number fields. That is, the area code and
 * prefix fields might only be allowed to contain 3 characters each while the extension field
 * might only be allowed to contain four characters. The PhoneDocumentListener class accomplishes
 * this by accepting a 'numbers allowed' parameter during construction. Every time a character is
 * entered in the phone number field to which this listener applies, the insertUpdate method is
 * called. The insertUpdate method checks the number of characters in the field and if the number
 * is equal to 'numbers allowed', focus is transferred to the next component.
 *
 * @author David Green
 * @version 1.0
 */

class PhoneDocumentListener implements DocumentListener {
    /** The phone number text field to which this listener applies */
    private JTextField txtField;
    /** The number of characters that will cause focus to be transferred */
    private int numsAllowed;

    /* PhoneDocumentListener */
    /**
     * The PhoneDocumentListener constructor.
     * @param tf The phone number text field to which this listener applies.
     * @param numsAllowed The number of characters that can be entered in this field.
     */
    public PhoneDocumentListener(JTextField tf, int numsAllowed) {
        txtField = tf;
        this.numsAllowed = numsAllowed;
    }

    /* insertUpdate */
    /**
     * Called when a character is typed in the field to which this listener is applied.
     * The field is examined for number of characters and if the number is equal to the
     * numbers allowed, as specified during construction, focus is transferred to the next
     * component.
     * <pre>
     * PRE:
     * POST: Focus is transferred if field length equals numsAllowed; else nothing happens.
     * </pre>
     * @param dEvent An event generated as a result of a character being entered in the
     * telephone number field to which this listener is applied.
     */
    public void insertUpdate(DocumentEvent dEvent) {
        if(dEvent.getDocument().getLength() == numsAllowed)
            txtField.transferFocus();
    }

    /** Empty implementation. Method necessary for implementation of DocumentLis
tener */
    public void removeUpdate(DocumentEvent dEvent) {}

    /** Empty implementation. Method necessary for implementation of DocumentLis
tener */
    public void changedUpdate(DocumentEvent dEvent) {}

}// End PhoneDocumentListener class
```

Save it.

PhoneFocusListener

In java4_Lesson13, edit **PhoneFocusListener** as shown in **blue** below:

CODE TO EDIT: PhoneFocusListener

```
package greenDB;

import java.awt.event.FocusEvent;
import java.awt.event.FocusListener;
import javax.swing.JTextField;

/* class PhoneFocusListener */
/**
 * A listener that will delete any data currently in the telephone number fields
 (area code, prefix,
 * and extension) whenever focus is detected for those fields. This listener is
used in conjunction with
 * the PhoneDocumentListener to prevent more than the expected number of charact
ers from being entered
 * in the telephone number fields. That is, the area code and prefix fields will
 only be allowed to
 * contain three characters each while the extension field will only be allowed
to contain four characters.
 *
 * @author David Green
 * @version 1.0
 */
class PhoneFocusListener implements FocusListener {

    /* focusGained */
    /**
     * Called when the field to which this listener applies gains focus. This me
thod will delete
     * any data currently contained in the field.
     * <pre>
     * PRE:  True.
     * POST: Any data in the telephone number field to which this listener appli
es is deleted.
     * </pre>
     * @param fEvent An event generated as a result of focus being gained on thi
s telephone number field.
     */
    public void focusGained(FocusEvent fEvent) {
        JTextField tf = (JTextField)fEvent.getSource();
        tf.setText("");
    }

    /** Not implemented */
    public void focusLost(FocusEvent fEvent){}

}// End PhoneFocusListener class
```

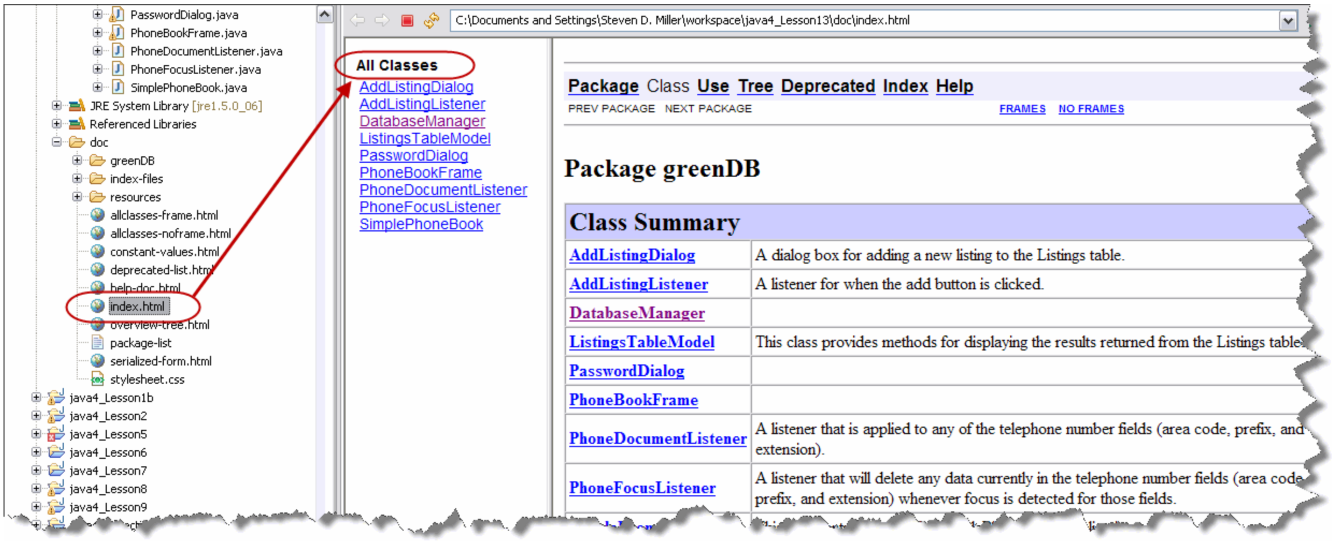
Save it.

The Package Documentation

All classes should be fine and error-free. Remember that the class to start it all was **SimplePhoneBook**. Select **SimplePhoneBook** and run it to verify that all is well. Everything should work the same as before; all that we changed was documentation.

Select the **java4_Lesson13** Project so that it is highlighted. In the top Eclipse Menu Bar, select **Project | Generate Javadoc...**. In the **Generate Javadoc** window that opens, click the **Configure...** button to get to the directory **C:\java\bin**. Select **javadoc.exe** to get the javadoc executable code. Keep the default of **Private** as it is. This will let us see all members (private, package, protected, and public). Keep the Destination as it is so the documentation will be located with the Project. Click **Finish**.

It's the same process we performed earlier--we are simply now making the API html pages for *all* of the classes. This time we want to see a listing with links to all of the classes, so open **doc | index.html**.



Package greenDB

Class Summary	
AddListingDialog	A dialog box for adding a new listing to the Listings table.
AddListingListener	A listener for when the add button is clicked.
DatabaseManager	
ListingsTableModel	This class provides methods for displaying the results returned from the Listings table
PasswordDialog	
PhoneBookFrame	
PhoneDocumentListener	A listener that is applied to any of the telephone number fields (area code, prefix, and extension).
PhoneFocusListener	A listener that will delete any data currently in the telephone number fields (area code prefix, and extension) whenever focus is detected for those fields.

On *your* generated docs index page, click on the links as you would any html page to see your class documentation. In fact, look at these pages until you see all that is offered, and how it all relates to the comments we wrote. For example, check out **PhoneBookFrame.html** and **AddListingDialog.html** and look for a **Nested Class Summary** to see their inner classes. Good documentation makes your work more refined and finished.

Additional Resources

We gone over many JDBC elements, but lots of additional techniques are available. For example, you might want to:

- Connect to a database in Windows and use the JDBC/ODBC bridge.
- Allow your code to connect to multiple databases, and use the **java.sql.DriverManager** class to assist in finding the correct driver for the current database
- Allow multiple connections to the database by using a **javax.sql.PooledConnection** when handling multiple threads of connectivity.

Here are additional links to assist you with your database applications:

SQL

- Interactive Online SQL Training
- A SQL Tutorial
- W3Schools' SQL Tutorial

Or just go to Google and do a search on **SQL tutorial**. Our last search got 149,000 hits--there are LOTS of tutorials on SQL out there! And, of course, for those who like something in their hands, there are **lots** of books on SQL!

Javadoc

- Oracle's page on Javadoc Technology, with the Javadoc Tool Reference Page
- Oracle Developer Network Javadoc tool page
- Oracle Developer Network How to Write Doc Comments for the Javadoc Tool page, with a list of tags

Annotations

Annotations look similar to Javadoc tags (use of @) but they are actually a form of metadata that can be added to Java source code. Annotations complement Javadoc tags. In general, if the markup is intended to affect or produce documentation, it should probably be a Javadoc tag; otherwise, it should be an annotation.

The use of the "@" symbol in both Javadoc comments and in annotations is not coincidental-—they are related conceptually. But note that the Javadoc deprecated tag starts with a lowercase **d** and the annotation starts with an uppercase **D**. We will see them more when we look at enterprise applications. For now, here are a few resources for you:

- Oracle's Annotation Tutorial
- Annotations (from *The Java Programming Language* (from Enhancements in JDK 5)
- If you are **really** into it, look up the Annotation Processing Tool.

What's Next?

In the next course we will continue our work with applications, but delve into *distributed* computing as well. Everything you learned in this course--GUIs, Exceptions, Threads, Database Connectivity, Documentation--you'll use again. You're cultivating some great skills!