Java Programming 3: Java Programming Foundations

Software Design Using Java

Welcome to the next phase of your Java education. In this third course of the OST Java Series, we'll dig deeper into the API and explore the foundational tools provided by Java that allow us to write big, modular, integrated programs.

Course Objectives

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

- develop code that is clean, reusable, flexible and user-friendly.
- design object-oriented software that utilizes fundamental programming constructs, multiple inheritence, abstraction, and packages.
- implement interfaces using the Model-View-Controller (MVC) architecture.
- enhance Java flexibility with enumeration, casting, interface extension, generics, and the collection framework.
- handle and map images.
- deploy applets and applications using Eclipse and Java JAR files.

The focus of this course is to build the programmer's repertoire of fundamental Java application capabilities. You will achieve an understanding of the structure and purposes for many classes in the Java API. In-depth experience with user interfaces, event and exception handling, Java I/O and the Collection Framework 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.

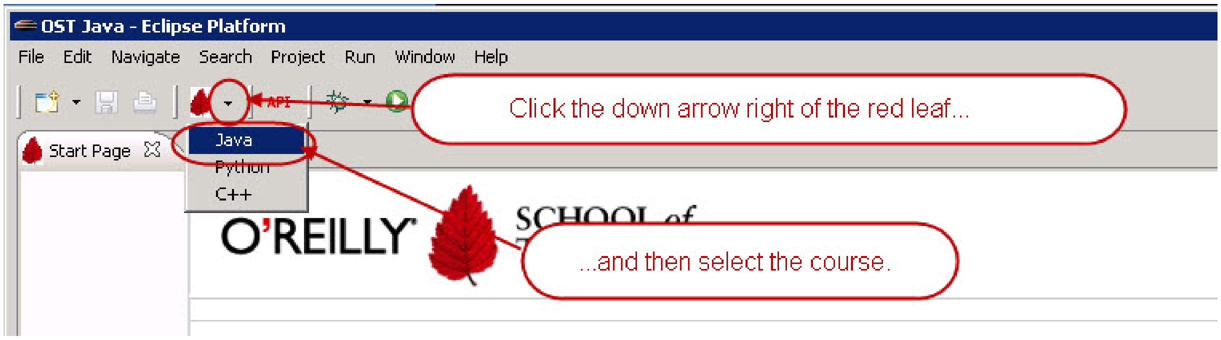
From beginning to end, you will learn by doing your own Java projects, within our Eclipse Learning Sandbox we affectionately call "Ellipse". These projects will add to your portfolio and provide needed experience. Besides a browser and internet connection, all software is provided online by the O'Reilly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

Perspectives and the Red Leaf Icon

The Ellipse Plug-in for Eclipse, developed by the O'Reilly School of Technology, adds the Red Leaf icon 🍁 to the Eclipse toolbar. This icon is your "panic button." Eclipse is versatile and enables you to rearrange your views, toolbars, and such. But if you get confused and want to return to the default perspective (window layout), 🍁 can help.

The 🍁 offers functions that allow you to:

- Reset the current perspective. You do that by clicking the Red Leaf icon.
- Change to different perspectives. You can change perspectives by clicking the drop-down arrow beside the Red Leaf icon and select the perspective designed for your particular course. Most of the perspectives look similar, but subtle changes may be present "under the hood," so it's best to use the perspective designed specifically for each course.


OST Java - Eclipse Platform
File Edit Navigate Search Project Run Window Help
API
Start Page
Java
Python
C++
O'REILLY
SCHOOL of

Click the down arrow right of the red leaf...

...and then select the course.

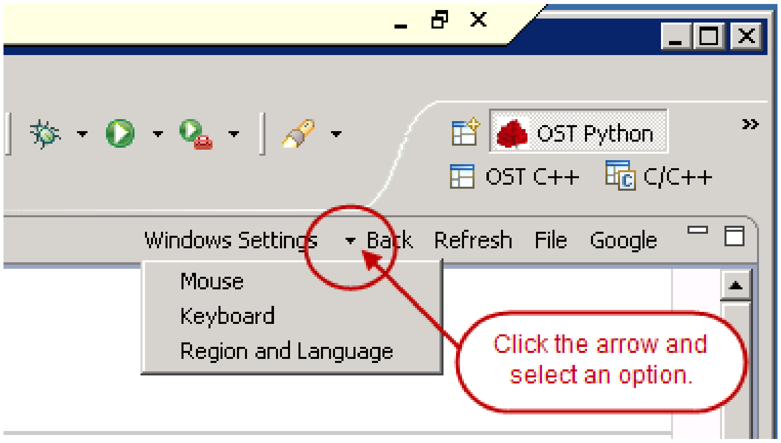
Working Sets

All projects created in Eclipse exist in the workspace directory of your account on our server. As you create multiple projects for each lesson, in each course, this space could become pretty cluttered. To make sure our work stays organized, we'll use *working sets*. A working set is a logical view of the workspace that behaves like a folder, but is really only an association of files. The difference between a working set and a folder is that a working set doesn't actually exist in the file system. A working set is a convenient way to group related items together. You can assign a project to one or more working sets. In some cases, for instance, when using the Python extension to Eclipse, new projects are created without regard for working sets and are placed in the workspace, but not assigned to a working set (appearing instead in "Other Projects"). To assign one of these projects to a working set, right-click on the project name and select the "Assign Working Sets" menu item.

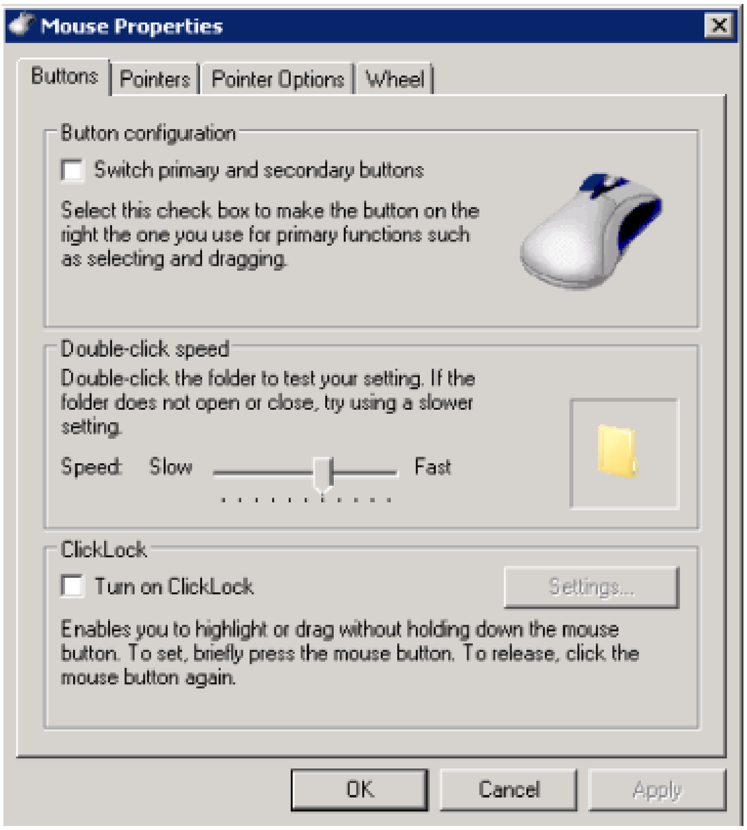
When you select a perspective using the Red Leaf Icon, working sets for that particular course are created if they do not already exist. You can turn the working set display on and off in Eclipse, following these instructions. Working sets allow you to limit the detail that you see at any given time.

Windows Settings

If you like, you can set your own Windows mouse, keyboard, and region; for example, if you are left-handed, you can switch the left and right button functionality on the mouse, or change date fields to use date formats for your local region. Click the down arrow on the Windows Settings button at the top right of the screen:



We won't discuss all of the details of these dialog boxes, but feel free to ask your instructor if you have questions.



Button configuration

□ Switch primary and secondary buttons

Select this check box to make the button on the right the one you use for primary functions such as selecting and dragging.

Double-click speed

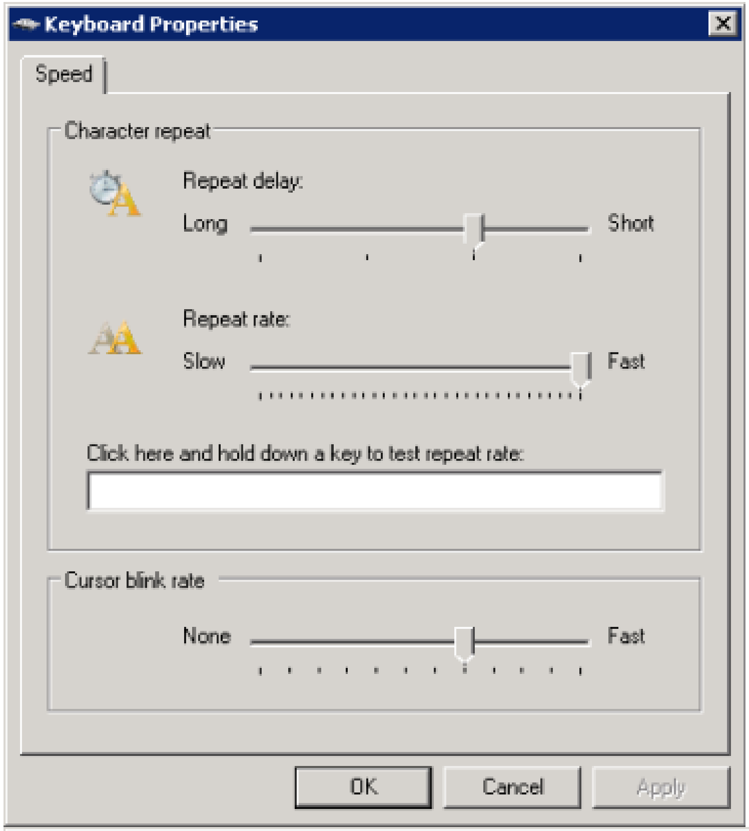
Double-click the folder to test your setting. If the folder does not open or close, try using a slower setting.

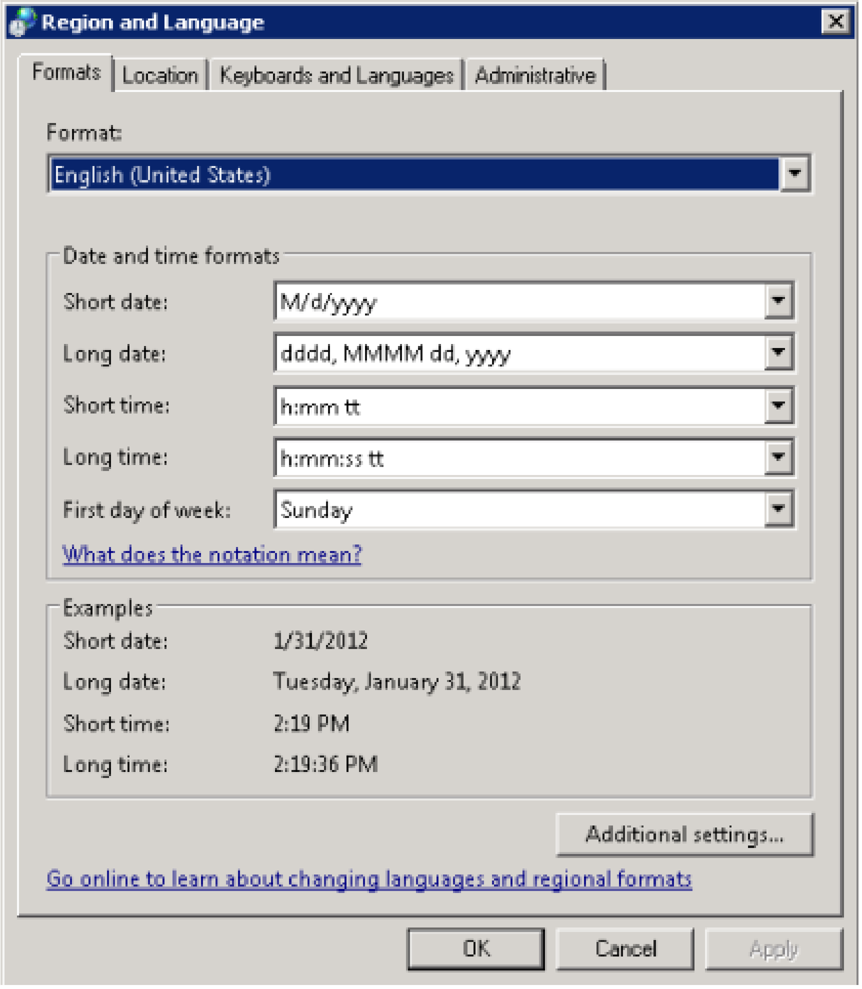
Speed: Slow — Fast

ClickLock

□ Turn on ClickLock

Enables you to highlight or drag without holding down the mouse button. To set, briefly press the mouse button. To release, click the mouse button again.





Format: English (United States)

Short date: M/d/yyyy

Long date: dddd, MMMM dd, yyyy

Short time: h:mm tt

Long time: h:mm:ss tt

First day of week: Sunday

Short date: 1/31/2012

Long date: Tuesday, January 31, 2012

Short time: 2:19 PM

Long time: 2:19:36 PM

Clean Classes

Okay, let's get to work! First, we'll focus on some of the smaller programming elements that will be most useful when we start designing larger systems.

Get Started: Main and main

In order to promote modularity and code reuseability, an Object should be "true to itself." That means a Class should define only aspects of *itself*—no extra methods or information about ways to use it. Generally, a class should not be designed in anticipation of its **usage**, but according to its own **identity**.

One way to keep application code separate from the code used to *start* the application is to write a class whose sole purpose is to start the application. Applications don't have a web browser to start them, so we'll use the **main()** method to do that. Java Programmers usually separate the thing being started from the starting of the thing, by making a Class named **Main.java**. The **Main.java** class serves a single purpose: it holds a **main()** method to instantiate and get the application kick-started.

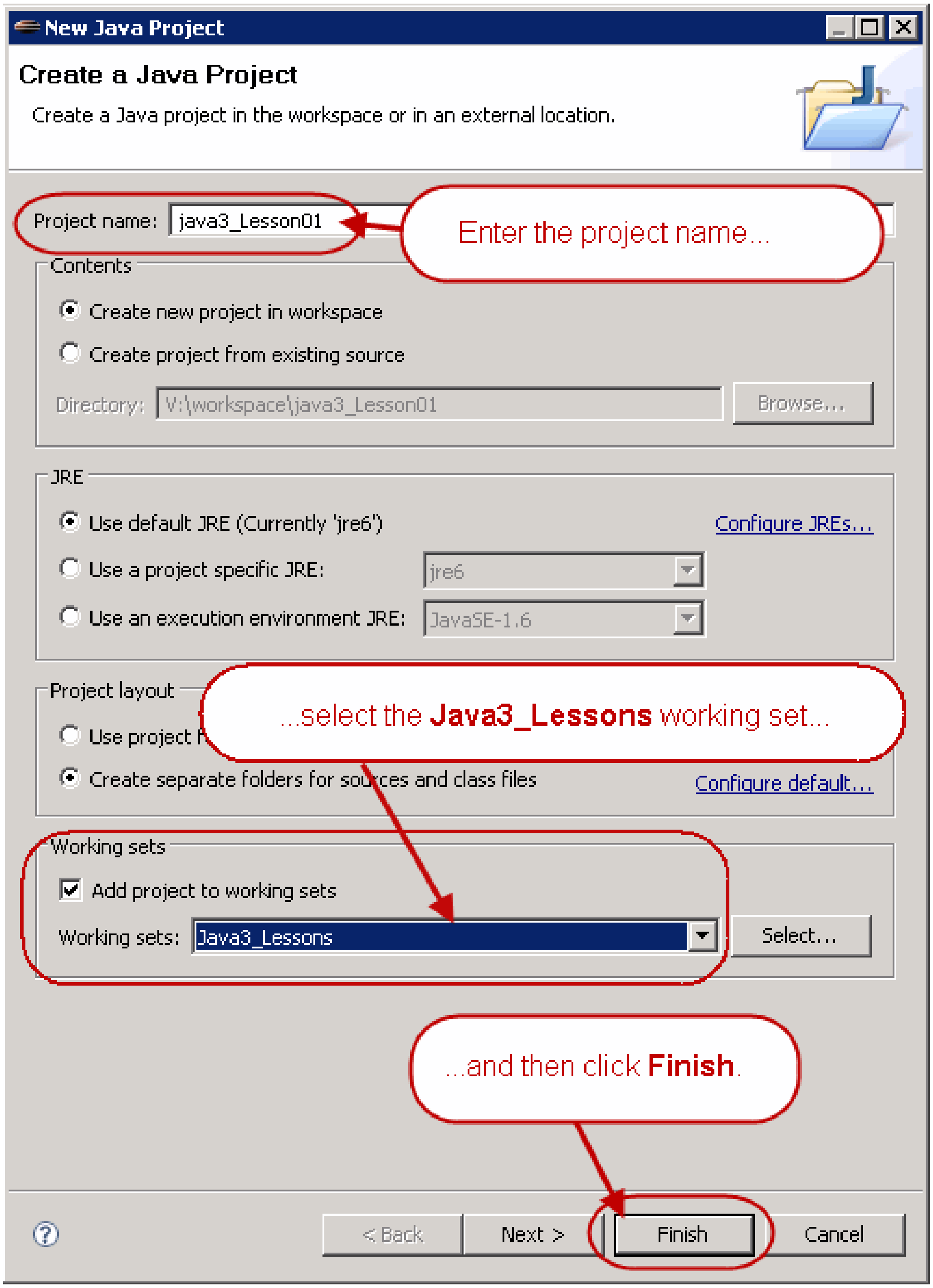
In our first example, we'll create a performance report for a group of salespeople. The **Main** class will instantiate the **SalesReport** class and call a method.

Write your **Main** class first, and then define a **SalesReport** class with the actual application code.

Keep your projects organized in Working Sets. If you haven't done so already, use the down arrow beside the Red Leaf icon on the toolbar to create the working sets for this course.

Use the **Java3_Lessons** working set. For details, see the Working Sets instructions.

We created projects and classes in earlier courses in this Java series, but let's go over it once more to refresh your memory. To create a new project for Lesson 1, select **File | New | Project** and choose **Java Project**. Name it **java3_Lesson01** and click **Finish**.



Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: java3_Lesson01

Contents

● Create new project in workspace

○ Create project from existing source

Directory: V:\workspace\java3_Lesson01

JRE

● Use default JRE (Currently 'jre6')

○ Use a project specific JRE: jre6

○ Use an execution environment JRE: JavaSE-1.6

Project layout

○ Use project f

● Create separate folders for sources and class files

Working sets

☑ Add project to working sets

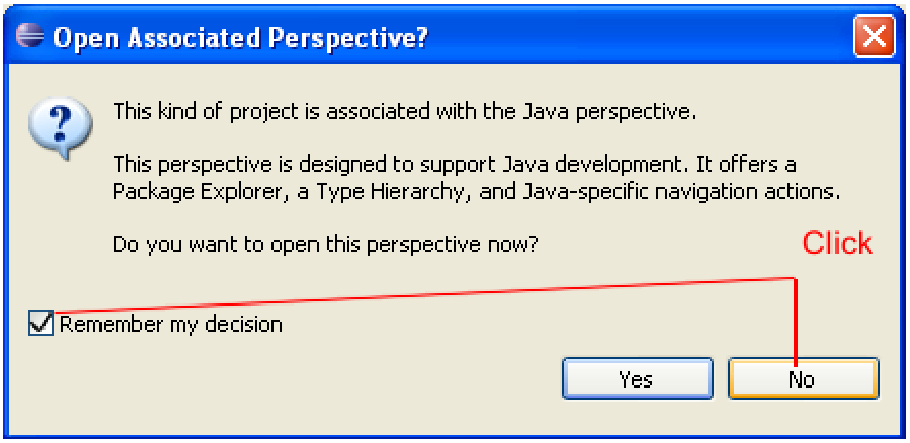
Working sets: Java3_Lessons

Enter the project name...

...select the **Java3_Lessons** working set...

...and then click **Finish**.

If you see the dialog below, go ahead and check the **Remember my decision** box and then click **No**.



This kind of project is associated with the Java perspective.

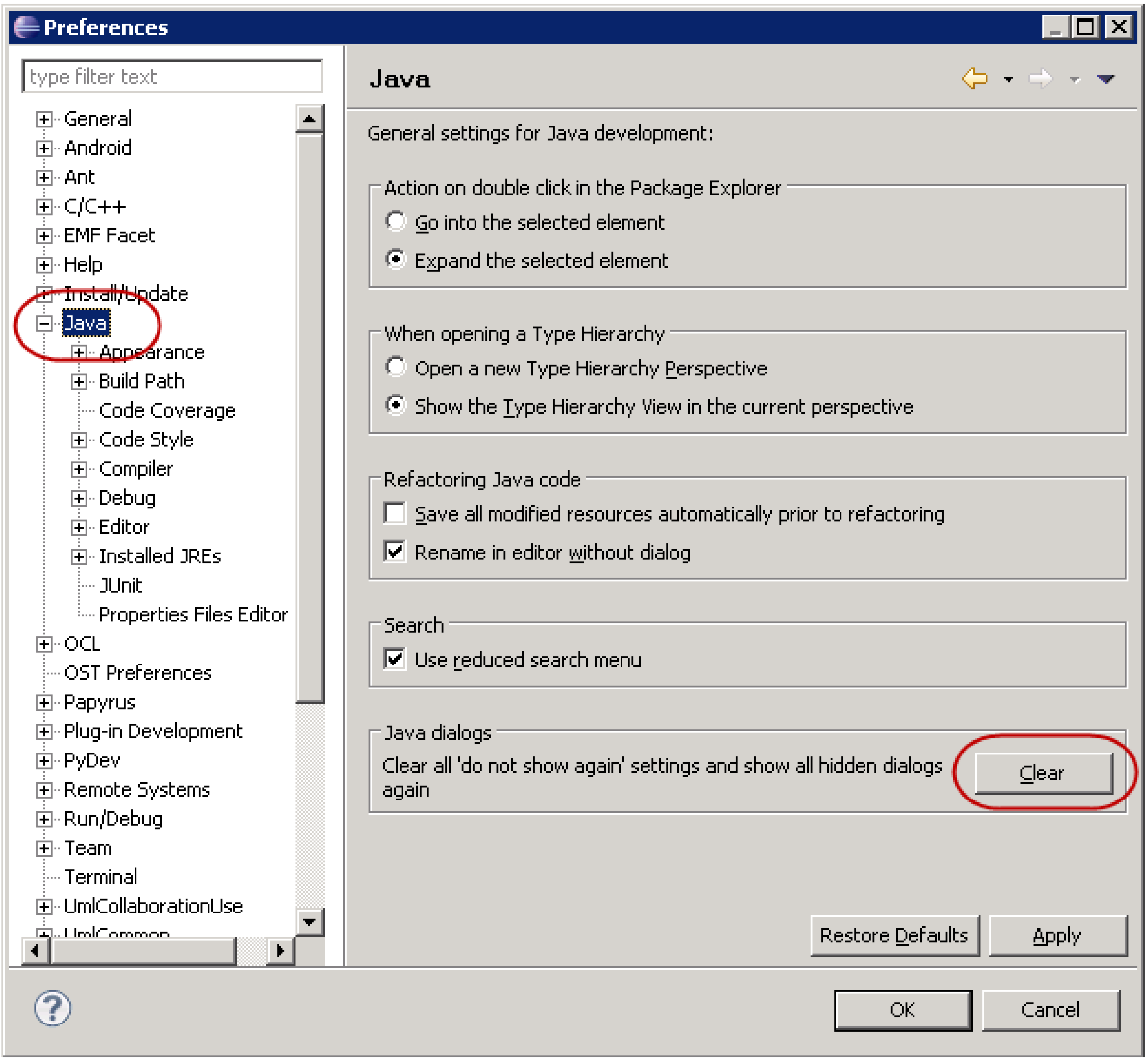
This perspective is designed to support Java development. It offers a Package Explorer, a Type Hierarchy, and Java-specific navigation actions.

Do you want to open this perspective now?

☑ Remember my decision

Click

If you clicked **Yes** on the above dialog by mistake, select the **Windows** menu and click **Preferences**. When the dialog appears, click on the **Java** item on the left. Then, click the **Clear** button as shown:



Java

General settings for Java development:

Action on double click in the Package Explorer

○ Go into the selected element

● Expand the selected element

When opening a Type Hierarchy

○ Open a new Type Hierarchy Perspective

● Show the Type Hierarchy View in the current perspective

Refactoring Java code

□ Save all modified resources automatically prior to refactoring

☑ Rename in editor without dialog

Search

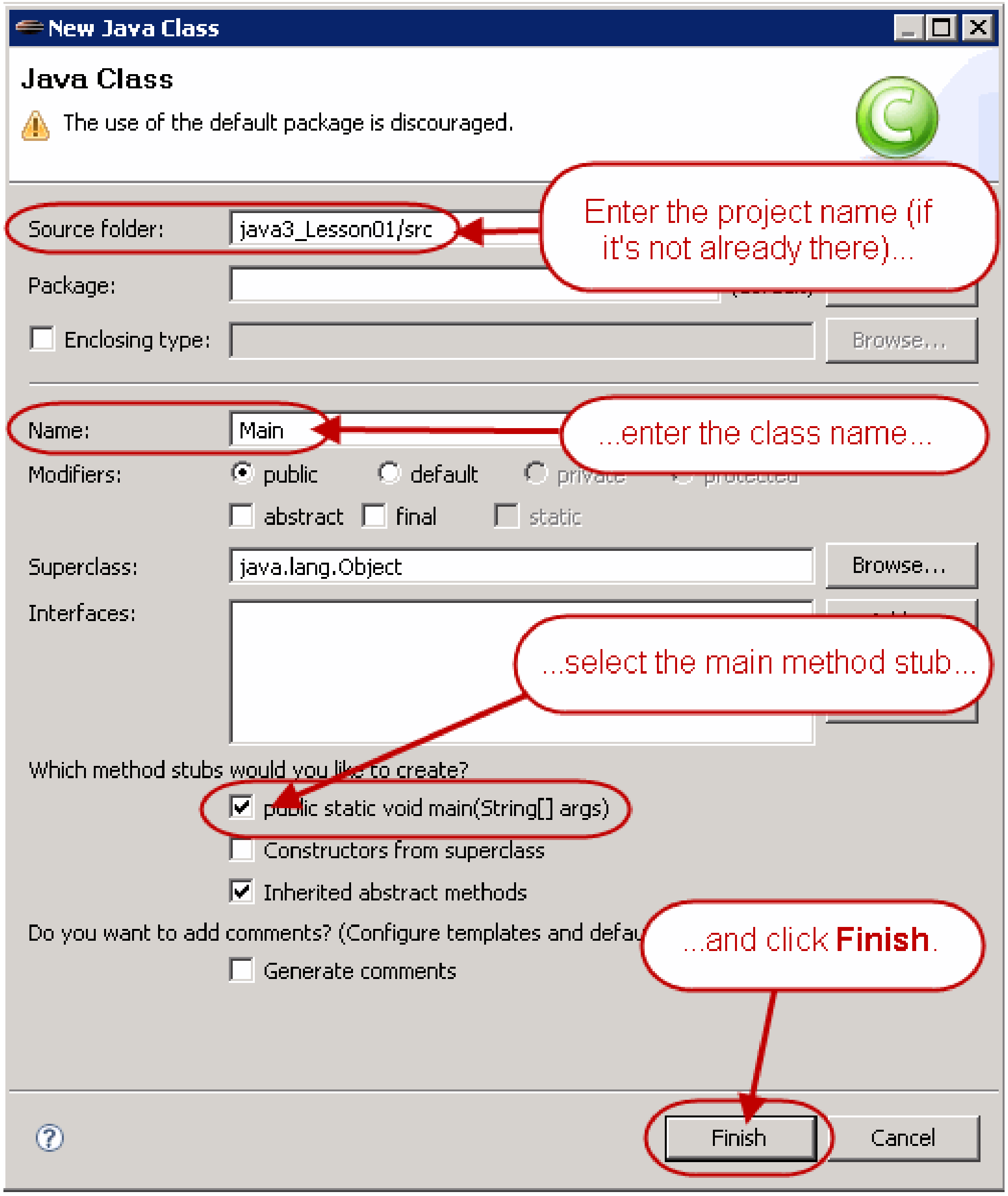
☑ Use reduced search menu

Java dialogs

Clear all 'do not show again' settings and show all hidden dialogs again

Clear

Now, let's create our first Class. Select the **java3_Lesson01** project you just created, select the **src** folder, and then right-click to get to the popup menu. Select **New | Class** (or **New | Other** and double-click **Class**). In the New Java Class window that opens, enter the information as shown:



New Java Class

Java Class

⚠ The use of the default package is discouraged.

Source folder: java3_Lesson01/src

Package:

☐ 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 defau

☐ Generate comments

Enter the project name (if it's not already there)...

...enter the class name...

...select the main method stub...

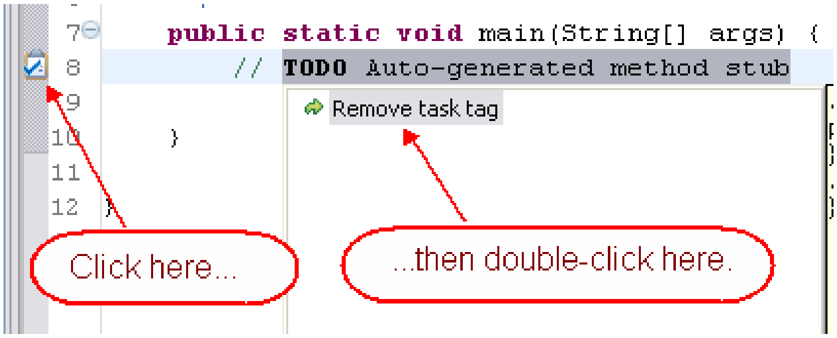
...and click **Finish**.

The new class appears in the Editor window:

OBSERVE: Main Application

```
public class Main {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}
```

To remove the **// TODO** comment and the symbol on the left, click the check box symbol once, and then, in the suggestion box that opens, double-click **Remove task tag**:



```
    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

Edit the **Main** class's **main()** method by adding the **blue** code shown:

CODE TO TYPE:

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

        SalesReport mySalesInfo = new SalesReport();
        mySalesInfo.testMe();
    }
}
```

Save it.

SalesReport is underlined in red because we have not defined it yet. Ignore that for now, and take a look at the class we have created:

OBSERVE:

```
public class Main{
    public static void main(String[] args){
        SalesReport mySalesInfo = new SalesReport();
        mySalesInfo.testMe();
    }
}
```

Here, the **Main** class has only one method: **main(String[] args)**. Its sole purpose is to create an instance of **SalesReport** (the application object), and then call a method to get it started: **mySalesInfo.testMe()**.

Tip

Generally, good practices dictates that only these two elements are located within the **main()** method:

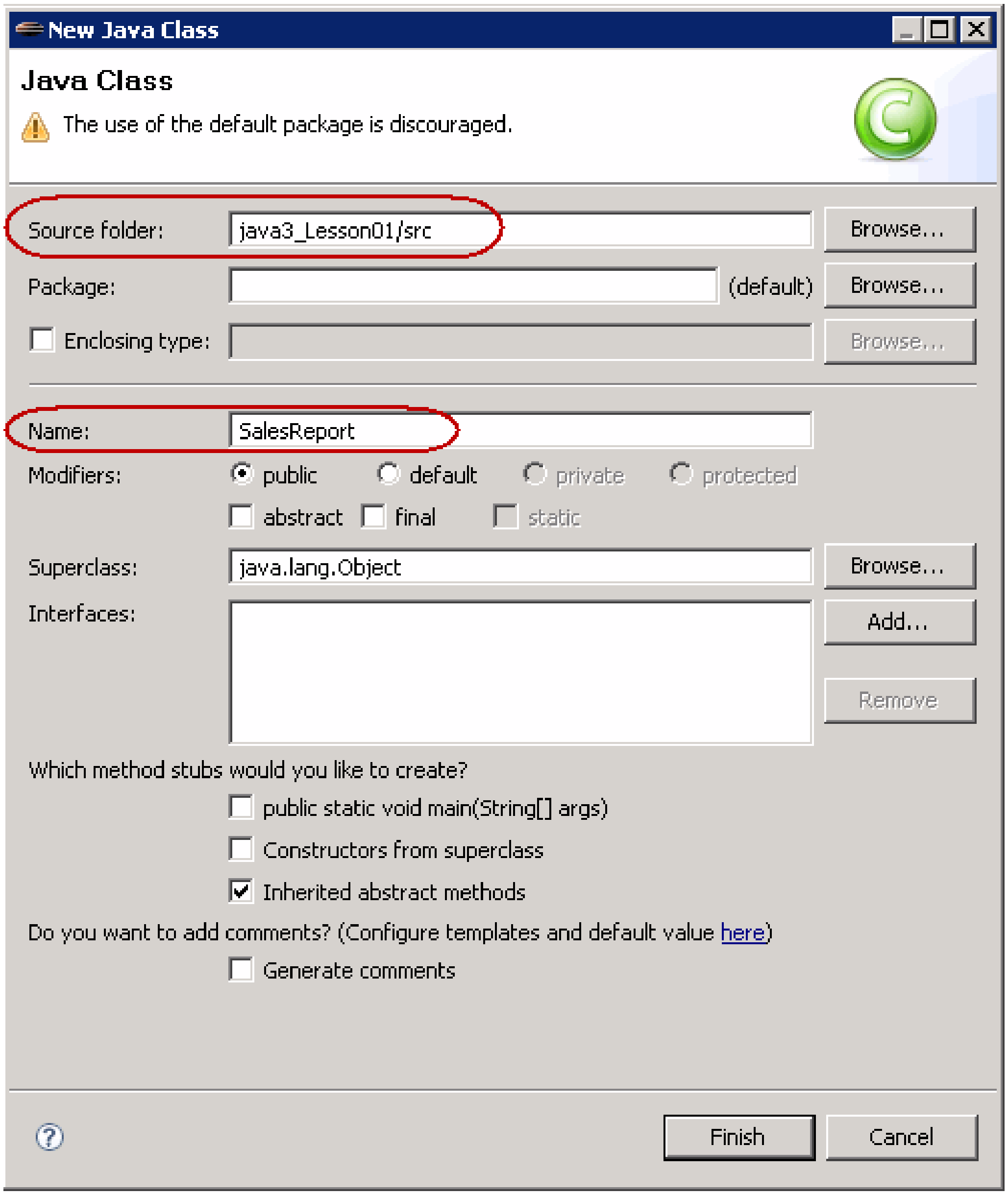
1. a message to instantiate a class.
2. a message to get that class started.

Any setup code required before starting the main program class goes inside the **main()** method.

Now, let's define that **SalesReport** class!

The Application: SalesReport

Create another new class in the **java3_Lesson01** project **src** folder, and name it **SalesReport**. (From now on when we use screen shots, we'll circle the non-default items for you to choose or enter, in red):



Java Class

The use of the default package is discouraged.

Source folder: java3_Lesson01/src

Package: (default)

□ 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

Under "Which method stubs would you like to create?" we do *not* choose **public static void main(String [] args)**, because we already have code in our **Main.java** to start this application.

Edit the **SalesReport** class and add the code shown in **blue**:

CODE TO TYPE:

```
import java.util.Scanner;

public class SalesReport {
    int salespeople;
    int sum;
    int sales[];

    public SalesReport(){
        this.salespeople = 3;
        this.sales = new int[salespeople];
    }

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

    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 + ": ");
            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 + "          " + sales[i]);
            sum = sum + sales[i];
        }
        System.out.println("\nTotal sales: " + sum);
    }
}
```

This class won't run because it does not have a **main()** method, and it does not extend **Applet**. Don't worry though, we'll show you how to run it in just a little bit.

OBSERVE: SalesReport

```
import java.util.Scanner;

public class SalesReport{
  int salespeople;
  int sum;
  int sales[];

  public SalesReport(){
    this.salespeople = 3;
    this.sales = new int[salespeople];
  }

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

  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 + ": ");
      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 + "         " + sales[i]);
      sum = sum + sales[i];
    }
    System.out.println("\nTotal sales: " + sum);
  }
}
```

In the constructor, we are instantiating a **sales** array, to track sales numbers for three **salespeople**.

The **getSalesInput()** method creates an instance of **Scanner**, named **scan**, using the **System.in** object as its parameter. This allows the **Scanner** to scan the console input and store information typed. Next, the method loops through each of the **sales** array elements and prompts the user to input the sales for each salesperson. Then the **scan** object is queried for the next integer typed into the console and stores that information in the current element of the **sales** array.

The **provideSalesOutput()** method prints a header to show what it is outputting to the console. Then it loops through the **sales** array and outputs the identifying number of the salesperson and that salesperson's sales. Our method accomplishes this by getting the value from the array. While it is looping through, it is adding each sale to the local **sum** variable. Then our method outputs the total sales.

We'll call the **testMe()** method from the **Main** class's **main()** method to test the **SalesReport** class. It calls the **getSalesInput()** method to fill the **sales** array and then it calls the **produceSalesOutput()** method to output the results to the console.

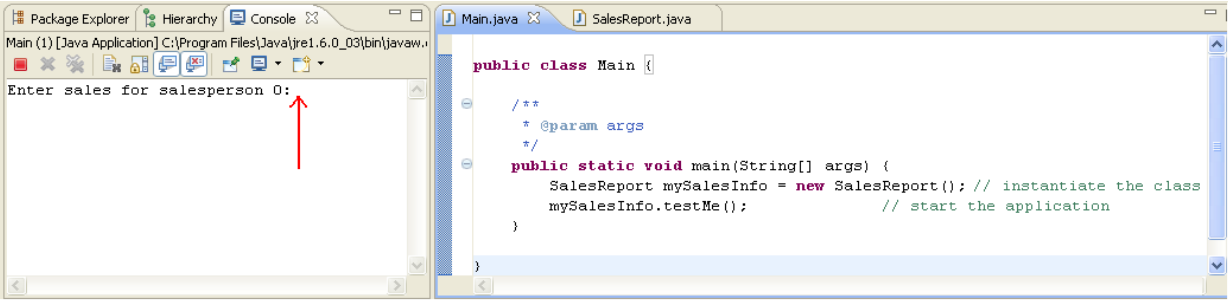
Save it. Now right-click in the editor and choose **Run As**. Hmm. The only option is **Run Configurations**—that doesn't look right.

Note If you choose **Run As** and neither **Java Applet** nor **Java Application** options appear, click *in the Editor Window* to make sure Eclipse knows you're running Java, and try again.

SalesReport doesn't extend **Applet**, so it's not an Applet. And since **SalesReport** doesn't have a **main()** method, Java doesn't know what to do. But we created the **Main** class to instantiate and start this application, so let's go to that class.

Open the **Main** class. Now that we've defined SalesReport, **Main** can find it. Right-click in the **Main** class editor window and choose **Run As | Java Application**. You can also run it by clicking the ▶ icon in the Eclipse toolbar. Eclipse will attempt to run the class in the current editor window. If it is an Applet, it will run as an Applet; if it has a **main()** method, it will run as an application.

The *Console* is now open, eagerly awaiting your input:


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

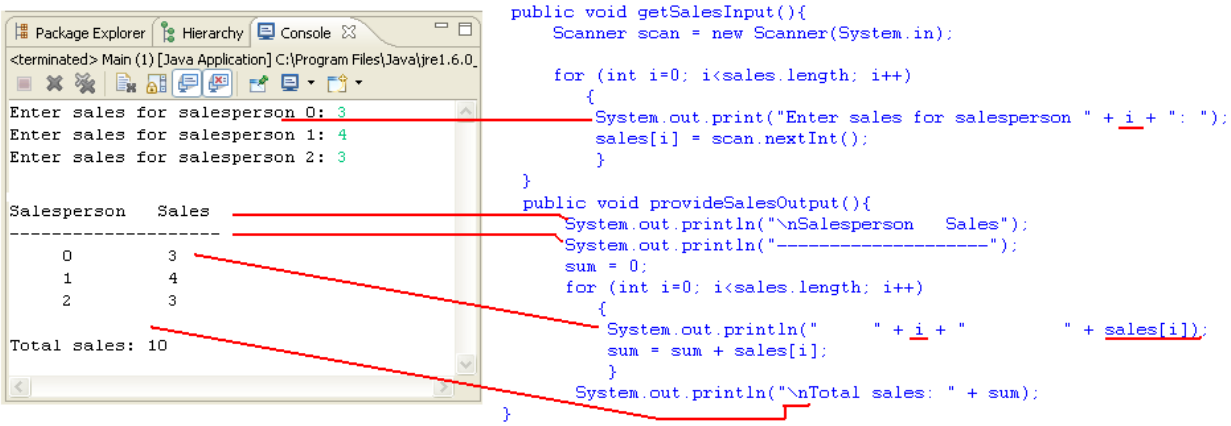
```
Enter sales for salesperson 0:
```

```java
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, enter a number, and click **Enter**. Now the Console opens and waits for you to provide input again. Enter another number. The console will ask for input three times, and will then provide the output.

Code Trace

In the figure below, a sample run is shown on the left and the corresponding code is displayed on the right. **System.out.print** lines are connected to their output by red lines:


Package Explorer
Hierarchy
Console
<terminated> Main (1) [Java Application] C:\Program Files\Java\jre1.6.0_

```
Enter sales for salesperson 0: 3
Enter sales for salesperson 1: 4
Enter sales for salesperson 2: 3


Salesperson   Sales
--------------------
     0          3
     1          4
     2          3


Total sales: 10
```

```java
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 + ": ");
        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 + "          " + sales[i]);
        sum = sum + sales[i];
        }
     System.out.println("\nTotal sales: " + sum);
}
```

The **sales[]** array was set to be the same size as **salespeople**. **salespeople** is set to 3, so **sales.length** will be 3, and so the **Scanner** asks for input 3 times (for **i** values of 0, 1, and 2).

The variable **sum** *accumulates* within the **for** loop: **sum = sum + sales[i];** to provide the sum of all of the sales values. The total is then printed, *outside* of the loop body.

We added **\n** to a couple of the **System.out.println** lines to insert a blank line between output lines.

Variable Setting by Users

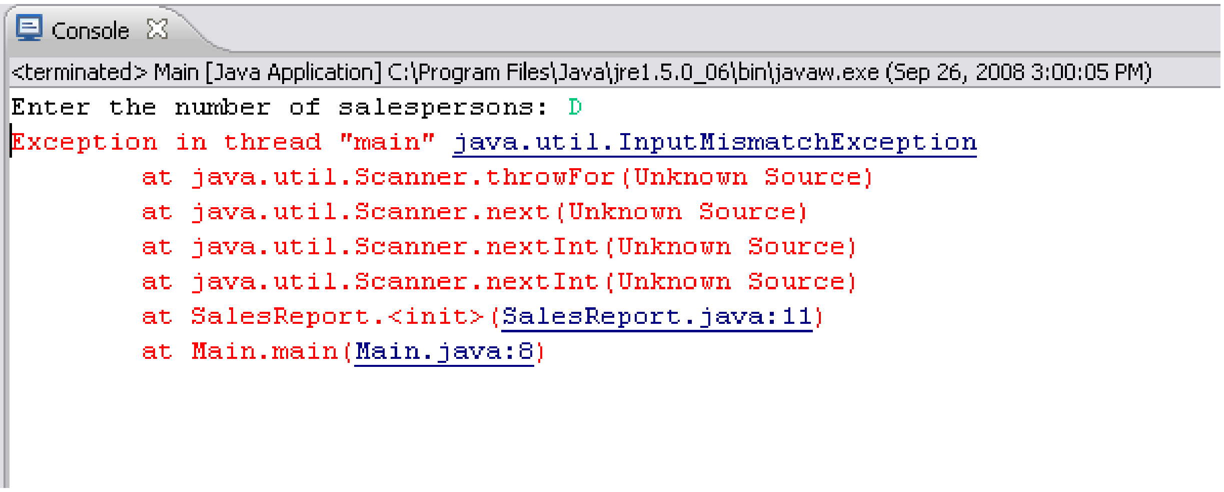
Using the constant **salespeople** is fine, but we still need to go into the code to change the variable, then save the file again—and users might not have access to the code. So let's incorporate another common practice that will allow the user to determine the number of **salespeople**.

Edit **SalesReport**. Add the code in **blue** and remove the code in **red** as shown:

CODE TO TYPE:

```
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 = 3 scan.nextInt(); //REMOVE the 3!!!
        this.sales = new int[salespeople];
    }

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

    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 + ": ");
            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 + "         " + sales[i]);
            sum = sum + sales[i];
        }
        System.out.println("\nTotal sales: " + sum);
    }
}
```

Save and run it from the **Main** class, for two salespeople. Then, run it again for four salespeople. Try entering a character rather than a number. You'll see an **Exception** in the Java Console:



```
<terminated> Main [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Sep 26, 2008 3:00:05 PM)
Enter the number of salespersons: D
Exception in thread "main" java.util.InputMismatchException
	at java.util.Scanner.throwFor(Unknown Source)
	at java.util.Scanner.next(Unknown Source)
	at java.util.Scanner.nextInt(Unknown Source)
	at java.util.Scanner.nextInt(Unknown Source)
	at SalesReport.<init>(SalesReport.java:11)
	at Main.main(Main.java:8)
```

We'll discuss **Exceptions** later in the course. For now, just run it once more using a number.

Now, take a look at the new code:

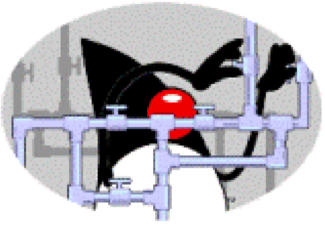
OBSERVE:

```
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 void getSalesInput(){
        for (int i=0; i < sales.length;  i++)
        {
            System.out.print("Enter sales for salesperson " + i + ": ");
            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 + "         " + sales[i]);
            sum = sum + sales[i];
        }
        System.out.println("\nTotal sales: " + sum);
    }
}
```

We changed the **Scanner** object, **scan**, into an instance variable, **removing it** from the **getSalesInput()** method as a local variable. This allows us to use the scan object in more than one method.

In the constructor, we ask the user to enter the number of salespeople and query the **scan** object for the next integer typed into the console, storing that value in the **salespeople** field.

In the next lesson, we'll continue to explore low-level design concepts using the **SalesReport** application that allows users to supply information. We want to make sure that our code flows freely and doesn't hold users back!



The User Experience

User Friendly Coding

In this lesson, we'll continue to work on our **SalesReport** application. We'll add features that make users happy by allowing them greater control over their experience! Always keep your users in mind when creating code.

User Input: Command Line

Let's try letting users enter initial start-up information at the *command line*. Although this technique is not as popular as it was pre-Web or pre-GUI days, it's still sometimes useful.

Edit **Main.java** from your java3_Lesson01 project as shown in **blue**:

CODE TO TYPE:

```
public class Main {

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

Now, check out the changes we made:

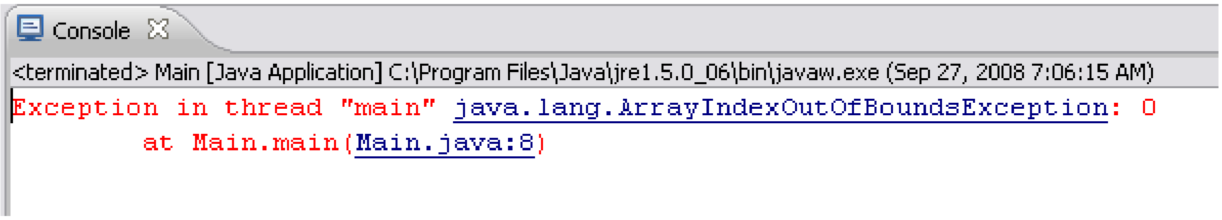
OBSERVE: Adding Command Line Arguments to Main

```
public class Main{

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

Some applications might not have a graphical user interface for user input, so Java requires that we declare the main method specifically *with arguments*, as **public static void main(String[] args)**.

Java uses the formal parameter **String[] args** to allow the user to provide input. The input provided is cast as a **String** and put into the array **args**. The **main ()** method in the class called receives these arguments, so the programmer has to access the **args** array in order to get those values. Specifically, **args[0]** (in the main method) are made up of whatever is entered for the first argument. The argument variable type is **String**, so if the user enters 5 for the first argument, **args[0]** will be equal to the String "5" (not the numeral). In our example, we check to see if **args[0]** is null. We generally assume that if there are no command line arguments, the 0 element of the array is null.

Save and run it.



```
<terminated> Main [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Sep 27, 2008 7:06:15 AM)
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
	at Main.main(Main.java:8)
```

The condition for the **if** is **(args[0] != null)**, so Java wants to go to **args[0]** immediately. But since *no arguments* were given, **args[0]** doesn't exist. In fact, even asking for **args[0]** is out of bounds. So, what should we do? Consider checking the length of **args** instead.

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

CODE TO TYPE:

```
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();
            mySalesInfo.testMe();
        }
    }
}
```

OBSERVE: Fixing Our Mistake

```
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();
            mySalesInfo.testMe();
        }
    }
}
```

Now, we check to see if the **length** of the **args** array is greater than 0. If it is, we set the local variable **argIn** to the int value of the **args**[0] element. Then we create a **SalesReport** object named **mySalesInfo** by passing **argIn** to the constructor of the new **SalesReport** object. This will produce an error because we have not yet modified the **SalesReport** class. Next, we call the **testMe()** method of the **SalesReport** class.

If the **args** array length is not greater than 0, we create our **SalesReport** class just like we did earlier. Then we call the **testMe()** method of the **SalesReport** class.

Go ahead and save this program.

new SalesReport(argIn) is underlined in red, because we haven't defined this additional constructor with a parameter (**SalesReport(int x)**) yet. Java/Eclipse offers some suggestions, but ignore them for now.

Here we give the user an opportunity to enter the number of salespeople before the code starts. If the user doesn't enter a number here, we'll prompt her to do so when the **SalesReport** class starts.

If the user does input a number, we want to pass it to the **SalesReport** class through its Constructor, so we'll need a new Constructor with the formal parameter of an **int** before we can test our new **Main**. That's why you see Java's comment: **Create constructor 'SalesReport(int)'**. If the user *doesn't* enter a number, we'll use our recently edited Constructor to *prompt* for it.

Overloading the Constructor

Let's write that Constructor, this time one that receives the number of **salespeople**. In object-oriented terms, we'll *overload* the Constructor method. Overloading occurs when a Class has two methods with the same name, but different *signatures* (numbers and/or types of parameters).

Edit **SalesReport** in your java3_Lesson01 project, as shown in **blue**:

CODE TO TYPE: SalesReport

```
import java.util.Scanner;

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

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

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

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

    public void getSalesInput(){
        for (int i=0; i < sales.length;  i++){
            System.out.print("Enter sales for salesperson " + i + ": ");
            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 + "         " + sales[i]);
            sum = sum + sales[i];
        }
        System.out.println("\nTotal sales: " + sum);
    }
}
```

Save it, then go back to the **Main** class. The error is gone now.

Run the application in the Main class. Since we didn't run it with input from the command line, Java prompts us for this information.

OBSERVE: Overloaded Constructor

```
import java.util.Scanner;

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

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

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

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

    public void getSalesInput(){
        for (int i=0; i < sales.length;  i++){
             System.out.print("Enter sales for salesperson " + i + ": ");
             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 + "         " + sales[i]);
            sum = sum + sales[i];
        }
        System.out.println("\nTotal sales: " + sum);
    }
}
```

This constructor lets us pass a parameter to the SalesReport class when we create it. The no-parameter constructor works as it did before, allowing the user to select the number of salespeople at run-time.

A Closer Look at Main

Take a look at the line we wrote in order to get the argument to pass to the **SalesReport** constructor, **SalesReport(int howMany)**:

OBSERVE: main args

```
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();
            mySalesInfo.testMe();
        }
    }
}
```

If we allow the user to specify an argument initially, and **(args.length > 0)** is true, what parameter is passed to the constructor of **SalesReport**?

The code for the Constructor seems reasonable. **argIn** is declared as an **int**, so passing it meets the formal parameter requirements for the Constructor **SalesReport(int howMany)** in the **SalesReport** class. But what is this **Integer.parseInt(args[0])**?

The variable in the **main()** is **(String [ ] args)**, so everything given to the method as an **args[]** is cast to a **String**. And because Java would interpret, for example, 5 as **String** "5" and not an **int**, we need to make **String** into an **int**.

API Go to the API to see the **java.lang.Integer** class, then consider the two snippets of code below (keep in mind that **args[0]** is a **String**):

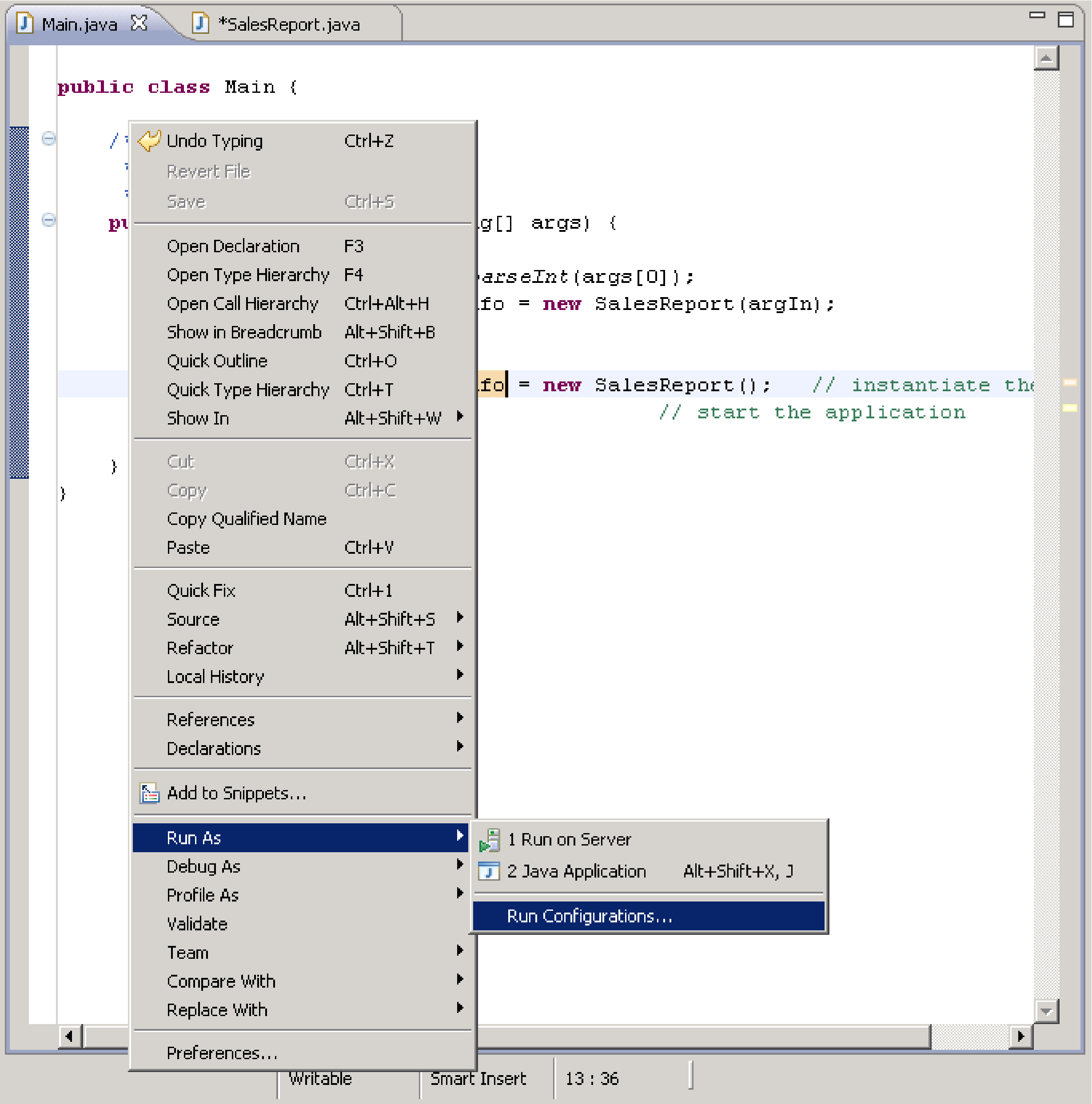
- Integer.parseInt(args[0])
- Integer.valueOf(args[0].intValue())

Ultimately, both techniques take a **String** argument and return an **int** value for it. Similar methods are used in other wrapper classes in **java.lang** (also called *convenience classes*) to parse from Strings to other primitive data types. These methods are particularly useful, because they allow Java to have a measure of conformity. Because Java interprets *all* input arguments as **String**s, and is unable to anticipate user input, it needs *some* conformity. But as a programmer, you know what to expect as input, and can convert the passed **String** to any type you want. In our code, we have a constructor that receives an **int**, so we convert that passed **String** argument to an **int**.

Entering Command Line Arguments

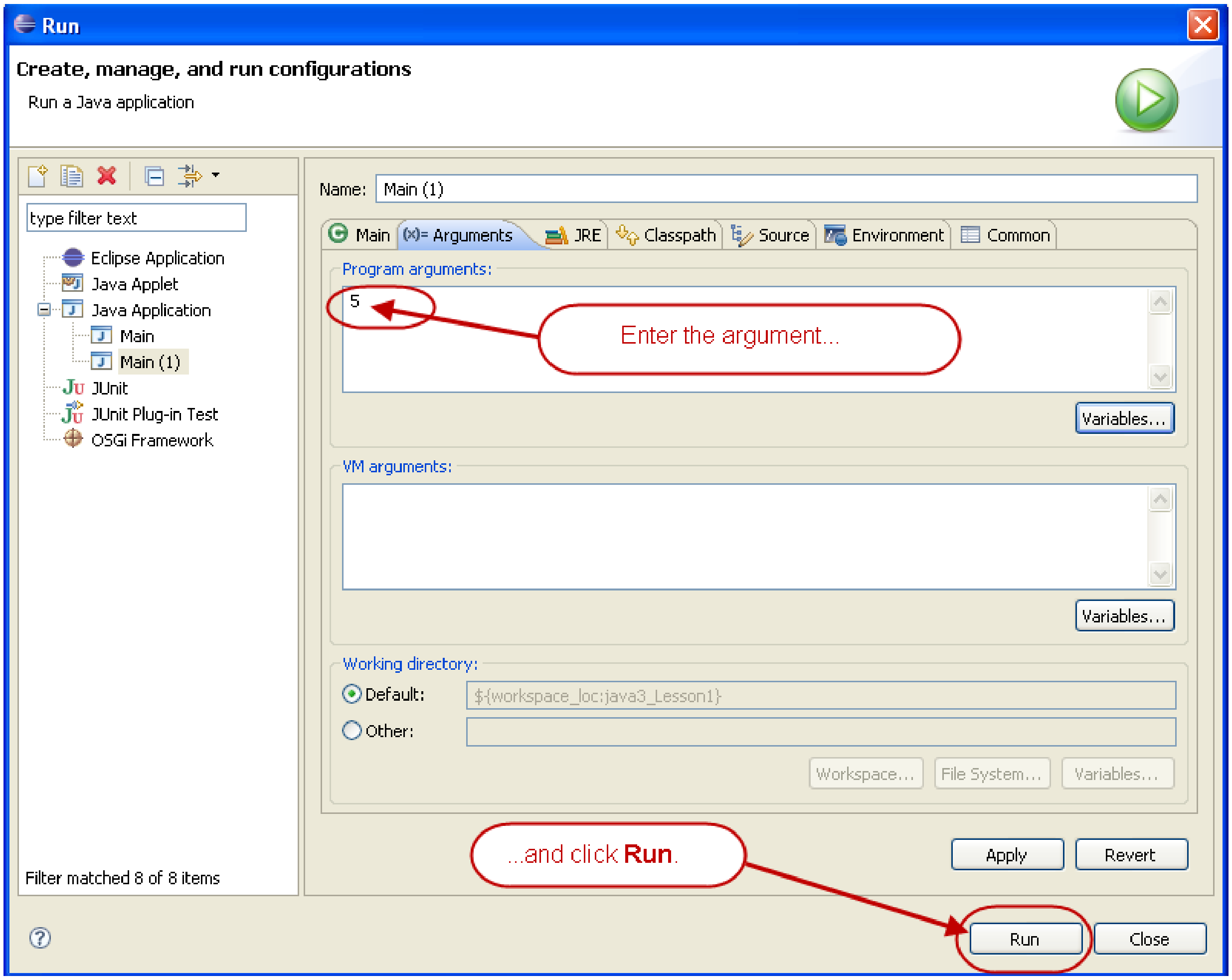
So, how do we allow users to give arguments from the command line? Let's go over that from within Eclipse. (Later, we'll show you how to do it when you're not in Eclipse.)

If you haven't done so already, save your **Main** class. Right-click it and select **Run As | Run Configurations**:



```
public class Main {

    /*
     *
     *
    pu                                   g[] args) {

                                        arseInt(args[0]);
                                        fo = new SalesReport(argIn);

                                        fo = new SalesReport();    // instantiate the
                                                       // start the application

    }
}
```

Select the **Arguments** tab and enter **5** in the **Program arguments:** box. Then click **Run**:



Name: Main (1)

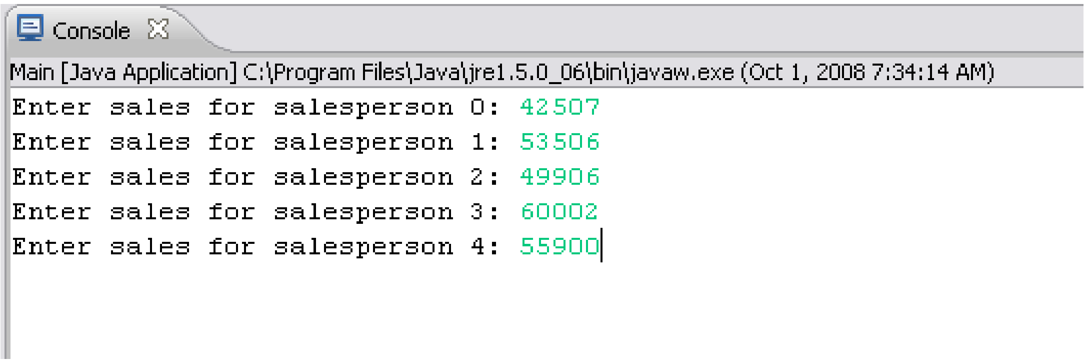
Program arguments: 5

VM arguments:

Working directory:
● Default: ${workspace_loc:java3_Lesson1}
○ Other:

Enter the argument...

...and click **Run**.



Main [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Oct 1, 2008 7:34:14 AM)

```
Enter sales for salesperson 0: 42507
Enter sales for salesperson 1: 53506
Enter sales for salesperson 2: 49906
Enter sales for salesperson 3: 60002
Enter sales for salesperson 4: 55900
```

There's a prompt in the Console to **Enter sales for salesperson:** for the number of times that you specified. Now users can enter any number of salespeople they want, and we won't have to edit the code. Users can provide their input either when they **Run**, or when prompted by the program. Sweet!

Run the program again with **Run As | Java Application** only. It still prompts for 5 salespeople. That's because we didn't remove the "5" argument. It's still set for 5 in Eclipse.

Choose **Run As | Run Configurations**, select the **Arguments** tab, and delete the number you set earlier. Now, **Run As | Java Application** again; it should prompt you as expected.

Code Flexibility Revisited

Fooling the User

Being diplomatic managers, we don't want to identify anyone on our staff as "Salesperson 0," but 0 is assigned automatically as Java's ID for the first index in the array. Fortunately, programmers have power. While we can't change the way Java is written, we can change the way it looks to the user. And we can do it without going to all the trouble of manipulating programming structures, such as arrays. Instead, we can make Java *print* a higher number to the user. Salesperson 1 is still identified with the array element sales[0], but he doesn't need to know that!

Let's give it a try. Edit **SalesReport** as shown in **blue**:

CODE TO TYPE:

```
import java.util.Scanner;

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

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

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

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

    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);
    }
}
```

Save and run it (as a Java Application from Main).

Tip It's often easier to change the appearance of our output by changing *print* statements than by changing programming structures.

Finding the Max

Suppose we want to find the maximum sales and the salesperson responsible for that number. We can do that!

Edit **SalesReport** as shown in **blue**:

CODE TO TYPE:

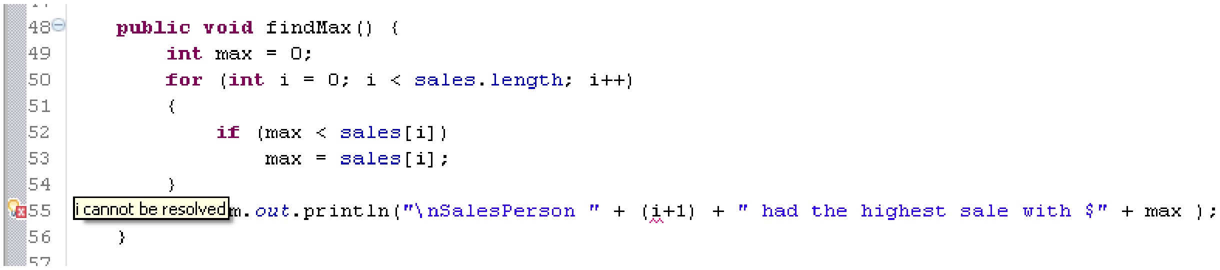
```
import java.util.Scanner;

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

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

    public SalesReport(){
        System.out.print("Enter the number of salespersons: ");
        this.salespeople = scan.nextInt();
        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 = 0;
        for (int i=0; i < sales.length;  i++)
        {
            if (max < sales[i])
               max = sales[i];
        }
        System.out.println("\nMaximum sales is " + max);
    }
}
```

Save and run it (as a Java Application from Main). The program works fine, but who has the *maximum* (largest) sale?

Edit **SalesReport** by changing the **findMax ()** method as shown in **blue**:

CODE TO TYPE:

```java
import java.util.Scanner;

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

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

    public SalesReport(){
        System.out.print("Enter the number of salespersons: ");
        this.salespeople = scan.nextInt();
        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 = 0;
        for (int i=0; i < sales.length;  i++)
        {
            if (max < sales[i])
            max = sales[i];
        }
        System.out.println("\nSalesperson " + (i+1) + " had the highest sale with $" + max);
    }
}
```



```
    public void findMax() {
        int max = 0;
        for (int i = 0; i < sales.length; i++)
        {
            if (max < sales[i])
                max = sales[i];
        }
        i cannot be resolvedm.out.println("\nSalesPerson " + (i+1) + " had the highest sale with $" + max );
    }
```

The **i** is underlined and Java tells us it **cannot be resolved**. That's because **i** is declared *in* the loop initialization, so its scope is only within the loop. Not to worry, I'm confident we can fix this! Let's try moving **i** out of the loop. Edit **SalesReport** by changing the **findMax()** method as shown in **blue**:

CODE TO TYPE:

```
import java.util.Scanner;

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

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

    public SalesReport(){
        System.out.print("Enter the number of salespersons: ");
        this.salespeople = scan.nextInt();
        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 = 0;
        int i;
        for (i=0; i < sales.length;  i++)
        {
            if (max < sales[i])
            max = sales[i];
        }
        System.out.println("\nSalesperson " + (i+1) + " had the highest sale wit
h $" + max );
    }
}
```

Save and run it (as a Java Application from Main).

Give it 3 salespeople with sales amounts of 3, 4, and 5:

OBSERVE: Output from SalesReport Application

```
Enter the number of salespersons: 3
Enter sales for salesperson 1: 3
Enter sales for salesperson 2: 4
Enter sales for salesperson 3: 5

Salesperson   Sales
--------------------
     1         3
     2         4
     3         5

Total sales: 12

Salesperson 4 had the highest sales with $5
```

Hmm. That's better, but there's still a problem. Who is this Salesperson 4 and why are they outselling our other salespeople?!

The real problem isn't *where* we declare the loop variable, but *when* the loop is done. The last loop iteration might not always be where **max** was set. When we come out of the loop, **i** will **always** be the last value Java saw in the loop. So, we need to remember **who** (which loop index) has the maximum sale by putting it in memory (that is, giving it a variable name and in doing so, a memory location).

Edit **SalesReport** as shown in **blue**:

CODE TO TYPE:

```
import java.util.Scanner;

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

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

    public SalesReport(){
        System.out.print("Enter the number of salespersons: ");
        this.salespeople = scan.nextInt();
        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 = 0;
        int who = 0;
        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 w
ith $" + max );
    }
}
```

Save and run it (as a Java Application from Main). We are still "fooling" the user by adding 1 to the index **who**. (Nobody wants to be salesperson 0, right?)

Don't Let the User Fool You

Now, let's suppose our salespeople are having a bad year and they all *lost* money. Run the program (from Main) and enter negative sales numbers for everyone. This gives 0 as the output for the max sold, even though nobody actually sold 0.

Edit **SalesReport** as shown in **blue**:

CODE TO TYPE:

```
import java.util.Scanner;

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

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

    public SalesReport(){
        System.out.print("Enter the number of salespersons: ");
        this.salespeople = scan.nextInt();
        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];
        int who = 0;
        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 w
ith $" + max );
    }
}
```

Save and run it (as a Java Application from Main), again with all negative sales numbers.

It's not really fair to blame the user for such weird numbers. Something like this *could* really happen. Java provides tools for its programmers to handle all kinds of errors. In later lessons, we'll look explicitly at **Exception** and **Error** **classes** that will assist us in dealing with strange input.

What's in store?

Now we're ready to look at other Java capabilities Java and get even cozier with the API. We'll start by digging into the top level of the API—packages—and from there, we'll explore each structure (including classes, interfaces, exceptions, and enumerations). Soon, you'll embrace Java and the API will become your best friend! See you in the next lesson...

Packages

You Have Great Potential

Java provides lots of pre-written classes that we can access through a convenient class library known as the Application Programming Interface (API).

The more familiar you are with the API, the better Java programmer you'll be. The Java API is huge, and as far as I know, nobody has it memorized. But if we get to know its organizational structure and resources well, we'll be able to wade through it faster and be much more productive. Because the API is essential to efficient Java programming, we always provide a link to the newest version of the API in your Eclipse menus.

API To view the API, click the API icon under the Eclipse menu bar.

Why Packages?

Organization

Good resources have an organizational pattern that allows users to search them efficiently. Java uses an organizational tool called **packages** to group together **classes** and **interfaces** that are related to each other, and this in turn enables modular groups.

You're probably familiar with the concept of folders (or directories) for files on computers. Typically, we organize our folders by putting related files into folders with appropriate names. Java's **package** concept is similar, but instead of folders, Java provides thousands of classes. It uses the namespace (container) of **packages** to organize related classes and interfaces into meaningful collections. The **packages** hold classes and interfaces (*compilation units*), which have been created to assist programmers with common tasks associated with general-purpose programming. Because these classes are already available, we can focus on designing our own applications, and avoid doing all of our programming from scratch.

Inside Packages

The members of packages are:

- subpackages
- top-level Interfaces declared in the package
- top-level Classes declared in the package (note that **Exceptions** and **Errors** are **Classes**)
- Enumerations and Annotation Types (which are also special kinds of classes and interfaces)

Java provides more detailed information about packages and other specifications in the Java Language Specification.

Packages are often organized using subpackages. You can read more about subpackages and their hierarchies, as well as other Java tools, in the Java Tutorial.

Related classes and interfaces (**compilation units**) are grouped together and declared with the same package name. Packages and their subpackages are separated by a dot. The examples listed below have *fully qualified names*:

Sample Java subpackages:

- **java.awt** (contains interfaces such as: **ActiveEvent**, **ItemSelectable**; contains classes such as: **Button**, **Canvas**, **Color**, **Frame**, **Graphics**, **Image**, and **Window**)
- **java.applet**
- **java.io**
- **java.lang**
- **java.beans**
- **java.util**

Sample subpackages of **java.awt**:

- **java.awt.color** (contains classes such as: **ColorSpace** and **ICC_ColorSpace**)
- **java.awt.event** (contains interfaces such as: **ActionListener**, **AdjustmentListener**, **MouseListener**; contains classes such as: **ActionEvent**, **AdjustmentEvent**, **MouseEvent**)
- **java.awt.image** (contains interfaces such as: **ImageConsumer**, **ImageProducer**; contains classes such as: **BufferedImage**, **ImageFilter**, **PixelGrabber**,**ImagingOpException**)

API Open the API browser by clicking on the browser tab or the API menu icon. If this doesn't open the list of packages, click the browser's back button until you get there. You'll see a list of the subpackages of **java** (java.applet, java.awt, java.beans, and so on.). Scroll up to the subpackages of **java.awt** (there are quite a few: java.awt.color, java.awt.datatransfer, java.awt.event, and java.awt.font). Scroll all the way down; the only items in the Packages listing are **packages** (there are no compilation units).

Scroll back up and click on the **java.awt** package. There are no **packages** inside of its listing (there are *only* compilation units).

Inside any given package, we see:

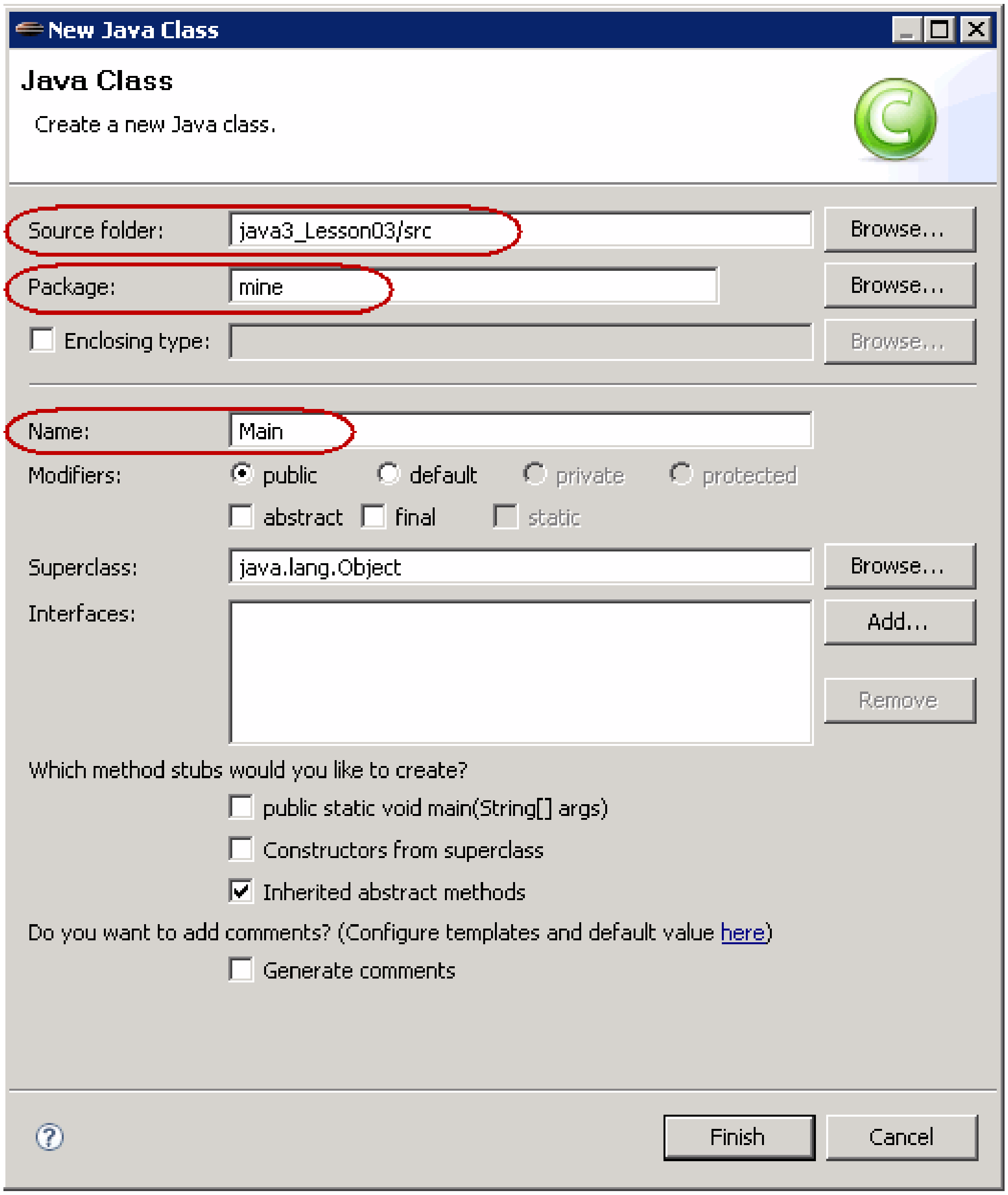
- Interface Summary
- Class Summary
- Enum Summary
- Exception Summary
- Error Summary

We'll go over each of these **compilation units** later in the course.

Your Own Java Package

Package Creation

Create a new **java3_Lesson03** project in the **java3_Lessons** working set. In this project, create a new Class, using the package **mine** and the name **Main**:


New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson03/src

Package: mine

□ 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

OBSERVE: Main.java

```
package mine;

public class Main {

}
```

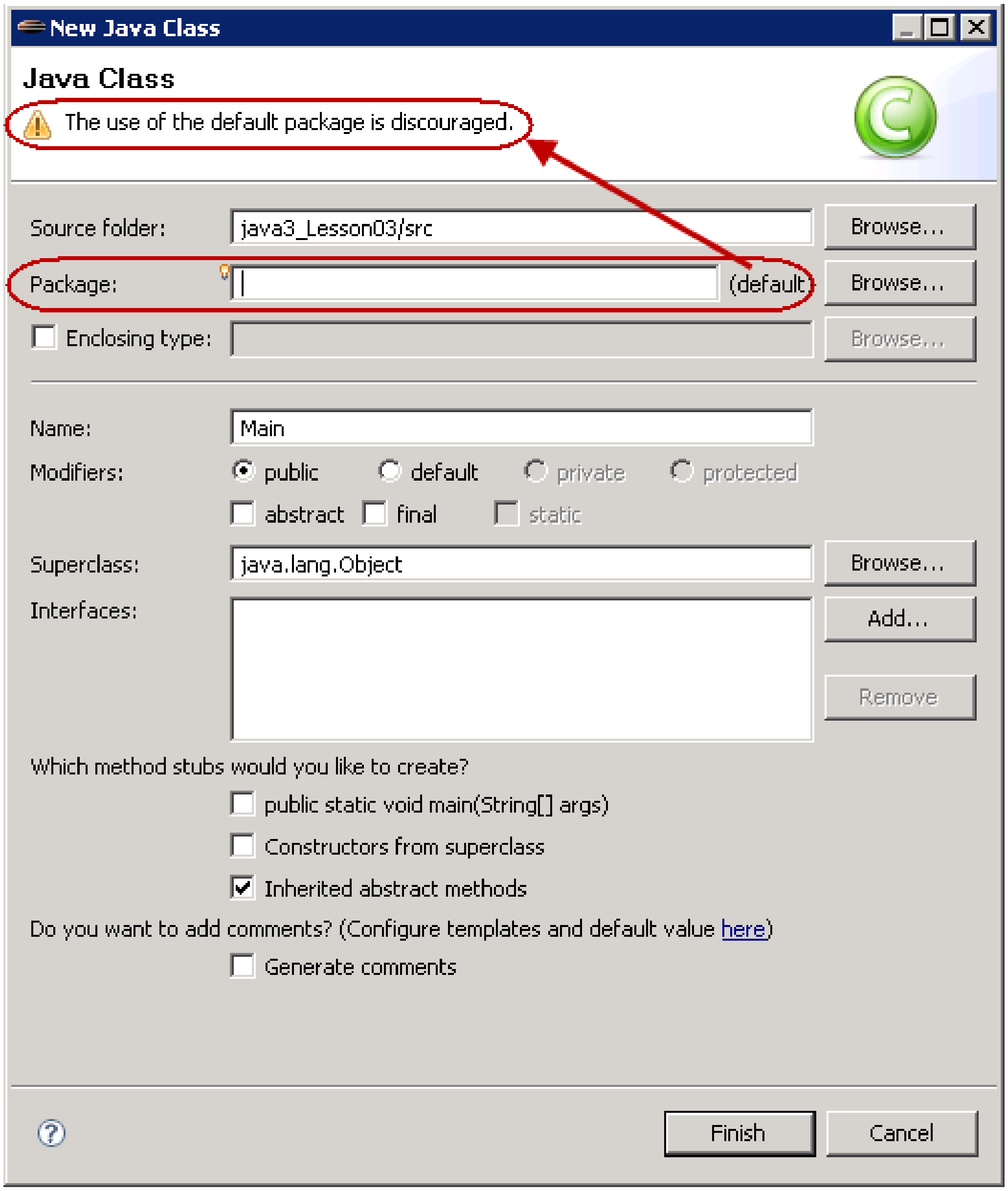
The first line of the new **Main.java** file is **package mine;**.

In general, to create a package, you put a **package** statement with its chosen name at the top of *every source file* that contains the types (classes, interfaces, and enumerations) that you want to include in the package.

Declaring a Package

If a package declaration statement appears in a Java source file, it must be the first item in that file (with the exceptions of comments and white space).

Because we did not specify package names in our earlier lessons, we used the *default package*. An unnamed (default) package should only be used for small or temporary applications, or at the beginning of the development process. You may remember Eclipse trying to talk you out of using unnamed packages earlier:



New Java Class

Java Class

The use of the default package is discouraged.

Source folder: java3_Lesson03/src

Package: (default)

□ 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

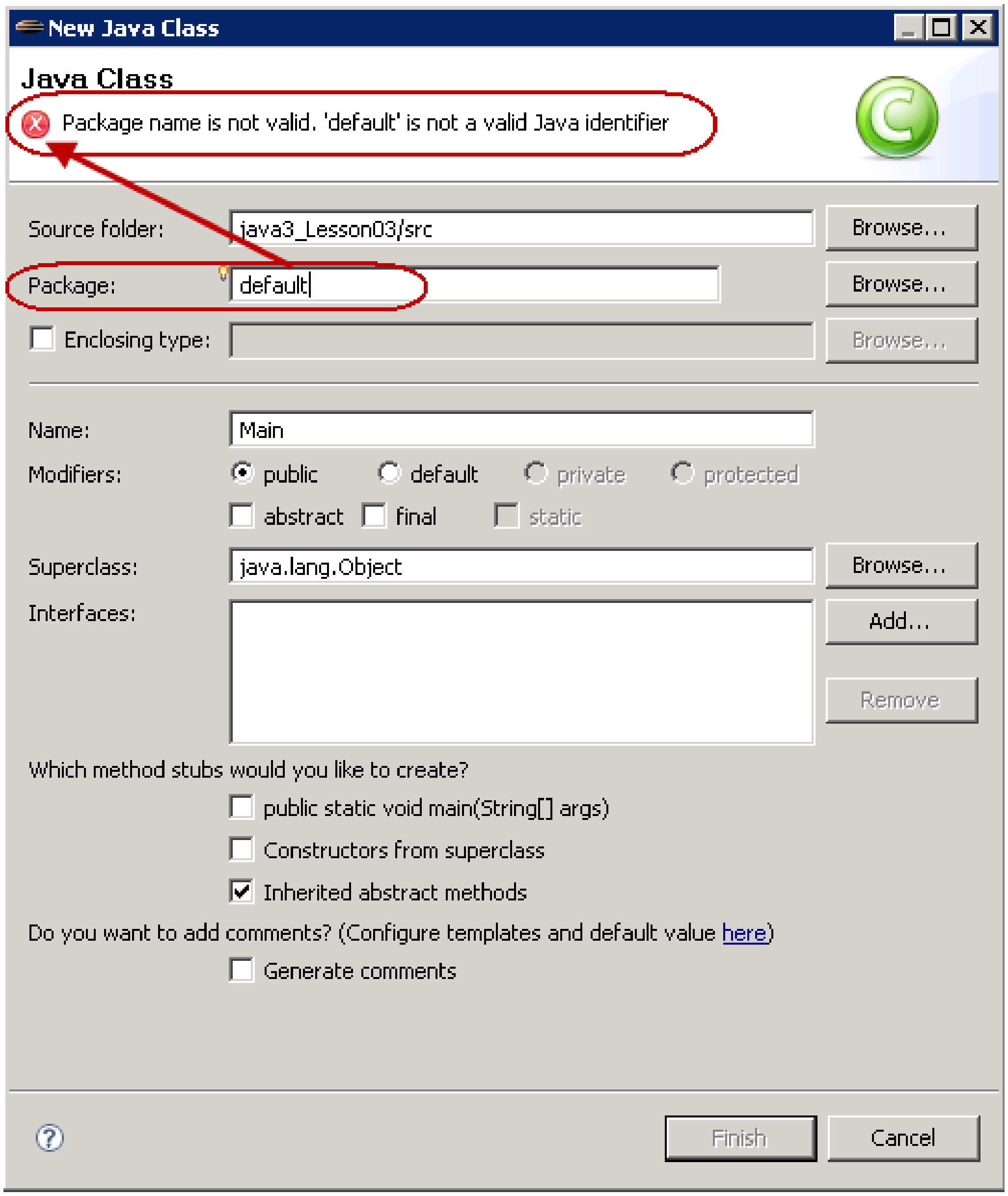
Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Note If no package name is specified at the beginning of a class, that class will be located in the default package. If no package names are specified, all class files in the same directory (or folder) will be in the same package, the default package. If you're working outside of Eclipse, the default package will be located in the current directory (that is, the directory where the class is defined).

In Lesson 1, we didn't specify a package name for our classes, so the classes were located in the default package. The **(default package)** you see in the Package Explorer appears in parentheses because it's not *really* a package named **default**.

Let's see if we can create a **default** package now in java3_Lesson03 and also create a new Class. In the Package field of the Create New Class window, enter **default**:



Java Class

Package name is not valid. 'default' is not a valid Java identifier

Source folder: java3_Lesson03/src

Package: default

☐ 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

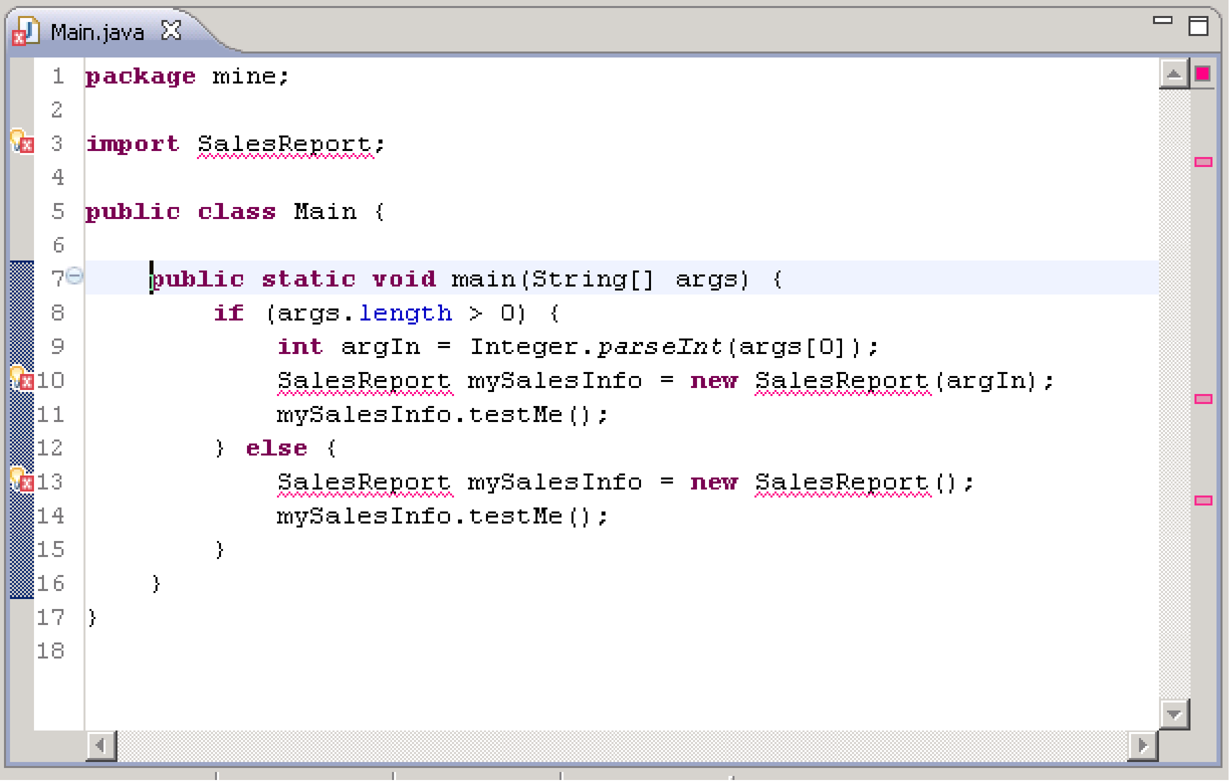
☑ Inherited abstract methods

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

☐ Generate comments

Oh well, we tried. Click **Cancel** to get out.

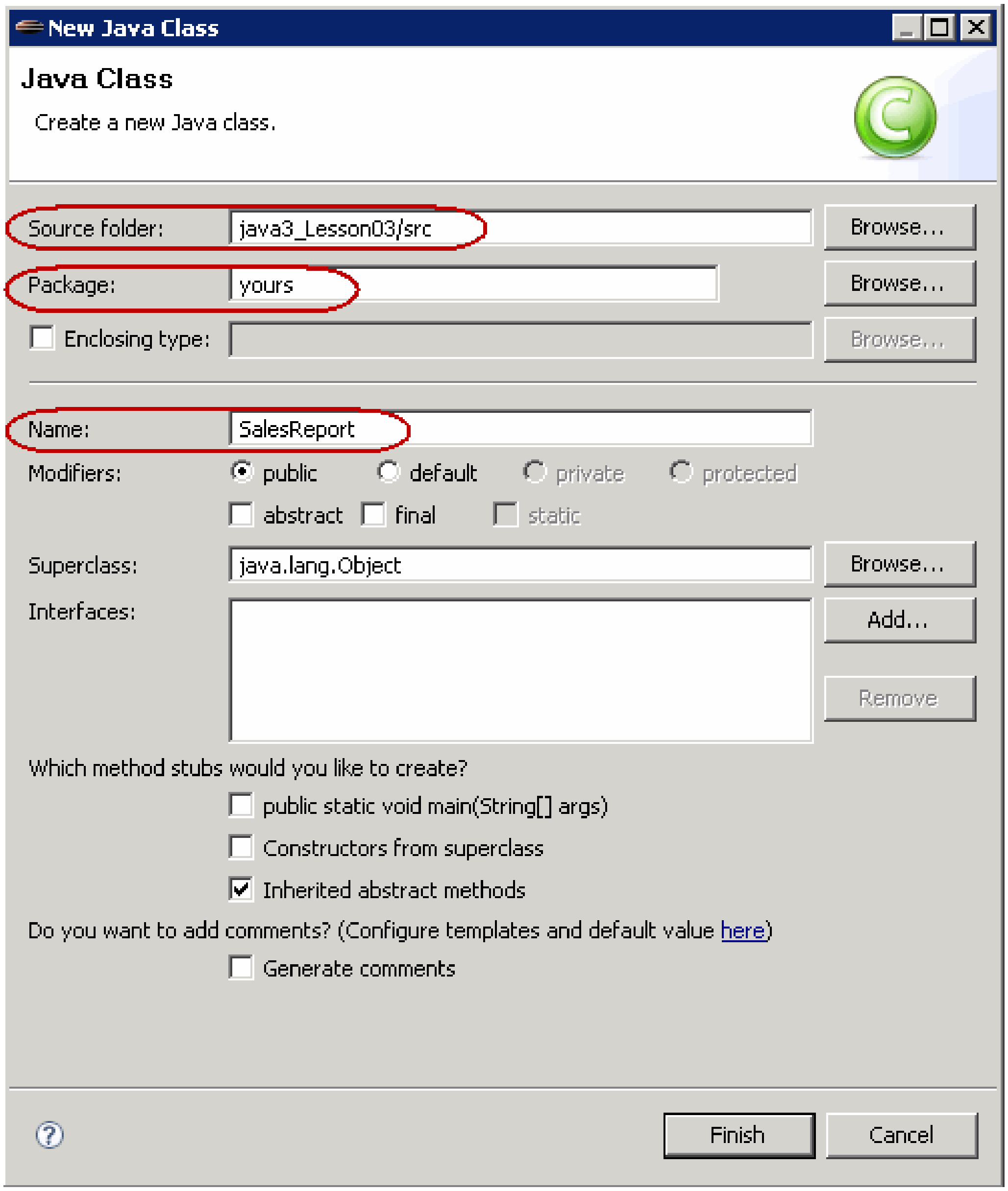
Let's try to copy the files we made in Lesson 1 over to Lesson 3 and give them package names to see how to use packages in code. Go to the folder **java3_Lesson01/src/(default package)**. Open the **Main.java** file. Highlight all of its contents, right-click the mouse, and select **Copy**. Close this Main.java file and open the **Main.java** we just created in **java3_Lesson03/src/mine**. **Keeping the package specification at the top**, right-click and select **Paste** underneath. Notice it also automatically added **import SalesReport;**. The code looks like this:



```
package mine;

import SalesReport;

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();
            mySalesInfo.testMe();
        }
    }
}

```

We could set it up to import the **SalesReport** class from our java3_Lesson01 project, but for now, we'll illustrate different packages using the same Eclipse project.

In the java3_Lesson03 project, create a new **SalesReport Class** as shown:



Java Class

Create a new Java class.

Source folder: java3_Lesson03/src

Package: yours

□ 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)

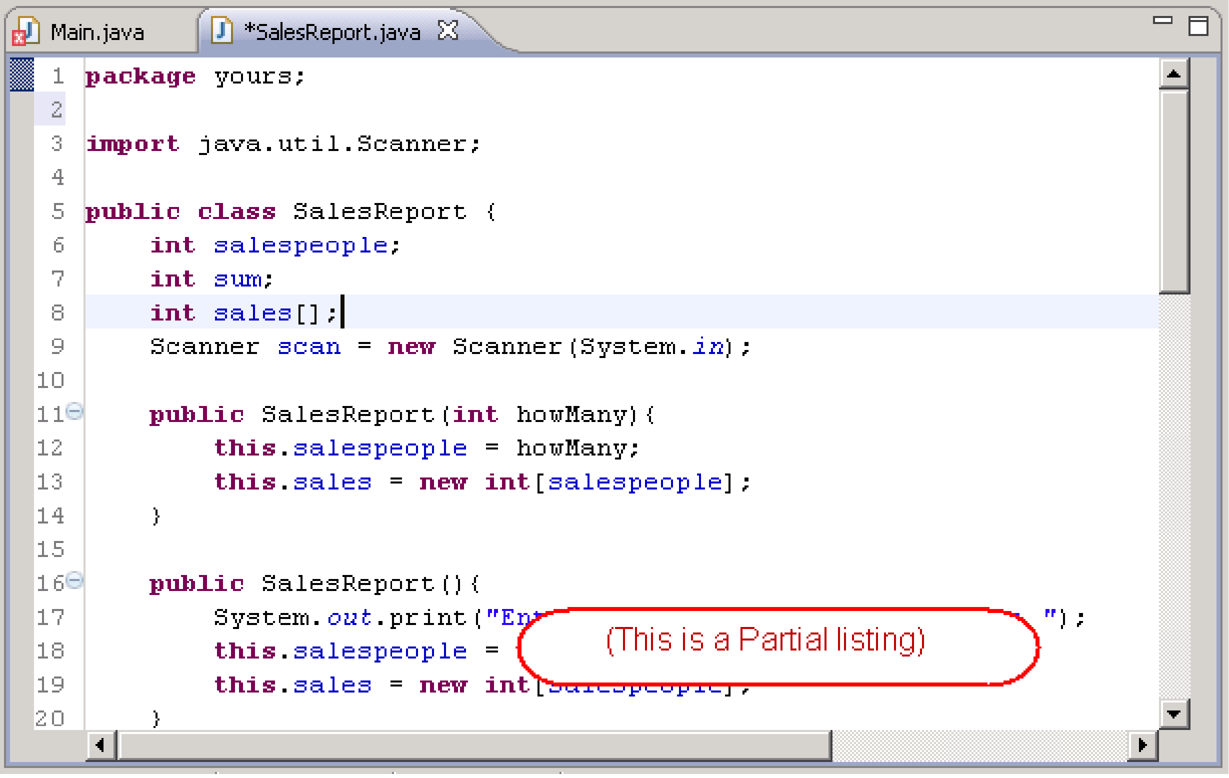
□ Constructors from superclass

☑ Inherited abstract methods

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

□ Generate comments

Now we'll copy from lesson 1. Go to **java3_Lesson01/src/(default package)** and open **SalesReport.java**. Highlight all of its contents and right-click to choose **Copy**. Close this file and open the **SalesReport.java** we just created in java3_Lesson03. **Keep the package specification** and paste the SalesReport code just below it. The code should look like this:



```
package yours;

import java.util.Scanner;

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

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

    public SalesReport(){
        System.out.print("En                        ");
        this.salespeople = 
        this.sales = new int[salespeople];
    }
```

(This is a Partial listing)

Package Access

In order to *use* classes and interfaces located inside packages (other than **java.lang** or the package that contains the class), we must tell the program where they're located.

Accessing Packages Using import

We direct programs to the location of packages that hold classes and interfaces using **import** statements. We already have our classes set up in different packages (**yours** and **mine**). Now, we will let one access the other. Since they are no longer in the *same package* (they were both in *default* in Lesson 1), we need to import the class with its fully qualified name so that Java knows where to find it.

Go back to the newly created Lesson 3 **Main.java** class and modify the code as shown in **blue**:

CODE TO TYPE: mine.Main

```
package mine;

import yours.SalesReport;

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...
            mySalesInfo.testMe(); //Start it
        }
    }
}
```

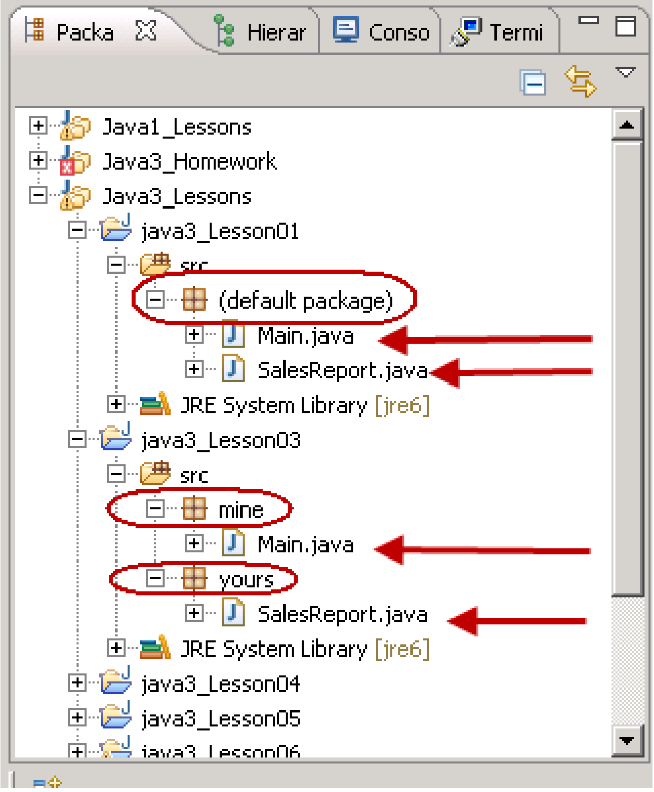
The fully qualified name now points to the class and its location (the package **yours**). This corrects all of the previous errors.

Save the **SalesReport.java** and **Main.java** files.

Run it (from Main). It should work the same way it did before.

The Classloader

Okay, now let's look at the Eclipse Package Explorer directory's structure, to reinforce our understanding and appreciate its coolness:



In the same way that *programmers* use the package organization of the API to look up information about classes, *programs* use the structure (through the package namespace) to *access* the code for the classes that we instruct it to use. Specifically, Java transforms a package name into a *path name* by concatenating the components of the package name and placing a file name separator between adjacent components.

For example, on a UNIX system, where the file name separator is */*:

The package name **oreilly.school.java.courses** would become the directory name **oreilly/school/java/courses/**.

In Windows, where the file name separator is \:

The package name **oreilly.school.java.courses** would become the directory name **oreilly\school\java\courses\**.

Classloading can be a bit complicated, but don't worry. We address it in greater detail later in the Java series. Until then, here's a Java World article if you're interested in looking into the basics of Java class loaders right now.

When to Import

The classes in the **java** package are available in any Java implementation. They are the only classes guaranteed to be available across different platforms and Java versions. Classes in other packages (Oracle, Netscape) may be available only for specific implementations.

Newer versions of Java include additional packages that used to be *plugins* (for example, **javax.swing** and **org.omg.CORBA**).

API Go to the Packages API page and scroll to the bottom. There are lists of packages and subpackages available under **javax**, **org.omg**, and many other links you'll find in the API.

Although all of the classes in the **java** package are available *by default*, *your* Java classes have access only to the classes in your current package (directory) and in the package **java.lang**. To use classes from any other package, you have to execute one of these actions:

- Refer to them explicitly by package name. For example, **java.util.Date today = new java.util.Date();**
- Import them to your source file. For example, **import java.util.*;**, then use **Date today = new Date();**. (This is usually the preferred method, because it requires less code to be written when more than one class is being used in the package.)

You can only use the * wildcard to import multiple **classes** from a specific package. You cannot use * to import multiple packages. And **import java.*;** won't import classes from multiple subpackages of **java**. Using the * (wildcard) to import all classes for a given package has no negative impact on compile time or code size, so go ahead and work it.

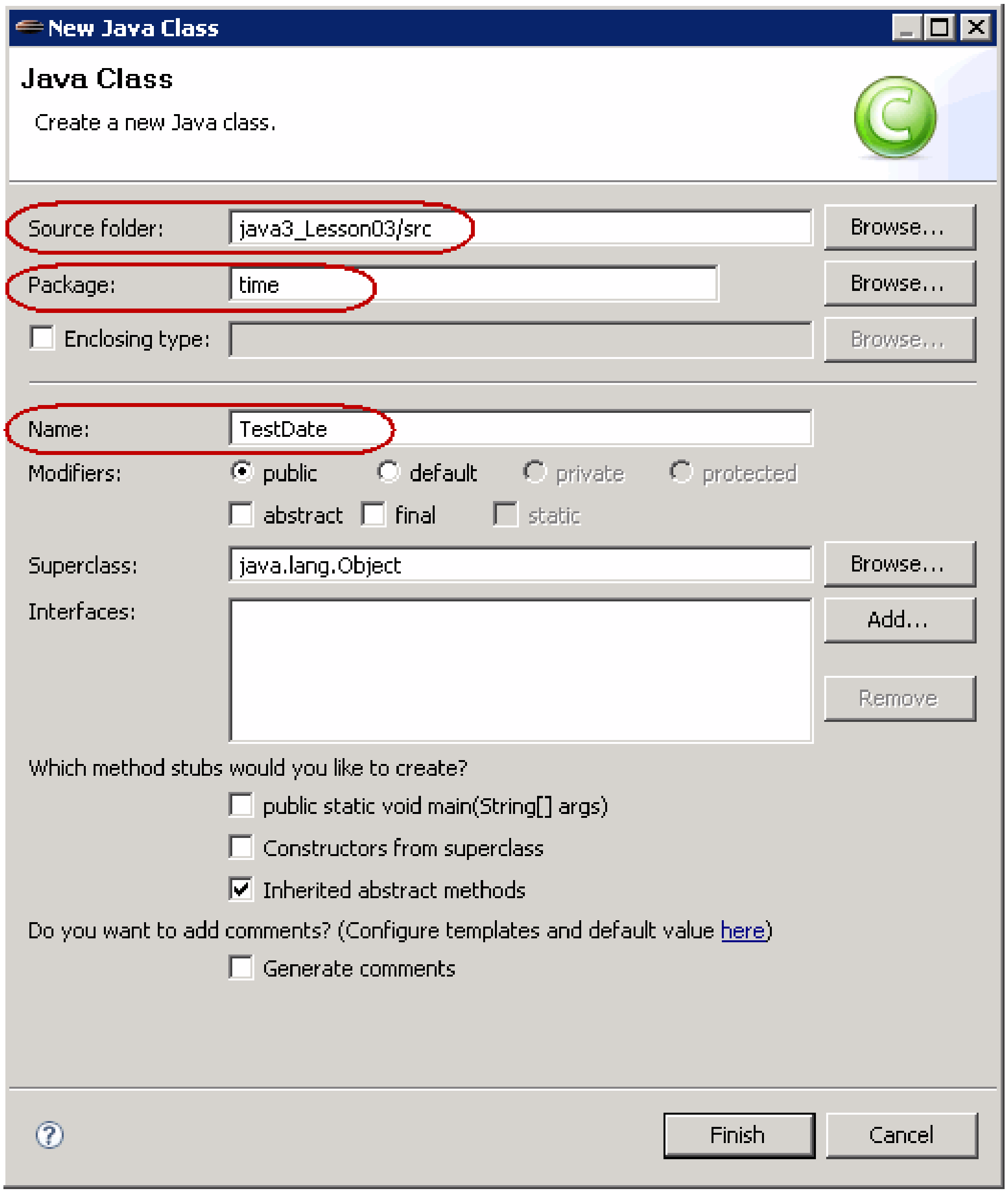
Note **import** does not work the same way as **#include** does in C. Java uses dynamic class loading—that is, it only loads classes when they are actually instantiated.

Naming Conflicts

Nobody in their right mind wants to search the entire API to find out if a class name has already been used. Fortunately, modularity and polymorphism allow you to replace existing class names with names tailored to fit your class and your package.

This could result in multiple packages with classes that share the same name, but that's not a problem. Because of Java's inherent modularity, we can name classes in packages whatever we like. Modularity allows you to specify exactly which class you want to use when there is more than one package with the same class name. If two packages have classes with the same name, Java just won't let you import them both.

Consider, for example, the class **Date**. In the java3_Lesson03 project, create a new **TestDate** Class:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson03/src

Package: time

□ Enclosing type:

Name: TestDate

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)

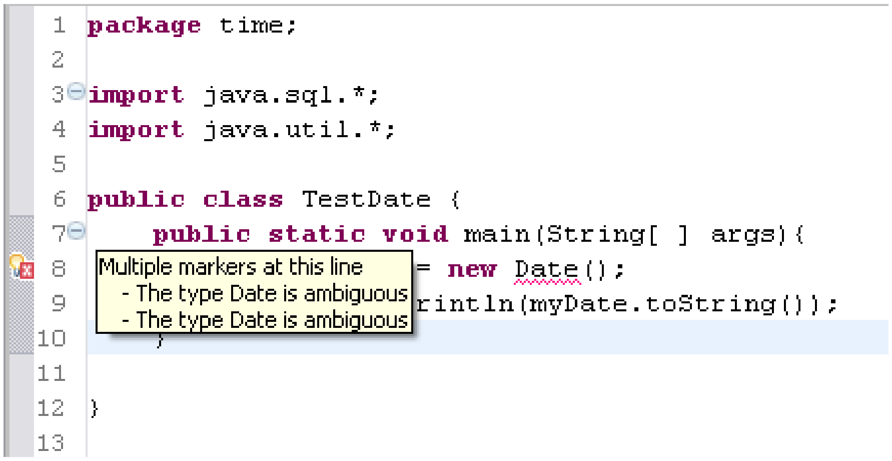
□ Generate comments

In **TestDate**, type the **blue** code as shown:

CODE TO TYPE:

```
package time;

import java.sql.*;
import java.util.*;

public class TestDate {

    public static void main(String[ ] args){
        Date myDate = new Date();
        System.out.println(myDate.toString());
    }
}
```

Move your cursor to the error marker by the line **Date myDate = new Date();**.



```
package time;

import java.sql.*;
import java.util.*;

public class TestDate {
    public static void main(String[ ] args){
                    = new Date();
                    rintln(myDate.toString());
    }

}
```

Multiple markers at this line
- The type Date is ambiguous
- The type Date is ambiguous

Go to the API and into the **java.sql** package. Scroll down to the Class Summary and the **Date** class. Okay, our **Date** class is there. Now, go back to the Packages Summary. Go into the **java.util** package. Scroll down to the Class Summary and the **Date** class. Hmm, it's in there too! No wonder Java said it was ambiguous. There are **Date** classes in both of the packages we tried to import.

Note Imported packages cannot allow ambiguity. If two packages have classes with the same name, then use the fully qualified name of the class to disambiguate.

Edit the **TestDate** class as shown in **blue**:

CODE TO TYPE:

```
package time;

import java.sql.*;
import java.util.*;

public class TestDate {

    public static void main(String[ ] args){
  java.util.Date myDate = new java.util.Date();
        System.out.println(myDate.toString());
 }
}
```

Save and run it.

Remove the line **import java.util.*;**. Because the class uses the fully qualified name: **java.util.Date**, removing that line has no impact on your result.

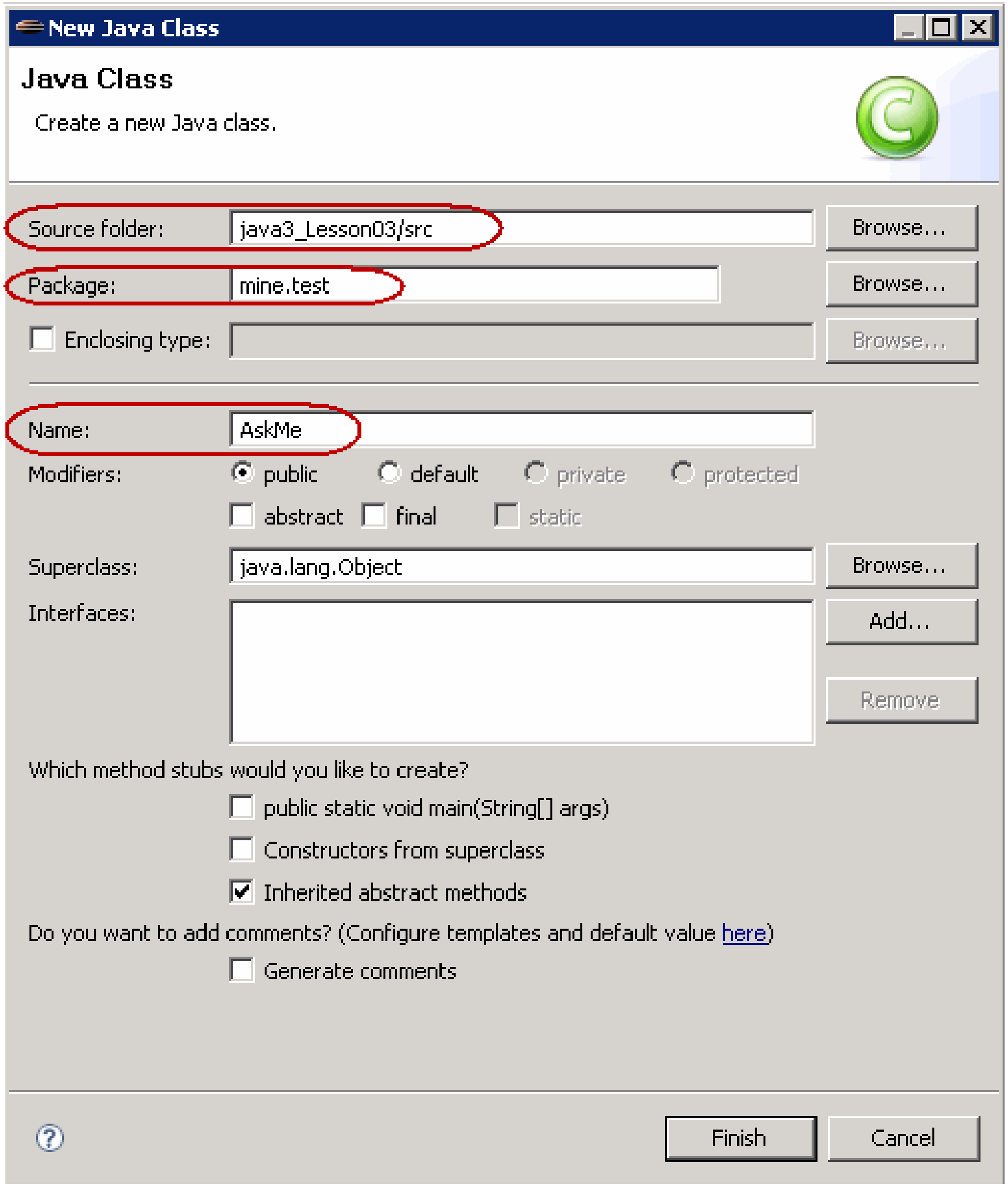
Note If you use fully qualified names for a class, then you don't need the import statement for the class.

Packages eliminate the potential for conflicting class names in different groups.

Naming a Subpackage

To name packages, we enter the **package** declaration as the first line of code. In Eclipse, we enter that package name when we set up the class. So how do we name *subpackages*? I'm glad you asked!

In the java3_Lesson03 project, create a new **AskMe** class, with the package name **mine.test**, as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson03/src

Package: mine.test

□ Enclosing type:

Name: AskMe

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)

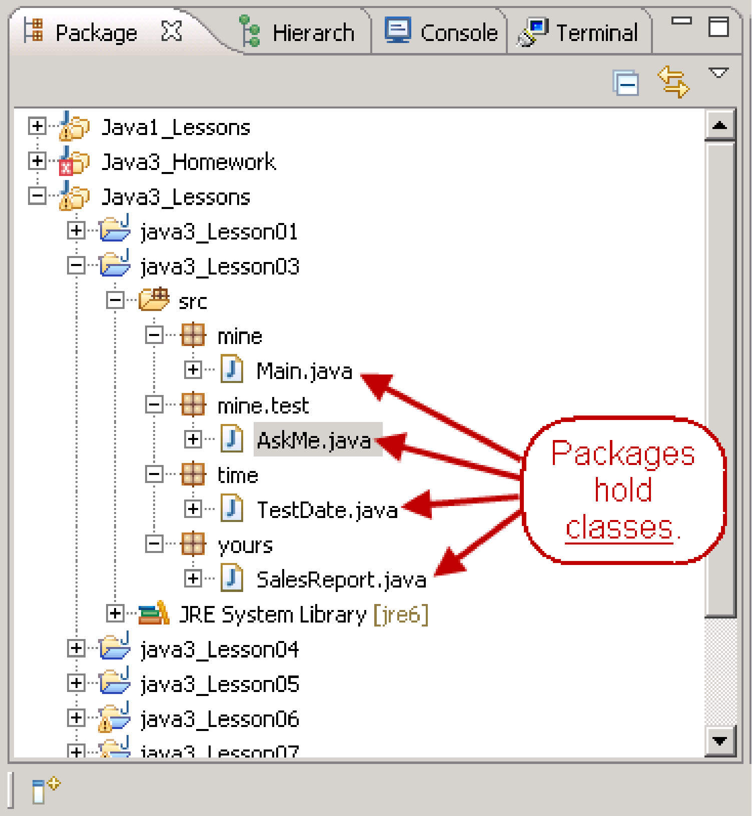
□ Constructors from superclass

☑ Inherited abstract methods

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

□ Generate comments

Voila! Check out the files in your Package Explorer window:



It's all there and named properly: New class, package, and subpackage. Nice!

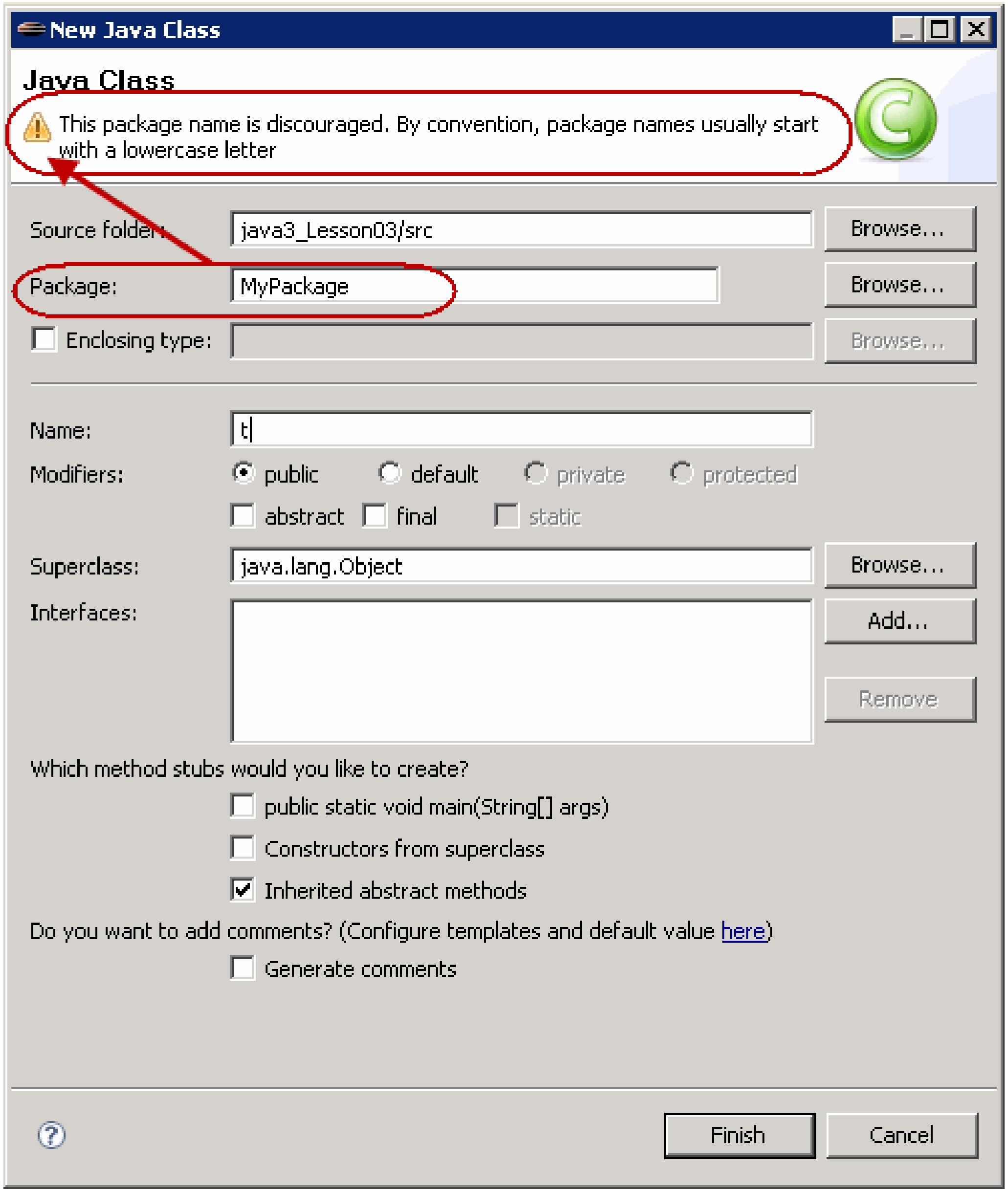
Conventions: Case Usage for Package and Class Names

Even with the freedom modularity allows us in naming, we Java programmers follow some hard conventions in naming as well. Here are the two basic naming conventions that we adhere to:

- Packages names consist of all lowercase letters.
- Classes begin with uppercase letters.

Eclipse will allow you to break these conventions, but it complains mightily when you do.

In the java3_Lesson03 project, create a new Class. In the New Java Class window that opens, for Source folder, enter (or accept) **java3_Lesson03/src**. For Package, enter **MyPackage**. Move to the Name field and type **t**. See the warning:



New Java Class

Java Class

This package name is discouraged. By convention, package names usually start with a lowercase letter

Source folder: java3_Lesson03/src

Package: MyPackage

□ Enclosing type:

Name: t

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)

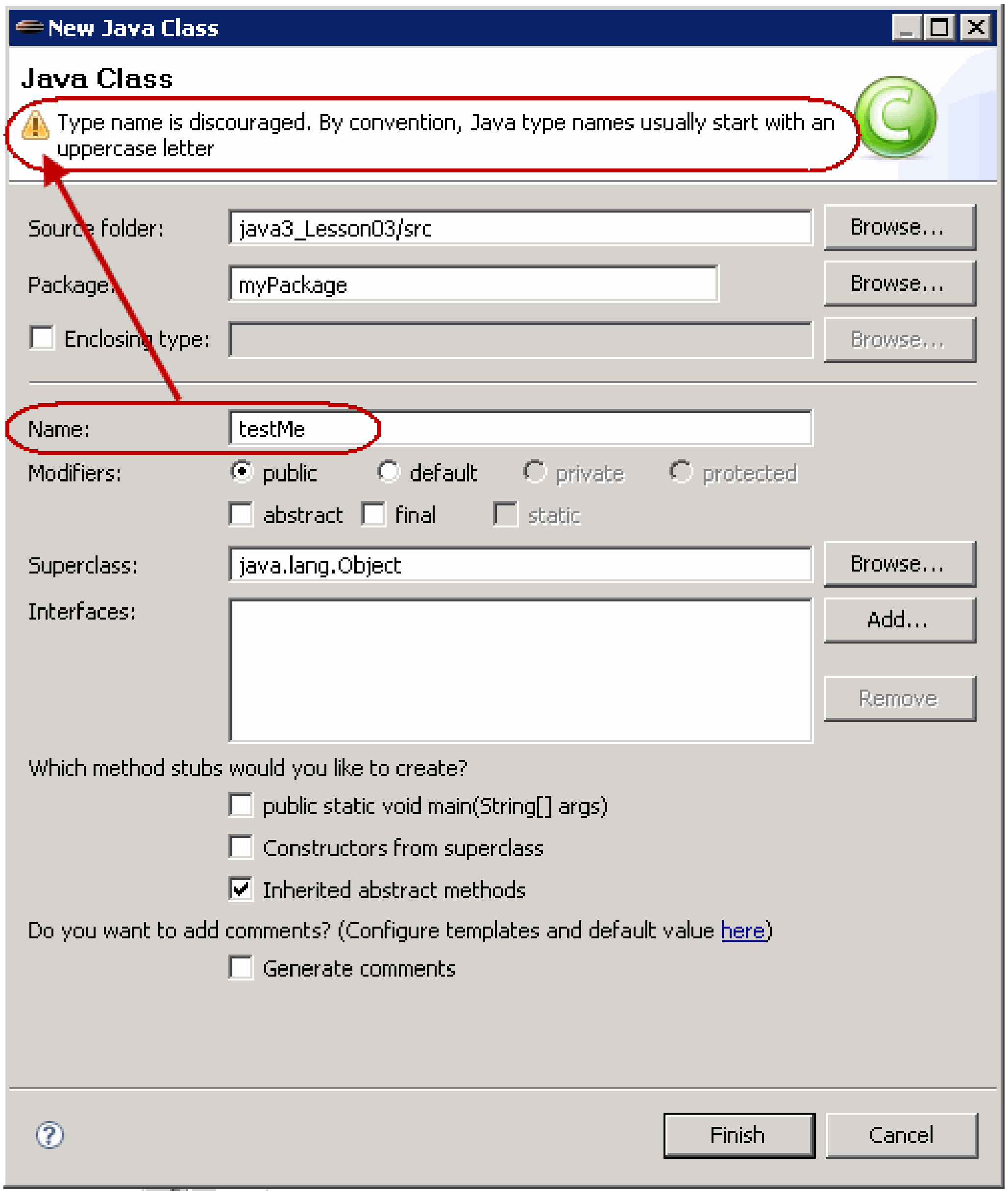
□ Constructors from superclass

☑ Inherited abstract methods

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

□ Generate comments

Go back and correct the Package name to **myPackage**, and then move again to the Name field, finish typing **testMe**, and observe the warning:



Type name is discouraged. By convention, Java type names usually start with an uppercase letter

Source folder: java3_Lesson03/src

Package: myPackage

□ Enclosing type:

Name: testMe

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)

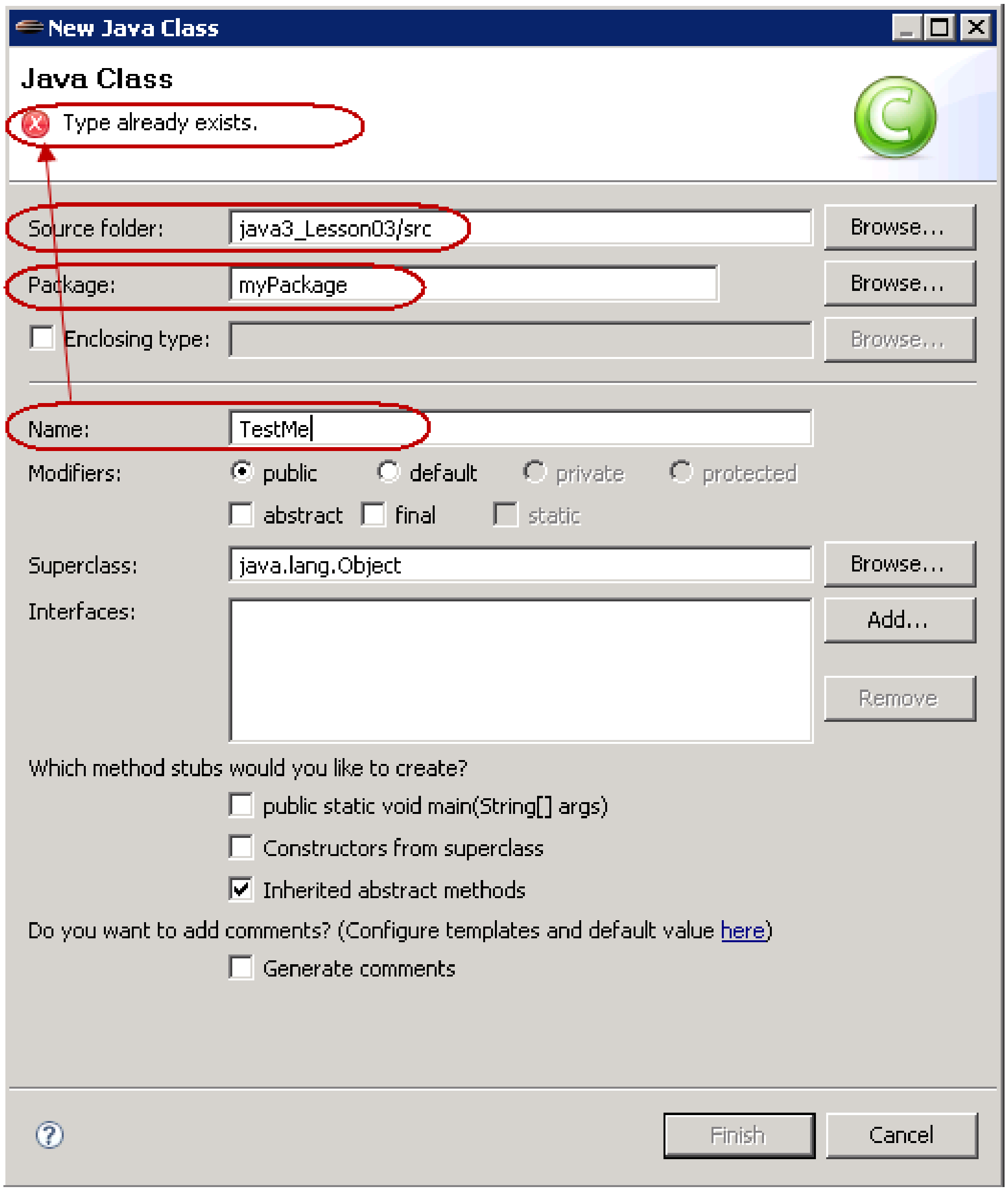
□ Generate comments

Change the class name to **TestMe** and click **Finish**.

Conventions: Duplicate Member Names

A package may not contain two members with the same name.

In the java3_Lesson03 project, create a new class. In the New Java Class window that opens, for Source folder, enter **java3_Lesson03/src**. For Package, enter **myPackage**. For Name, enter **TestMe**. See the warning:



Type already exists.

Source folder: java3_Lesson03/src

Package: myPackage

☐ Enclosing type:

Name: TestMe

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

Click **Cancel**.

Conventions: Company Names

One last convention to be aware of is that companies use their reversed internet domain names to begin their package names. For example, com.oreilly.school.java1 would be used for a package named java1 created by a programmer at school.oreilly.com.

Packages Highlights

Object-oriented programs:

- allow modular groups of classes to be made available.
- eliminate potential conflicts between class names in different groups.

There are three ways you can use a public package member from outside of its package:

- Refer to the member by its fully qualified name.
- Import the package member.
- Import the member's entire package using * (wildcard).

Uses for packages:

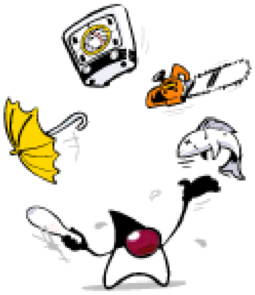
- Packages in Java are tools for grouping together related classes and interfaces.
- A class does not import packages, it imports classes and interfaces *in* packages.
- Dots (for subpackages) are like subfolders or subdirectories to the classloader.

Rules for using packages:

- There can only be one package statement in each source file.
- If a package statement appears in a Java source file, it must come first in that file (except for comments and white space).
- In **import**s, the * (wildcard) gets only the top-level compilation units in a package; it will not get classes and interfaces in a package's subpackages.
- A package cannot contain two members with the same name (see the Java Language Specification).
- Files of a package should be located in a subdirectory that matches the full package name.

Coming Attractions

Good job so far! Let's move on and learn about the stuff that's found *inside* of these packages: inheritance trees, classes, interfaces, exceptions, errors, enum, and more! See you in the next lesson...



Software Design: Inheritance

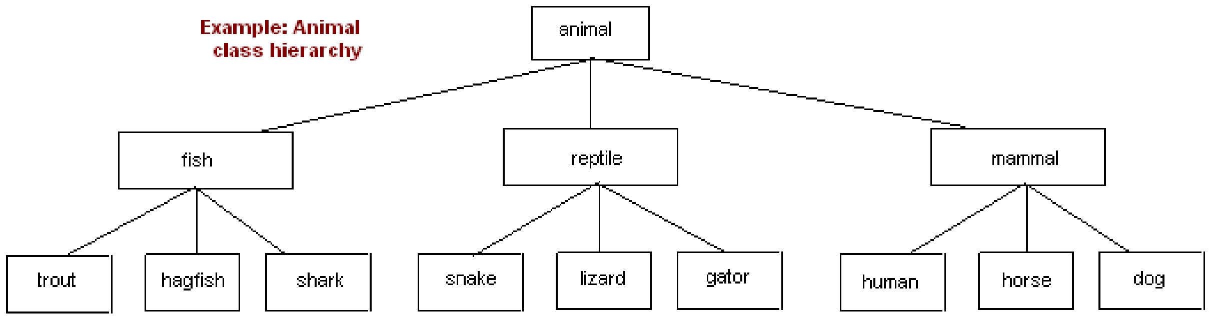
Origins and Organization

So far we've learned that **packages** contain lists of related **classes** and **interfaces**. We've also learned about **variables and fields** and **methods** contained within classes.

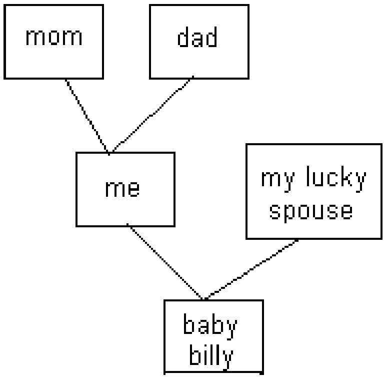
A key design construct in object-oriented programming is *inheritance*. We've seen how inheritance works for individual classes; now let's explore inheritance and how it works with other design elements.

Classification

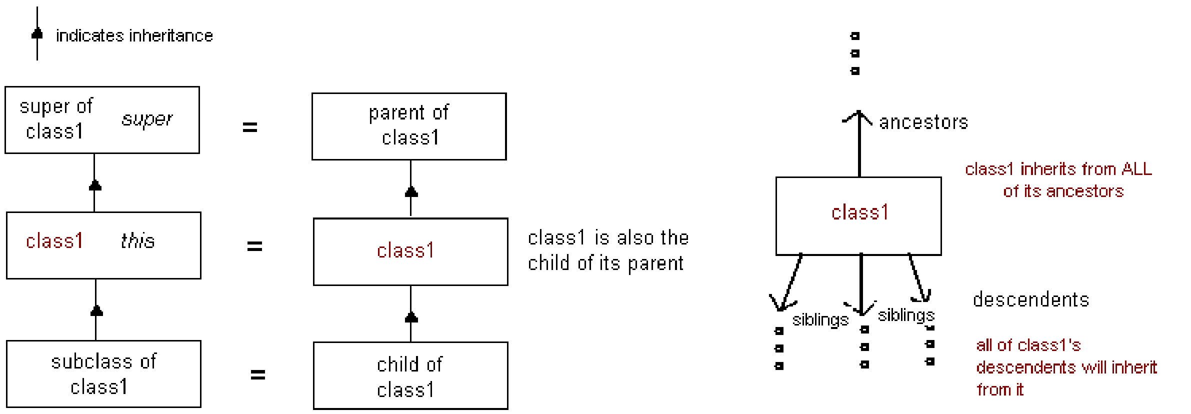
Object-oriented programming borrows the practice of classification from the field of biology. You might remember learning about classification back in the day, in a general biology class. It worked like this:



I'm no biologist, but I think you get the picture. Using classification in programming is similar, with a few differences. When we discuss class hierarchies and inheritance trees, it's not quite the same as biological inheritance trees, or the nodes in those trees that depict, for example, ancestral inheritance:



In programming, a subclass *must* possess every trait of its parent class, as well as additional features. By having additional features, it becomes specialized. The relationship between class and subclass works like this:



See also Superclasses and Subclasses: Java Language Specification.

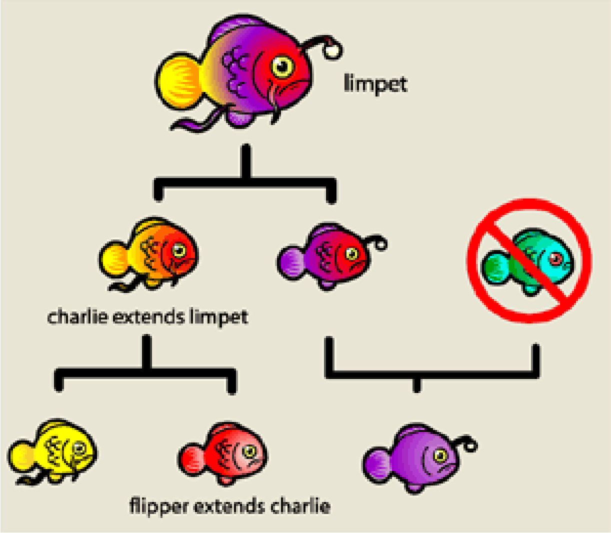
Programmers often use what is known as the *Is-A* test to confirm that a subclass is proper. As a programmer, you'd ask yourself (quietly, on the inside, so as not to seem weird), "*Is* the subclass truly *A* special case of the parent?" That is, "does the subclass contain all that the parent contains and more?" Hopefully, the answer is "yes," and the only characteristics established in a subclass (child) are those traits that distinguish the class from its parent and siblings.

Note A parent class is a generalization; a subclass is a specialization of the parent.

Because inheritance from the parent is a default activity, a subclass that does not possess *every* trait of the parent should not be a subclass.

Inheriting

In Java programming, a child may have only a single parent (super) class. However, a child may have many ancestors. A child inherits *every* trait from *every* every ancestor unless that trait has been overridden by an ancestor between them



Note Only non-private traits (variables and methods) are directly inherited. A trait that is private can only be accessed by accessing the ancestor object.

Inheritance: Shadowing

If you don't want a child to inherit each and every trait of each and every ancestor, you can *override* the methods or *shadow* the variables.

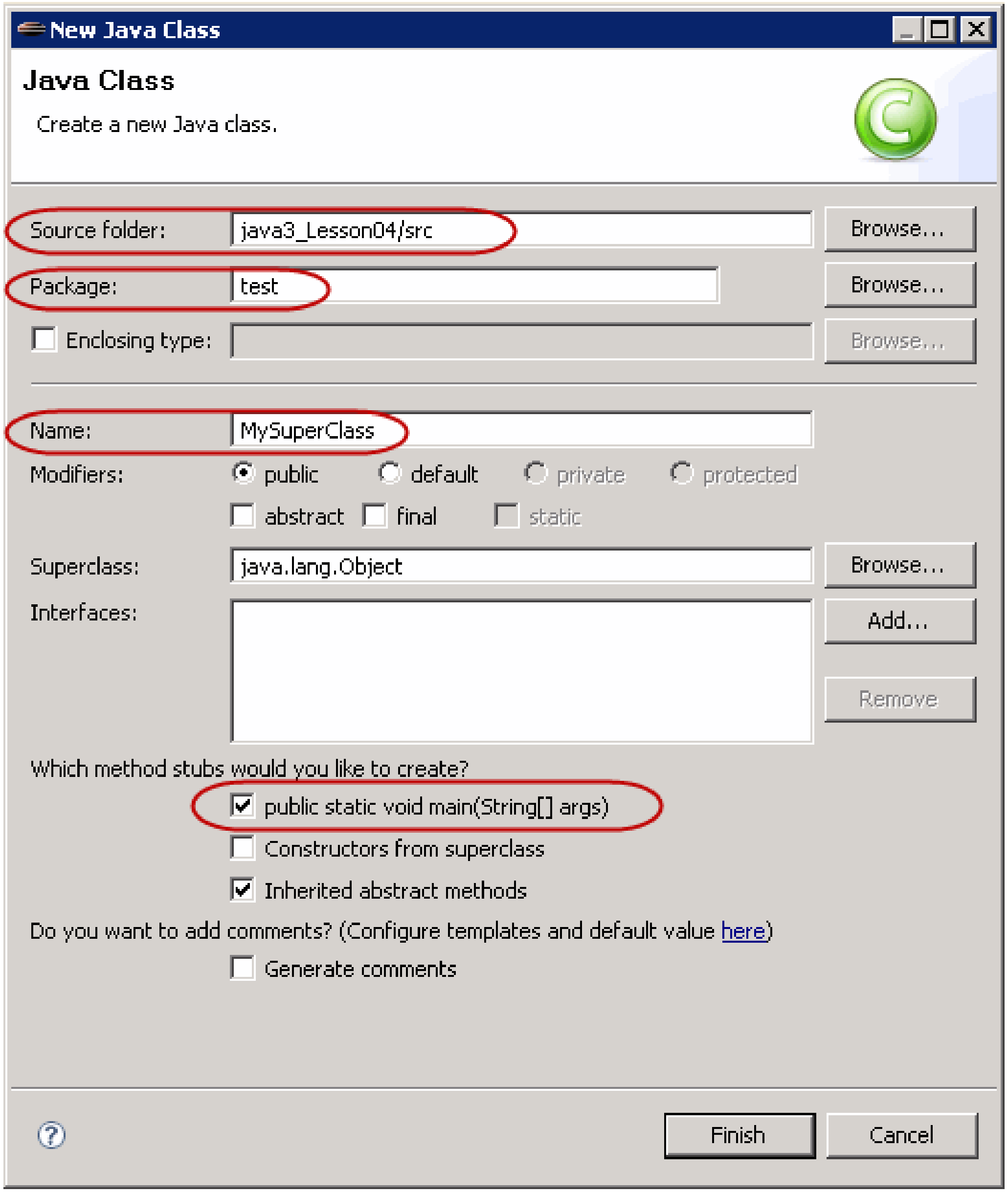
Note The only way *not* to inherit from the parent is for the child to *override* the methods of the parent. Overriding occurs when a class has a method with the same name, return type, and parameter listings (signature) as its superclass.

Shadowing variables occurs when a field (instance variable or class variable) is defined in both a class and its superclass.

Let's play with these concepts. Keep in mind that in our example code:

- **this** refers to the *current* instance of the class that is running the particular method at that time.
- **super** is the parent of the *current* class.

Create a new **java3_Lesson04** project in the java3_Lessons working set, then in that project, create a new **MySuperClass** class:



Java Class

Create a new Java class.

Source folder: java3_Lesson04/src

Package: test

□ Enclosing type:

Name: MySuperClass

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)

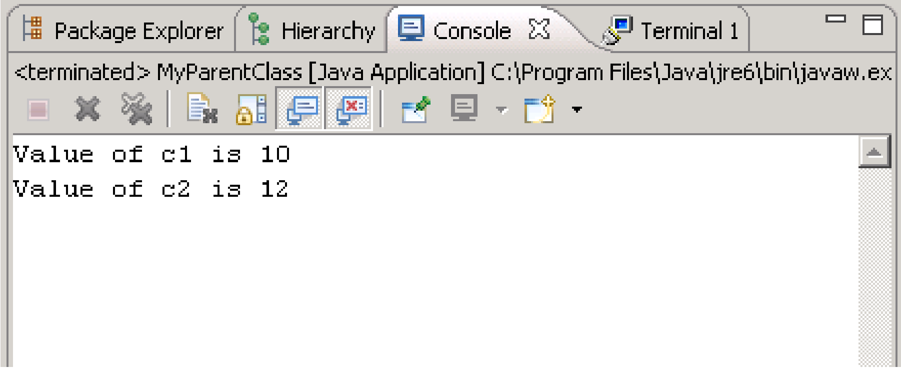
□ Generate comments

Type **MySuperClass** as shown in **blue**:

CODE TO TYPE:

```
package test;

public class MySuperClass {

    int i;

    public static void main(String[] args) {
        MySuperClass c1 = new MySuperClass();
        System.out.println("Value of c1 is " + c1.i);
        MySuperClass c2 = new MySuperClass(12);
        System.out.println("Value of c2 is " + c2.i);
    }

    public MySuperClass() {
        i = 10;
    }

    public MySuperClass(int value){
        this.i = value;
    }
}
```

Save and run it.



<terminated> MyParentClass [Java Application] C:\Program Files\Java\jre6\bin\javaw.ex

```
Value of c1 is 10
Value of c2 is 12
```

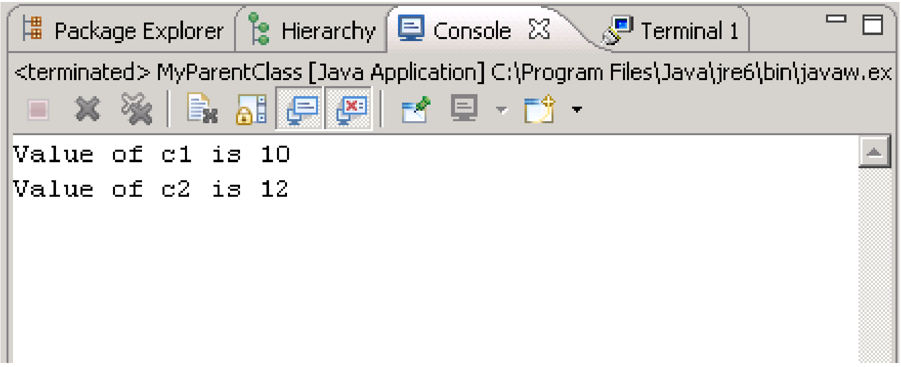
Do you understand the result? **c1**'s **i** value is derived from the constructor with no passed parameter, where **i** is set to 10 in the constructor; **c2** calls the constructor with an integer (12) passed, so its **i** is set to the passed value.

In the two constructors, switch **this.i** with **i**, as shown, adding the code shown in **blue** and removing the code shown in **red**:

CODE TO TYPE:

```
package test;

public class MySuperClass {

    int i;

    public static void main(String[] args) {
        MySuperClass c1 = new MySuperClass();
        System.out.println("Value of c1 is " + c1.i);
        MySuperClass c2 = new MySuperClass(12);
        System.out.println("Value of c2 is " + c2.i);
    }

    public MySuperClass() {
        this.i = 10;
    }

    public MySuperClass(int value){
        this.i = value;
    }
}
```

Save and run it.

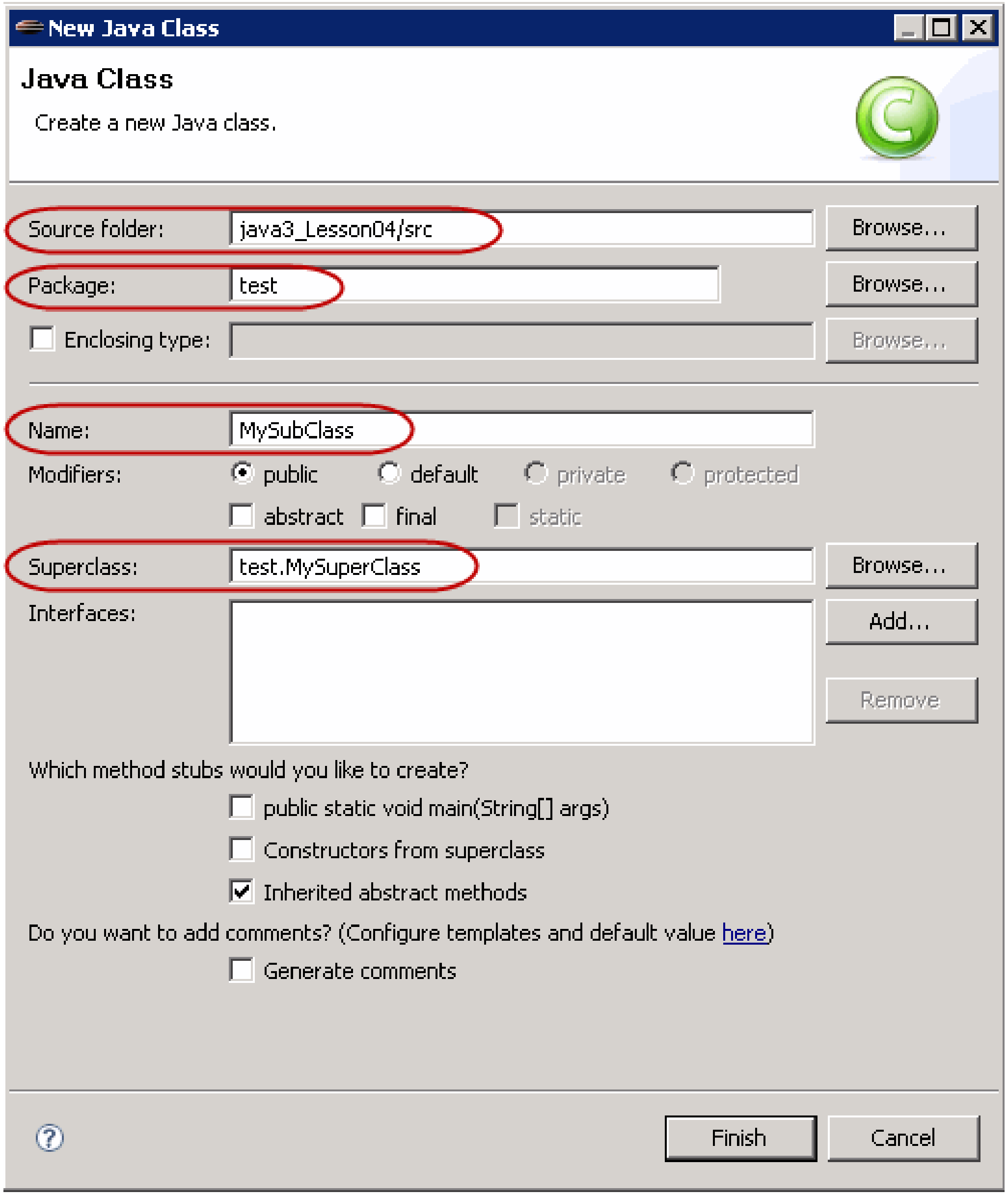


<terminated> MyParentClass [Java Application] C:\Program Files\Java\jre6\bin\javaw.ex

```
Value of c1 is 10
Value of c2 is 12
```

We got the same result. You can see that within a class, placing **this** in front of access to a class or instance variable (or method) is optional. Its presence is inferred. **this** should *not*, however, be placed in front of the variable declaration as a field in the class. (Try it—Eclipse will let you know, in **red**, just how wrong that is.)

Now, let's make a subclass. Click on the package **test**, and then right-click for the popup menu and choose **New | Class**. Create the new class. For Superclass, replace **java.lang.Object** with **MySuperClass**. Use the assist light bulb—press Ctrl and the space bar at the same time (Ctrl+space)—it will fill in **test.MySuperClass**:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson04/src

Package: test

□ Enclosing type:

Name: MySubClass

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: test.MySuperClass

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 code in **blue** into **MySubClass** as shown:

CODE TO TYPE:

```
package test;

public class MySubClass extends MySuperClass {   // MySuperClass is now the pare
nt (super) of MySubClass
    public static void main(String[] args){
        MySubClass testing = new MySubClass();
        System.out.println("From MySubClass, the value of testing.i is " + testi
ng.i);
        System.out.println("Notice how this will not work " + testing.super.i);
    }
}
```

Oops! We have an error.

OBSERVE:

```
package test;

public class MySubClass extends MySuperClass {
    public static void main(String[] args){
        MySubClass testing = new MySubClass();
        System.out.println("From MySubClass, the value of testing.i is " + testi
ng.i);
        System.out.println("Notice how this will not work " + testing.super.i);
    }
}
```

We did not define an instance variable **i** in **MySubClass**. Actually, **MySubClass** doesn't have *any* of its own methods or variables; it only has the **main()** method to get it started. The **main()** method is not considered a method of **MySubClass**. **MySubClass** will need to inherit everything from its parent, **MySuperClass**.

In order to reference itself and its super, **this** and **super** must be contained *within* the code of a class. Java doesn't like the inclusion of **testing.super.i** in your code. **this** and **super** do not point to anything, except when they are *within* a specific class's methods.

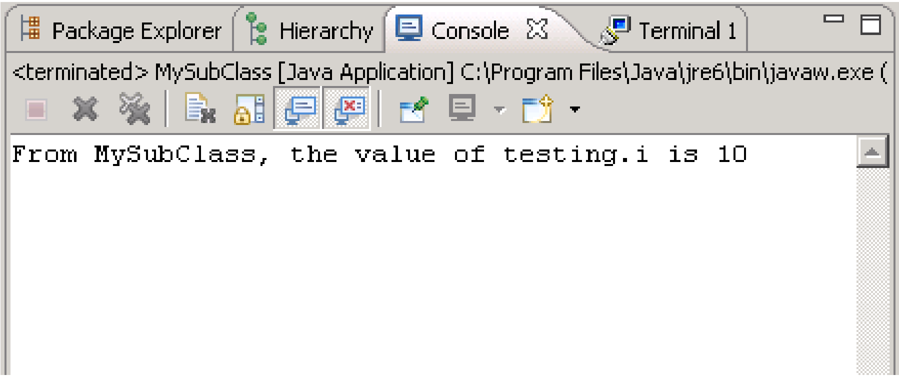
super.super.i wouldn't be able to access a superclass of a superclass either. **super.super.i** isn't legal syntax.

Remove the line shown in **red** from your code:

CODE TO EDIT:

```
package test;

public class MySubClass extends MySuperClass {

    public static void main(String[] args){
        MySubClass testing = new MySubClass();
        System.out.println("From MySubClass, the value of testing.i is " + testi
ng.i);
        System.out.println("Notice how this will not work " + testing.super.i);
    }
}
```

Save and Run it.



<terminated> MySubClass [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (

```
From MySubClass, the value of testing.i is 10
```

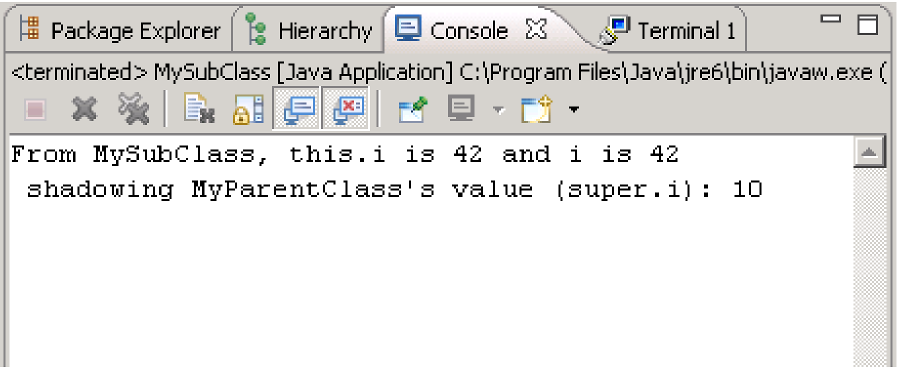
Even though **MySubClass** did not have a variable defined for **i**, it was able to get a value for **testing.i**. So as expected, the instance **testing** of class **MySubClass** inherited the variable **i** from its parent **MySuperClass**.

That's all well and good, but a little boring. Let's spice things up—let's give the subclass a variable **i** too. Edit MySubClass, adding the code shown in **blue** and removing the code shown in **red**:

CODE TO TYPE:

```
package test;

public class MySubClass extends MySuperClass {

    int i = 42;

    public static void main(String[] args){
        MySubClass testing = new MySubClass();
        System.out.println("From MySubClass, the value of testing.i is " + testi
ng.i);
        testing.whatsHere();
    }

    public void whatsHere() {
        System.out.println("From MySubClass, this.i is " + this.i + " and i is "
 + i);
        System.out.println(" shadowing MySuperClass's value (super.i): " + super
.i);
    }
}
```

Save and run it. Is this what you expected?



```
<terminated> MySubClass [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (
From MySubClass, this.i is 42 and i is 42
 shadowing MyParentClass's value (super.i): 10
```

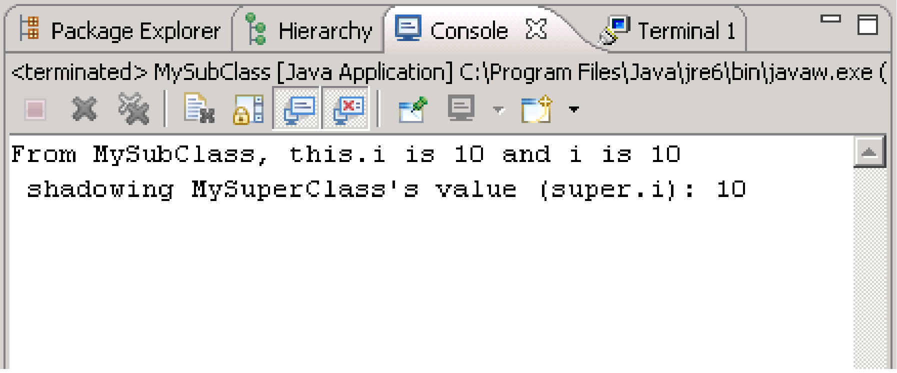
Note In MySubClass, click on the **i** in **+ this.i +**. It displays **int test.MySubclass.i**—*and* it highlights the **i** in int i = 42 at line 5. Then, click the **i** in **+ super.i**. It displays int test.MySuperClass.i. So, Eclipse knows the difference between the two variables and can help you figure it out as well!

In **MySubClass**, delete the line **int i = 42** as shown in **red**:

CODE TO EDIT:

```
package test;

public class MySubClass extends MySuperClass {

    int i = 42;

    public static void main(String[] args){
        MySubClass testing = new MySubClass();
        testing.whatsHere();
    }
    public void whatsHere(){
        System.out.println("From MySubClass, this.i is " + this.i + " and i is "
 + i);
        System.out.println(" shadowing MySuperClass's value (super.i): " + super
.i);
    }
}
```

Save and run it. Interesting, yes?



```
<terminated> MySubClass [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (
From MySubClass, this.i is 10 and i is 10
 shadowing MySuperClass's value (super.i): 10
```

All is as it should be, because **MySubClass** inherits by default.

Tip We could also shadow (or mask) a variable in a superclass by having a variable in the subclass with the same name, but of a different *type*. For example, **i** could be declared in the super as **int i** and it could also be declared in the subclass as **double i**. Use of **i** in the subclass would access the **double**. If you wanted to access the **int**, you could do it via **super.i** Be aware that these are two distinct **i** variables.

Inheritance: Overriding

Now that we have some understanding of *shadowing variables*, let's look at **overriding**. We'll give both classes a method with the same signature. Edit **MySuperClass**, adding the code shown in **blue** and removing the **main()** method as shown in **red**:

CODE TO TYPE:

```
package test;

public class MySuperClass {

    int i;

    public static void main(String[] args) {
        MySuperClass c1 = new MySuperClass();
        System.out.println("Value of c1 is " + c1.i);
        MySuperClass c2 = new MySuperClass(12);
        System.out.println("Value of c2 is " + c2.i);
    }

    public MySuperClass() {
        this.i = 10;
    }

    public MySuperClass(int value){
        i = value;
    }

    public void addToI (int j) {
        i = i + j;
        System.out.println("After MySuperClass addToI, i is " + i);
    }
}
```

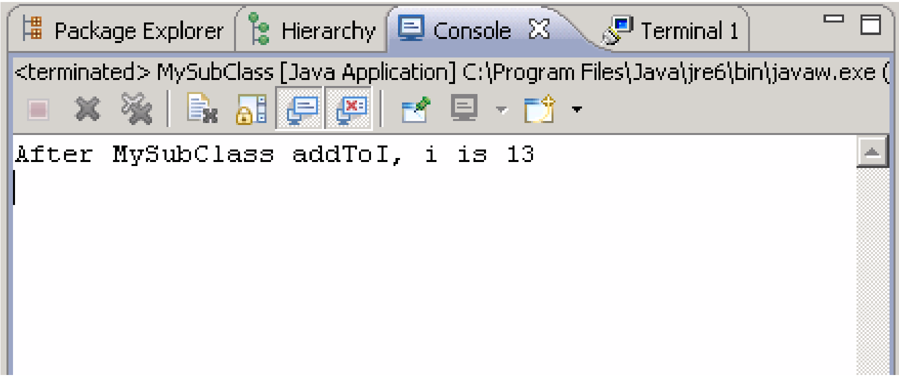
Save it.

Edit **MySubClass**, adding the code shown in **blue** and removing the **whatsHere()** code as shown in **red**:

CODE TO TYPE:

```
package test;

public class MySubClass extends MySuperClass {

    public static void main(String[] args){
        MySubClass testing = new MySubClass();
        testing.whatsHere();
        testing.addToI(6);
    }

    public void whatsHere() {
        System.out.println("From MySubClass, this.i is " + this.i + " and i is "
 + i);
        System.out.println(" shadowing MySuperClass's value (super.i): " + super
.i);
    }

    public void addToI (int j) {
        i = i + (j / 2);
        System.out.println("After MySubClass addToI, i is " + i);
    }
}
```

Save and run it.



```
<terminated> MySubClass [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (
```

```
After MySubClass addToI, i is 13
```

Let's trace the code.

OBSERVE:

```
package test;

public class MySubClass extends MySuperClass {

    public static void main(String[] args){
        MySubClass testing = new MySubClass();
        testing.addToI(6);
    }

    public void addToI (int j) {
        i = i + (j / 2);
        System.out.println("After MySubClass addToI, i is " + i);
    }
}
```

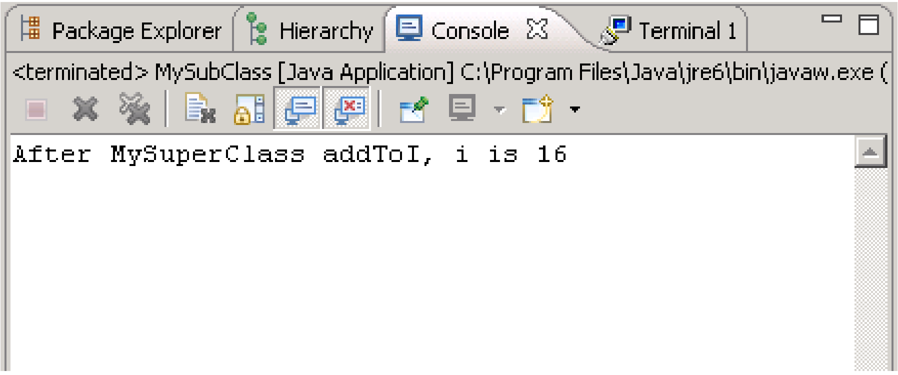
Rather than inheriting the method from **MySuperClass**, the **addToI()** method of **MySubClass** is used. The subclass inherits the **i** value (10), then uses the passed **j** value (6); 10 + (6/2) is 13. Without an **addToI()** method of its own, **MySubClass** would need to inherit the **addToI()** method from **MySuperClass**.

Edit **MySubClass** by commenting out the **addToI()** method as shown in **blue**:

CODE TO TYPE:

```
package test;

public class MySubClass extends MySuperClass {

    public static void main(String[] args){
        MySubClass testing = new MySubClass();
        testing.addToI(6);
    }

/*
    public void addToI (int j) {
        i = i + (j / 2);
        System.out.println("After MySubClass addToI, i is " + i);
    }
*/


}
```

Save and run it.



```
<terminated> MySubClass [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (
```

```
After MySuperClass addToI, i is 16
```

OBSERVE:

```
package test;

public class MySuperClass {
    int i;

    public  MySuperClass() {
        this.i = 10;
    }

    public MySuperClass (int value){
        i = value;
    }

    public void addToI (int j) {
        i = i + j;
        System.out.println("after MySuperClass addToI, i is " + i);
    }
}
```

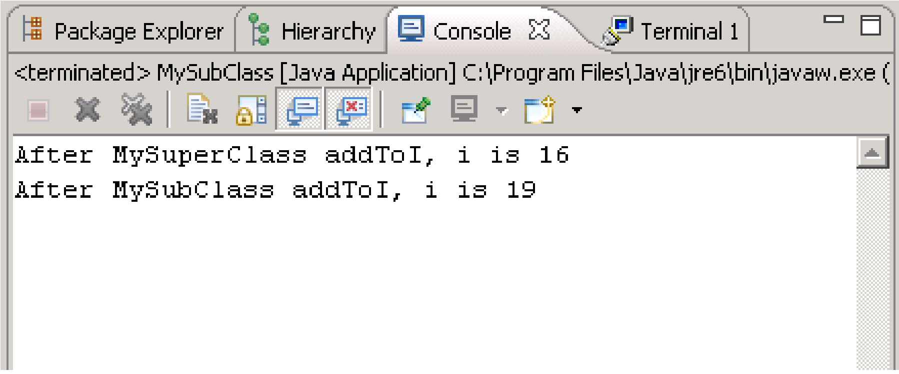
The value of **i** is 16, which means that **MySuperClass**'s method added the inherited **i** value (10) to the passed **j** value (6). At other times, we may want our classes to do what's known as a *wrap-around.* That is, we want them to inherit, and then do their own stuff as well. Let's check out an example.

Edit **MySubClass**, adding the code shown in **blue**, and uncommenting the **addToI()** method as shown in **red**:

CODE TO TYPE:

```
package test;

public class MySubClass extends MySuperClass {

    public static void main(String[] args){
        MySubClass testing = new MySubClass();
        testing.addToI(6);
    }
/*
    public void addToI (int j) {
        super.addToI (j);
        i = i + (j / 2);
        System.out.println("After MySubClass addToI, i is " + i);
    }
*/
}
```

Save and run it.



<terminated> MySubClass [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (

```
After MySuperClass addToI, i is 16
After MySubClass addToI, i is 19
```

In **MySubClass**, we inherit the **i** value (10). We use **j**'s passed value (6), and pass this within super's call. This produces **i** as a value of 16. Then we return to the local method in **MySubClass**, which adds **i**'s current value (16) to the 6/2 and we get a new **i** value of 19. Great!

Note If the method that you want to wrap-around is a *constructor*, use this syntax: **super();** If you want to pass from one constructor to another within a class, use this syntax: **this(*values to be passed*);** When the call is within a constructor itself, do not add the name of the constructor.

Working with Constructors

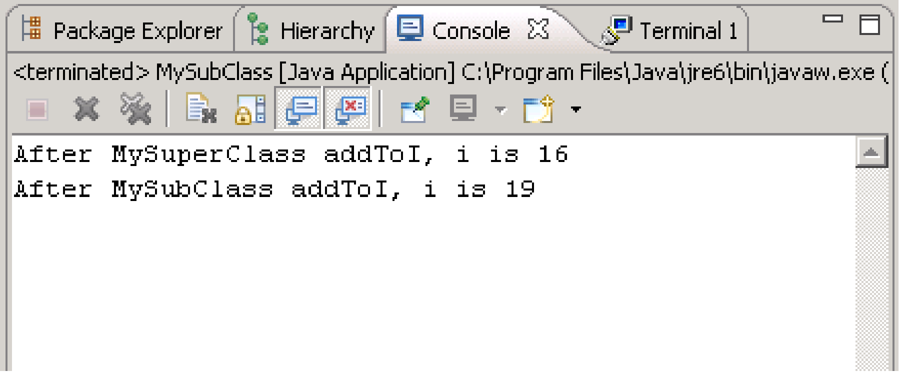
Chaining

We can never experiment too much, right? Edit **MySuperClass** as shown in **blue**:

CODE TO TYPE:

```
package test;

public class MySuperClass {
    int i;

    public  MySuperClass() {
        this(10);
    }

    public MySuperClass (int value){
        this.i = value;
    }

    public void addToI (int j) {
     i = i + j;
     System.out.println("after MySuperClass addToI, i is " + i);
    }
}
```

Save it. Go to **MySubClass** and run it.



```
<terminated> MySubClass [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (
After MySuperClass addToI, i is 16
After MySubClass addToI, i is 19
```

Nothing changed—perfect!

OBSERVE:

```
package test;

public class MySuperClass {
    int i;

    public MySuperClass()  {
        this(10);
    }

    public MySuperClass (int value){
        this.i = value;
    }

    public void addToI (int j) {
     i = i + j;
     System.out.println("after MySuperClass addToI, i is " + i);
    }
}
```

We changed the **MySuperClass()** constructor to pass a default value of 10 to the other **MySuperClass(**int **value)** constructor, which sets the instance variable **i** to that value.

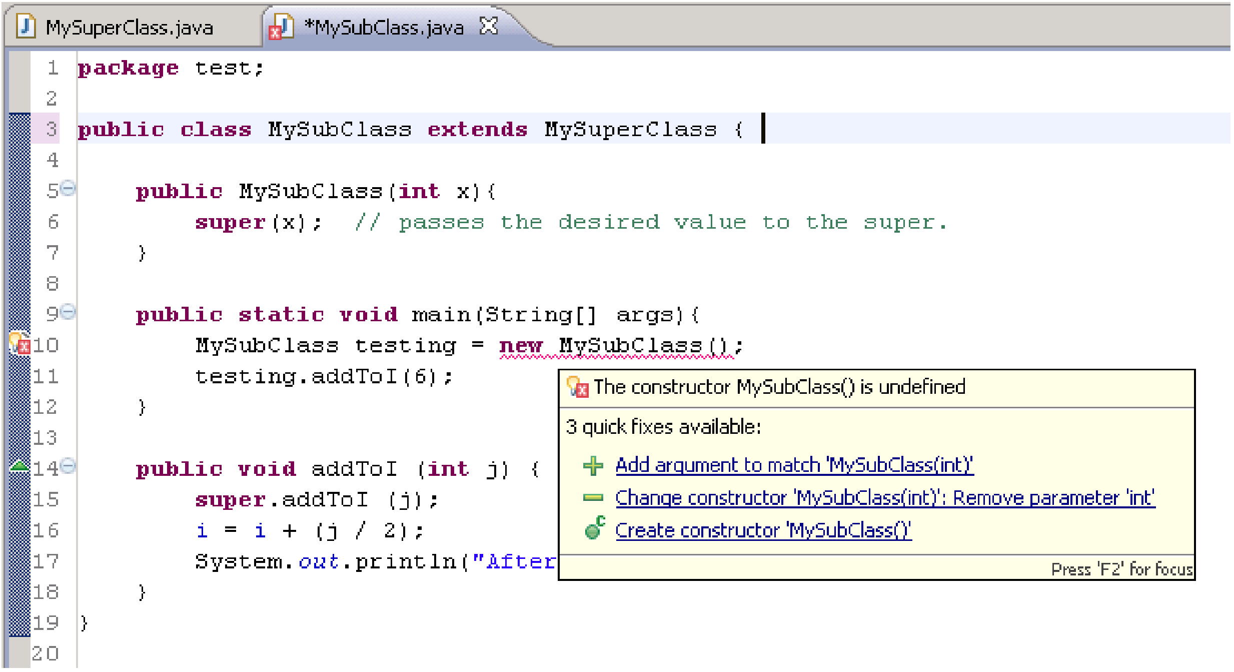
This is called *constructor chaining*, and it allows us to build up a chain of constructors to account for various ways of constructing an object when it is instantiated. When we call **this(10)**, we are chaining the **MySuperClass()** constructor to the **MySuperClass(**int **value)** constructor.

Now we know the syntax used in order to "chain" constructors that are within the same class. So, what's the correct syntax to use to call a parent's constructor? Edit the code in **MySubClass** as shown:

CODE TO TYPE:

```
package test;

public class MySubClass extends MySuperClass {

    public MySubClass(int x){
        super(x);  // passes the desired value to the super.
    }

    public static void main(String[] args){
        MySubClass testing = new MySubClass();
        testing.addToI(6);
    }

    public void addToI (int j) {
        super.addToI (j);
        i = i + (j / 2);
        System.out.println("after MySubClass addToI, i is " + i);
    }
}
```

Java doesn't seem to like the line where we instantiated **MySubClass**:



```
package test;

public class MySubClass extends MySuperClass {

    public MySubClass(int x){
        super(x);  // passes the desired value to the super.
    }

    public static void main(String[] args){
        MySubClass testing = new MySubClass();
        testing.addToI(6);
    }

    public void addToI (int j) {
        super.addToI (j);
        i = i + (j / 2);
        System.out.println("After
    }
}
```

The constructor MySubClass() is undefined

3 quick fixes available:

- Add argument to match 'MySubClass(int)'
- Change constructor 'MySubClass(int)': Remove parameter 'int'
- Create constructor 'MySubClass()'

Press 'F2' for focus

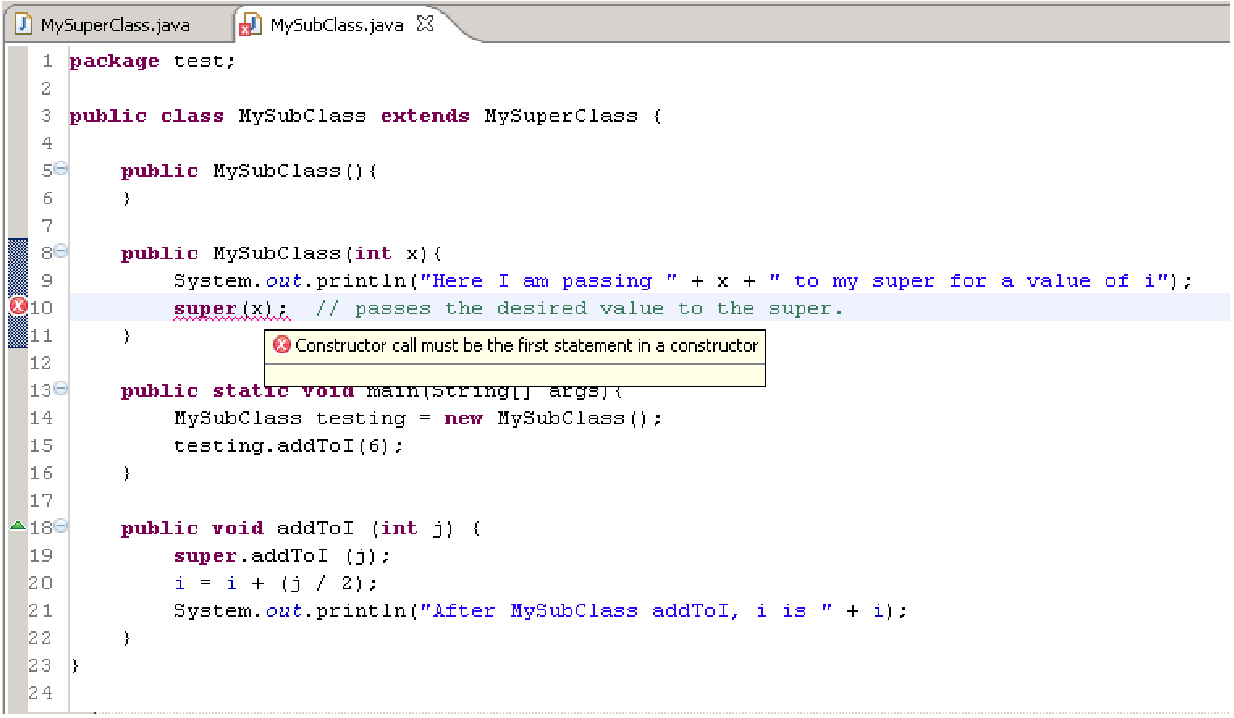
So, why didn't Java complain about that before? Because, when there are *no constructors defined*, Java provides an empty *default constructor*, a constructor containing no arguments. But, *if there is a defined constructor* in your class, the default constructor will not be supplied, and the **MySubClass()** constructor does not exist.

We'll add an empty default constructor to our code. We'll also test a restriction placed on the order in which super's constructors are called. When a constructor calls another constructor, the call must be located within the first line of the constructor code. Check out what happens if it isn't. Edit **MySubClass** as shown:

CODE TO TYPE:

```
package test;
public class MySubClass extends MySuperClass {

    public MySubClass(){
    }

    public MySubClass(int x){
        System.out.println("Here I am passing " + x + " to my super for a value
of i");
        super(x);  // passes the desired value to the super.
    }

    public static void main(String[] args){
        MySubClass testing = new MySubClass();
        testing.addToI(6);
    }

    public void addToI (int j) {
        super.addToI (j);
        i = i + (j / 2);
        System.out.println("after MySubClass addToI, i is " + i);
    }
}
```

Progress! Now we have a new error message:



```
package test;

public class MySubClass extends MySuperClass {

    public MySubClass(){
    }

    public MySubClass(int x){
        System.out.println("Here I am passing " + x + " to my super for a value of i");
        super(x);  // passes the desired value to the super.
    }

    public static void main(String[] args){
        MySubClass testing = new MySubClass();
        testing.addToI(6);
    }

    public void addToI (int j) {
        super.addToI (j);
        i = i + (j / 2);
        System.out.println("After MySubClass addToI, i is " + i);
    }
}
```

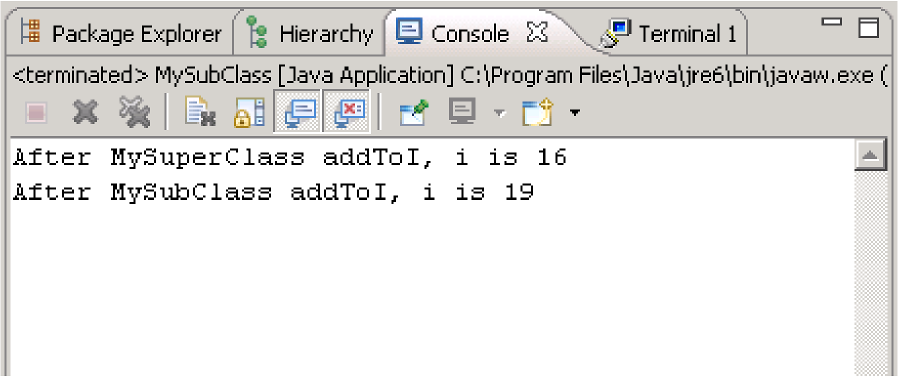
Constructor call must be the first statement in a constructor

If we call a **super()** constructor, it *must* be the first statement in the constructor. Switch the order of the statements in the second constructor as shown in **blue**:

CODE TO TYPE:

```
package test;

public class MySubClass extends MySuperClass {
    public MySubClass(){
    }

    public MySubClass(int x){
        super(x);  // passes the desired value to the super.
        System.out.println("Here I am passing " + x + " to my super for a value of i");
    }

    public static void main(String[] args){
        MySubClass testing = new MySubClass();
        testing.addToI(6);
    }

    public void addToI (int j) {
        super.addToI (j);
        i = i + (j / 2);
        System.out.println("after MySubClass addToI, i is " + i);
    }
}
```

Save and run it.



```
<terminated> MySubClass [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (
```

```
After MySuperClass addToI, i is 16
After MySubClass addToI, i is 19
```

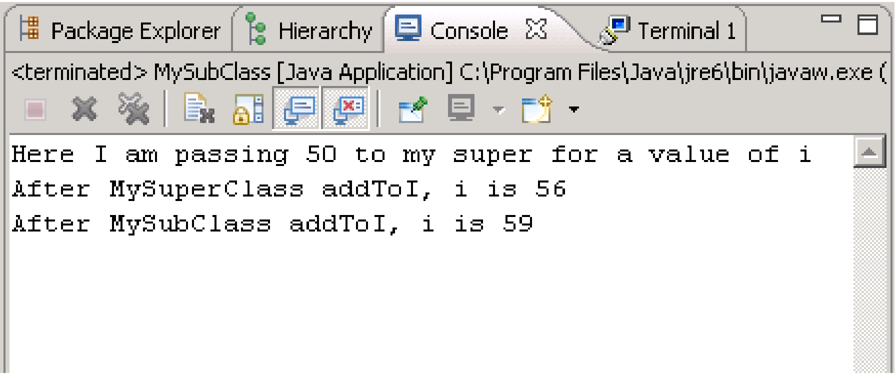
There is no change to our result. Because we still called with the *default constructor*, Java returned the default value for **i**.

Edit the instantiation code in **MySubClass** as shown:

CODE TO TYPE:

```
package test;

public class MySubClass extends MySuperClass {

    public MySubClass(){
    }

    public MySubClass(int x) {
        super(x); // passes the desired value to the super.
        System.out.println("Here I am passing " + x + " to my super for a value
of i");
    }

    public static void main(String[] args){
        MySubClass testing = new MySubClass(50);
        testing.addToI(6);
    }

    public void addToI (int j) {
        super.addToI(j);
        i = i + (j / 2);
        System.out.println("after MySubClass addToI, i is " + i);
    }
}
```

Save and run it. Ahh, change!



<terminated> MySubClass [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (

```
Here I am passing 50 to my super for a value of i
After MySuperClass addToI, i is 56
After MySubClass addToI, i is 59
```

Chain of Command

When we inherit from our **super**s, we are in essence creating new instances of them because we *are* them. That's deep, huh? Let's get a clear understanding of the way inheritance works during construction and make this concept less abstract. So what is the "construction path" anyway? That is, which constructors are called first in the inheritance chain?

Variables are set from the top of the inheritance chain and work their way down, because as we go down the chain, we get more *specific*. The sequence Java uses when instantiating an instance of a class is to allocate memory and then initialize defaults in this order:

1. superclass initialization
2. instance variable initialization
3. constructor initialization

If the first statement in a constructor is *not* an explicit call to a constructor of the superclass with the **super** keyword, then Java inserts the call **super()**. In other words, Java calls the constructor with no arguments.

If the first line of a constructor (**C1**) uses the **this()** syntax to invoke another constructor (**C2**) of the class, this is an exception to the default call to the **super()** for initialization. Java relies on **C2** to invoke the superclass constructor and does not insert a **super()** call into **C1**. Java waits until it actually starts a constructor to either implicitly or explicitly start the call to **super()**.

So, upon instantiation, **MySubClass** inherits everything from its ancestors. Let's follow the path of this instantiation by adding some **System.out.printlns**. While we're at it, let's add some more constructors. Edit **MySuperClass**. Add the **blue** code and delete the **red** code as shown:

CODE TO TYPE:

```
package test;

public class MySuperClass {
    int i, otherVariable;

    public  MySuperClass() {
        this(10);
        System.out.println("\nMySuperClass(): the default value is " + 10);
    }

    public MySuperClass(int value){
        this(value, 42);
        System.out.println("\nMySuperClass(int value): value is " + value + " with a new default of 42");
    }

    public MySuperClass(int value, int value2){
        this.i = value;
        this.otherVariable = value2;
        System.out.println("\nMySuperClass(int value, int value2): Something I inherit from Object: " + this.toString());
        System.out.println("  i is " + i + " and otherVariable is " +otherVariable);
    }

    public void addToI (int j) {
        i = i + j;
        System.out.println("after MySuperClass addToI, i is " + i);
    }
}
```

Save it. Edit **MySubClass**. Add the **blue** code and delete the **red** code as shown:

CODE TO TYPE:

```
package test;

public class MySubClass extends MySuperClass {
    int j;

    public MySubClass(){
        //  default of super() will first be called by Java
        System.out.println("\nMySubClass(), returned after waiting for everythin
g to get done and come back to me");
        System.out.println("  after supers are called by default, inherited i is
 " +i + " and my own j is initialized to " +j);
        System.out.println("  when all is done here, j is now " + ++j);
    }

    public MySubClass(int x){
        super(x);
        System.out.println("\nMySubClass(int x), returned after passing value of
 " + x +
        "  and then waiting for everything to get done and come back to me");
    }

    public static void main(String[] args){
        MySubClass testing = new MySubClass(50);
        testing.addToI(6);
        System.out.println("\nEnd of main after instantiation. Value of i is " +
 testing.i);
    }

    public void addToI (int j) {
        super.addToI (j);
        i = i + (j / 2);
        System.out.println("After MySubClass addToI, i is " + i);
    }
}
```

Save and run it. Follow your output lines to verify the order of initialization.

OBSERVE: MySuperClass Code

```
public MySuperClass() {
    this(10);
    System.out.println("\nMySuperClass(): the default value is " + 10);
}

public MySuperClass(int value) {
    this(value, 42);
    System.out.println("\nMySuperClass(int value): value is " + value + " with a
 new default of 42");
}

public MySuperClass(int value, int value2){
    this.i = value;
    this.otherVariable = value2;
    System.out.println("\nMySuperClass(int value, int value2): Something I inher
it from Object: " + this.toString());
    System.out.println(  i is " + i + " and otherVariable is " +otherVariable);
}
```

OBSERVE: MySubClass Code

```
public class MySubClass extends MySuperClass {
    int j;

    public MySubClass(){
        //  default of super() will first be called by Java
        System.out.println("\nMySubClass(), returned after waiting for everythin
g to get done and come back to me");
        System.out.println("  after supers are called by default, inherited i is
 " +i + " and my own j is initialized to " +j);
        System.out.println("  when all is done here, j is now " + ++j);
    }

    public MySubClass(int x){
        super(x);
        System.out.println("\nMySubClass(int x), returned after passing value of
 " + x +
        "  and then waiting for everything to get done and come back to me");
    }

    public static void main(String[] args){
        MySubClass testing = new MySubClass();
        System.out.println("End of main after instantiation.  Value of i is " +
testing.i);
    }
}
```

OBSERVE: MySubClass Output

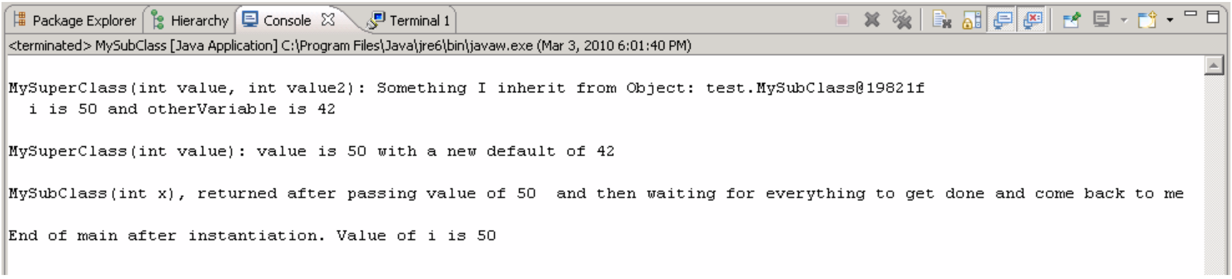
```
MySuperClass(int value, int value2): Something I inherit from Object: test.MySub
Class@addbf1 (Note: Address after @ may be different on various systems.)
  i is 10 and otherVariable is 42

MySuperClass(int value): value is 10 with a new default of 42

MySuperClass(): the default value is 10

MySubClass(), returned after waiting for everything to get done and come back to
 me
  after supers are called by default, inherited i is 10 and my own j is initiali
zed to 0
  when all is done here, j is now 1

End of main after instantiation. Value of i is 10
```

Now, edit the **main()** method to again pass 50 as a parameter as shown:

CODE TO TYPE:

```
package test;

public class MySubClass extends MySuperClass {

    int j;

    public MySubClass(){
        //  default of super() will first be called by Java
        System.out.println("\nMySubClass(), returned after waiting for everythin
g to get done and come back to me");
        System.out.println("  after supers are called by default, inherited i is
 " +i + " and my own j is initialized to " +j);
        System.out.println("  when all is done here, j is now " + ++j);
    }

    public MySubClass(int x){
        super(x);  // passes the desired value to the super.
        System.out.println("\nMySubClass(int x), returned after passing value of
 " + x +
        "  and then waiting for everything to get done and come back to me");
    }

    public static void main(String[] args){
        MySubClass testing = new MySubClass(50);
        System.out.println("\nEnd of main after instantiation. Value of i is " +
 testing.i);
    }
}
```

Save and run it.



```
<terminated> MySubClass [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Mar 3, 2010 6:01:40 PM)

MySuperClass(int value, int value2): Something I inherit from Object: test.MySubClass@19821f
  i is 50 and otherVariable is 42

MySuperClass(int value): value is 50 with a new default of 42

MySubClass(int x), returned after passing value of 50  and then waiting for everything to get done and come back to me

End of main after instantiation. Value of i is 50
```

See how it works? We go to the top, inherit our variables, and come back down. Play around with these and make sure you understand the workings of inheritance.

In a Nutshell

Class Properties and Relationships:

- A class is defined by the members it defines *and* the members it inherits from its ancestors.
- A class only has one parent. That parent is its *super*.
- A class inherits every **method** from its parent and ancestors, unless the class overrides the method (defines the method with the same signature).
- A class inherits every **field** from its parent and ancestors, unless the class shadows its ancestor's field.
- Inheritance is *transitive*. If classA inherits from classB, and classB inherits from classC, then classA also inherits from ClassC. A class inherits from *all* of its ancestors.
- There is no limit to the number of children a class can have.
- Two children with the same parent are siblings.
- Siblings are not related by inheritance (one is not derived from the other). They do share characteristics passed down from the common parent and ancestors.

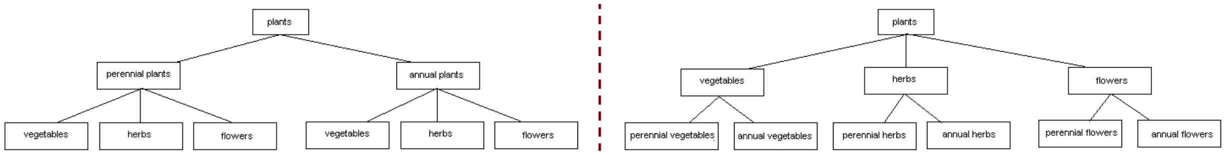
This table illustrates class (field or method) access and the use of *access modifiers* (an example of access rights (permission) granted for a particular member's information):

Modifier	Visibility
public	All classes
private	None (only within own class)
protected	Classes in package and subclasses inside or outside package
none (default)	Classes in same package (sometimes called *package private*)

Design considerations for classes and class hierarchies:

- Common features should be kept as high in the hierarchy as possible.
- Classes should be "as simple as possible and no simpler" (Gotta love that Einstein!).
- The only characteristics explicitly established in a child class should be those that make the class distinctive from its parent and from its siblings.
- There is no single best hierarchical structure.

This figure illustrates that last bullet point:



Which should it be?

Either one of these hierarchies could be useful. The specific requirements of your application will influence your design choices. Try to be as class-conscious as possible and think carefully about your organization. There are plenty of pitfalls out there that programmers might fall into when creating hierarchical structures, so choose wisely grasshopper.

Good design of classes and the class hierarchy structure means your code will be reusable and easier to maintain. These are signs of excellent programming, and will make you a hero to all programmers who follow in your footsteps.

Java's Design

So enough already with the particulars! Let's look at an example of larger applications that work through inheritance and well-planned class design. Where will we find such an example, you ask? In the API API within the *Java Programming Language* itself. The *Java Programming Language* is a shining example of good design, code reuse, and nice, clean use of inheritance. Somebody really smart put this language together.

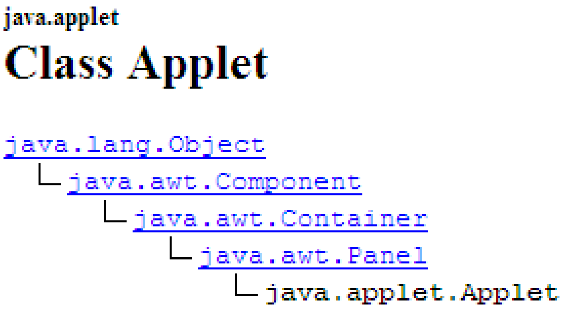
The API

Applets don't require **Window**s of their own, because the web browser provides Applets with windows as well as menus. But Applets do need **Panel**s in their browser windows to display the **Applet** Java code. Applications are different from Applets in that they require **Frame**s which surround **Window**s. These **frame**s allow menu items to be added, which in turn enable the Java code to run. **Panel**s and **Window**s are **Containers**, which are **Components** with graphical representations (meaning they can be displayed on the screen and interact with the user). These GUIs are **Object**s, the base class in Java.

Frames are the basic GUIs for applications (Java *not* running on the web), that require a GUI for interaction. **Applet**s are the basic GUIs for browsers (Java running on the web).

Click the API icon API under the Eclipse menu bar, then click the **JavaAPI** Java link tab to show its content.

Click on the **java.applet** package. Scroll down and open the **java.applet.Applet** class, then look at its inheritance tree:



java.applet

Class Applet

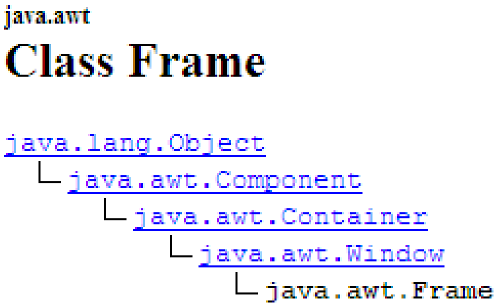
```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └java.awt.Panel
              └java.applet.Applet
```

Applet inherits everything defined in **Panel**, which inherits everything from **Container**, which inherits everything from **Component**, which inherits everything from **Object**. So, **Applet** gets all of the **fields** and **methods** from each of these ancestors.

Now, go back to the **java.applet.Applet** API page. Scroll down to the **Field Summary**. Notice that **Applet** does not define any of its own **fields**, but instead inherits **fields** from **Component** and **ImageObserver**. Scroll down past the **Method Summary**. Not only does **Applet** define a number of its own **methods**, but **Applet** also inherits the methods of **Panel**, **Container**, **Component**, and **Object**.

So why didn't **Applet** display any inherited **fields** from **Panel**, **Container**, and **Object**? You can find the answer to this and many other befuddling Java-related questions in the API.

Okay, now go back to the Packages page (you can get there by clicking the **Back** button twice). Click on the **java.awt** package. Scroll down and open the **java.awt.Frame** class, then look at its inheritance tree:

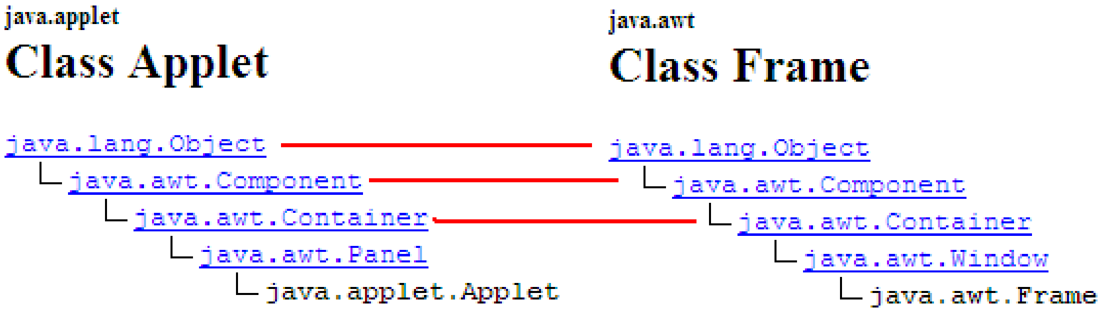


java.awt

Class Frame

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └java.awt.Window
              └java.awt.Frame
```

That's really cool. I'll go over the reasons I think it's so cool in a minute, but first, look at this:



java.applet

Class Applet

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └java.awt.Panel
              └java.applet.Applet
```

java.awt

Class Frame

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └java.awt.Window
              └java.awt.Frame
```

Here the designers of Java show us how inheritance can be used in a powerful way. Classes like **Object**, **Component**, and **Container** are defined classes with specialized capabilities and specific purposes. The presence of such clearly defined classes with corresponding inheritance trees for both frames and applets, means that GUIs can be defined the same way whether running on the web or not.

Go to the JavaAPI tab, open the **java.awt.Frame** class and its inheritance tree, and read its general description. Now open the **java.awt.Window** class, and read its general description. Now click on **java.awt.Container**, and read its general description. At the top of this page, click on **java.awt.Component**, and read its general description. And finally, click on **java.lang.Object**, and read its general description.

Note The "windowing" elements are in **java.awt**, while the ever-present **Object** class is within **java.lang** (which is conveniently available without **import**ing). Now that's good **packag**ing!

The API is a warehouse of good Java program design. As we observed:

- It defines classes cleanly and succinctly so their use and potentials are clear and specific.
- Its classes are reusable for multiple related purposes.
- Because of early good design, its classes are easier to *maintain*.
- This low-maintenance style of design allows changes we make to a parent to be reflected automatically in the descendants.
- In addition, because classes are succinct, it's clear where changes should be made.

When classes are made with clear and specific specifications and capabilities, it's easier to *build* applications with them. The API is a Java launch pad for programmers. It provides ready-made classes that can be inherited and then extended right away for programmers' specific application requirements.

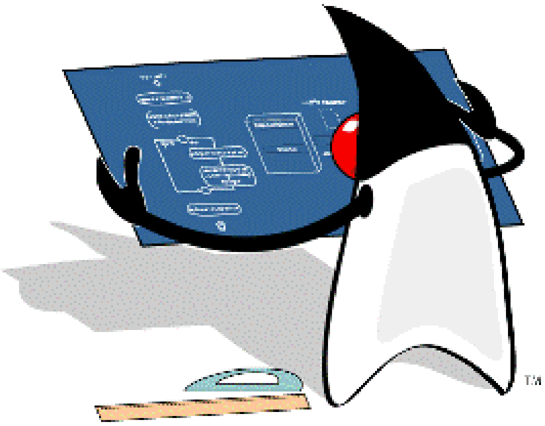
Making Our Own: Early Design

In our upcoming labs, we'll develop a tool for drawing graphical objects (squares, circles, ellipses, triangles, and such). But we're not just going to *draw* the figures and leave it at that; we're going to *move* them around, *resize* them, and manipulate them in all sorts of ways. Each figure will be an **Object**, which will enable us to manipulate them individually. Specifically, each figure drawn must be an **instance** of a **class**.

Click on this example (allow blocked content if necessary) to get a feel for the project we'll be working to create. Go ahead and play around with it. You'll see in the example that:

- The appropriate action button must be chosen.
- There are only two graphical objects present.
- If you *draw* a figure, you can specify *which* figure to *move* with your mouse.
- Each drawing is an individual **instance**.

In the next lesson, **Abstract Classes**, we'll continue with this example and start implementing code.



There's a lot to digest here. Hang in there. You're doing great so far, keep it up! See you in the next lesson...

Software Design: Abstract Classes

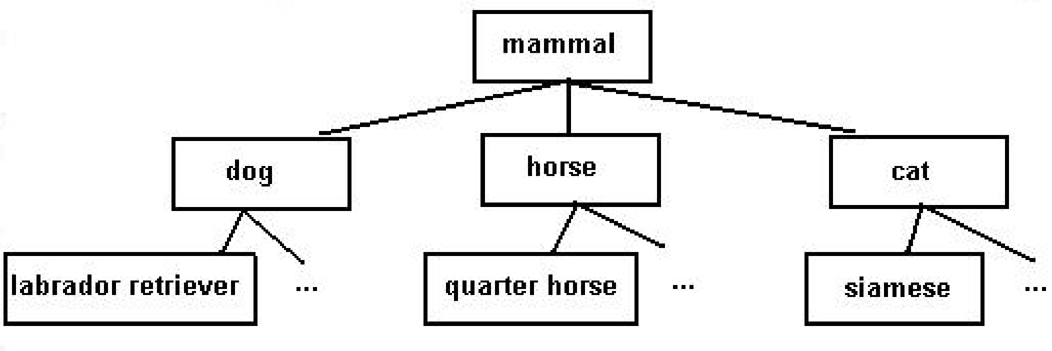
The Power of Abstraction

"There is nothing so prolific in utilities as abstractions."- Michael Faraday

Picture a mammal in your mind. Now, picture a dog. Did you imagine this particular mammal, or this particular dog? Probably not. There are, of course, many different types of mammals, and many different types of dogs. And while all dogs are mammals, not all mammals are dogs.

These familiar creatures will help us understand the concept of abstraction within hierarchies in Java. Mammals and dogs are classes of objects. Often, some classes within a class hierarchy are more "abstract" than others. Each step up in a hierarchy is broader and more abstract, than the one below. Using our example, "poodle" is more specific than "dog," and "dog" is more specific than "mammal."

In programming, we use hierarchies in a similar way, depending on our design goals. For example, within the mammal class, we have many subclasses (dogs, cats, giraffes) and each of those subclasses may have many more subclasses (for example, the subclass "dogs" contains labradors, schnauzers, collies, poodles, and so on).

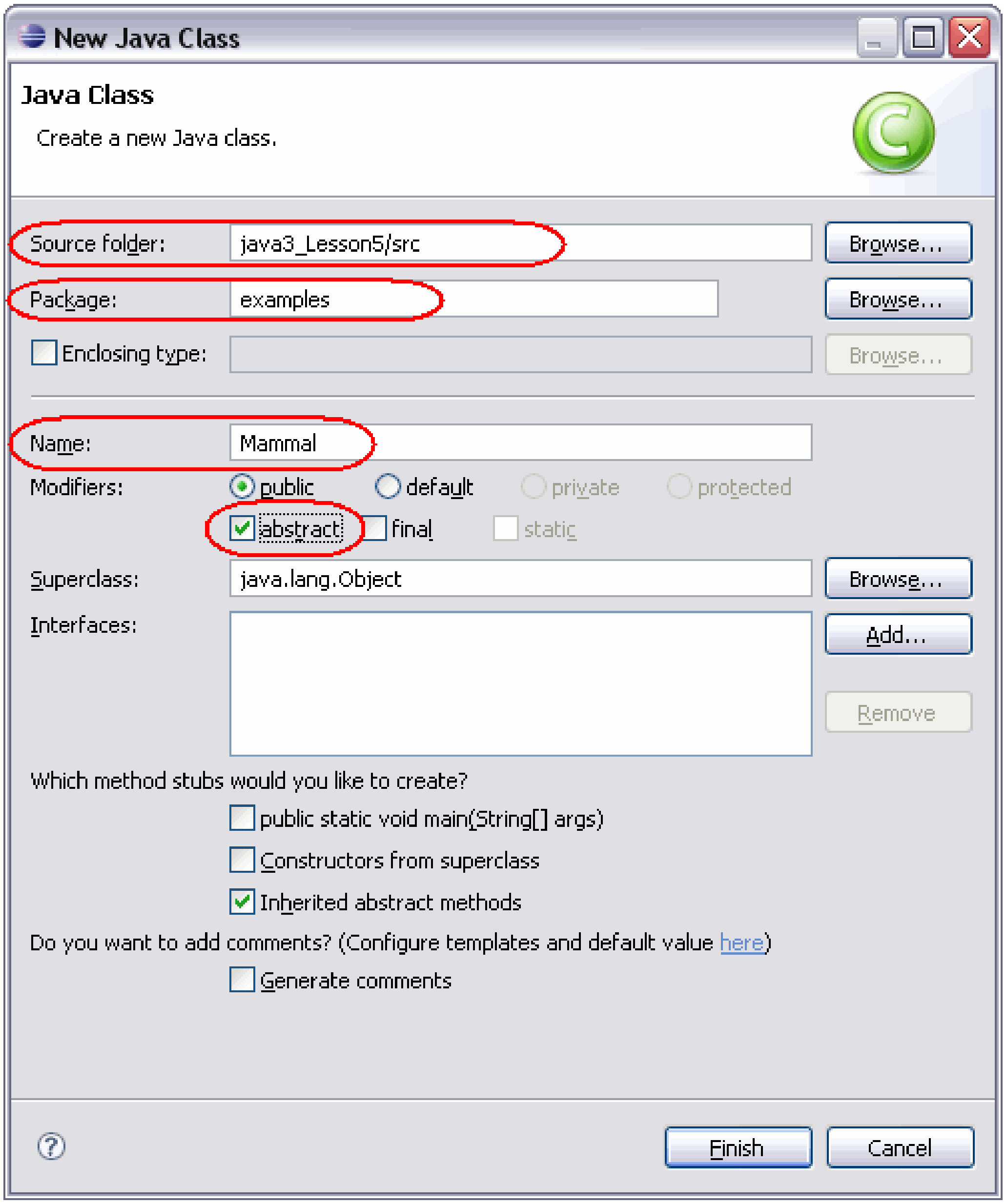


Java allows us to create *abstract classes*, which have structures similar to *interfaces*. An interface is a class that has no methods implemented. In an abstract class, some of the methods it contains may be implemented, others may not. Abstract classes are "place holders" or intermediate steps within a classification tree. If a programmer specifically names a class *abstract*, then the user cannot create instances of that class; the abstract class may only be *extended* as a subclass, and then those subclasses can be instantiated. In object-oriented programming, a **class** is essentially abstract. The *template* indicates which methods an **instance** will possess. But until you actually instantiate the class, you don't have a *thing*; you only have its template.

Abstract classes allow us to define classes that specify *abstract* methods. That is, to *identify* methods that any subclass of that class will be able to do (which *defines* and *differentiates* the class). However, we do not specify how the method will be used until Java gets to the subclass with a specific implementation, because the different subclasses will perform these actions in different ways. For example, the mammal class may have an abstract method called **walk()**, and while both Dogs and Seals *walk*, they do so differently.

Syntax: Abstract Classes

Let's experiment with **abstract** classes. Create a new **java3_Lesson05** project in your java3_Lessons working set. In your new project, create a new class. In the New Java Class window, enter the information as shown:



Java Class

Create a new Java class.

Source folder: java3_Lesson5/src

Package: examples

□ Enclosing type:

Name: Mammal

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

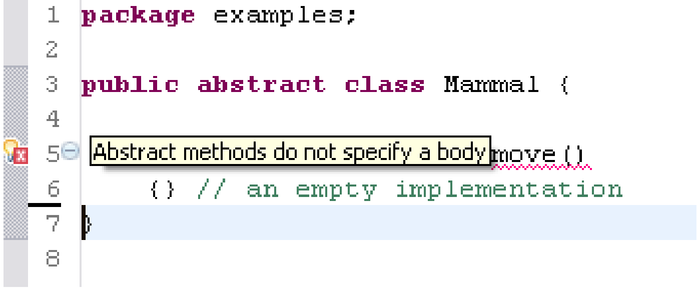
Click **Finish** when you're done.

Type the **Mammal** class code, adding the code in **blue** as shown:

CODE TO TYPE: Mammal.java

```
package examples;

public abstract class Mammal {

 public abstract void move(){ }
}
```

Hmm, Java didn't seem to like this:



```
package examples;

public abstract class Mammal {

    Abstract methods do not specify a body move ()
        {} // an empty implementation
}

```

Java has a problem with the presence of those braces { } because even though they're empty, they really *are* implementing the method—as a *no-op* (*no operation*). Java is disturbed because we are creating code that does nothing on purpose. Instead of confusing Java this way, we want the method to be specified (declared) as a method to help define the class, but we do not want the method implemented. We can do that. Edit **Mammal**. Add the **blue** code (the semicolon) and remove the **red** code as shown:

CODE TO EDIT: Mammal

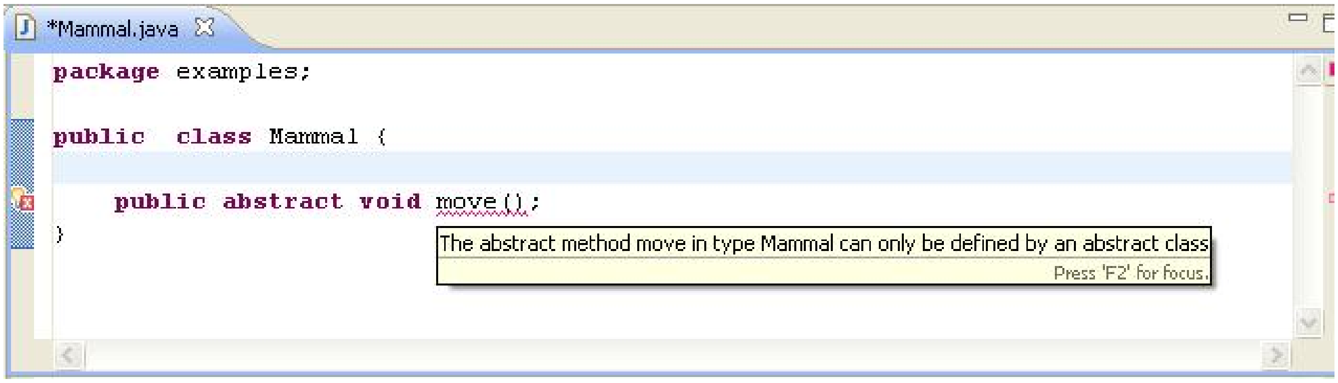
```
package examples;

public abstract class Mammal {

    //The semicolon in the next line is blue.
    public abstract void move();{ }
}
```

Ah, that's better. To create an **abstract** method, we *declare* the method, but we don't *implement* it. So, what happens if we have an abstract method, but not an abstract *class*? Let's find out. Remove the **red** code as shown:

CODE TO EDIT: Mammal

```
package examples;

public abstract class Mammal {
    public abstract void move();
}
```

Java didn't like this either:



```
package examples;

public  class Mammal {

    public abstract void move();
}
```

The abstract method move in type Mammal can only be defined by an abstract class

Press 'F2' for focus.

When a class that is not abstract contains an abstract method, an error occurs. Alright then, what happens if we have an abstract class without any abstract *methods*? Go ahead and edit **Mammal**. Remove the **red** code and add the **blue** code as shown:

CODE TO TYPE:

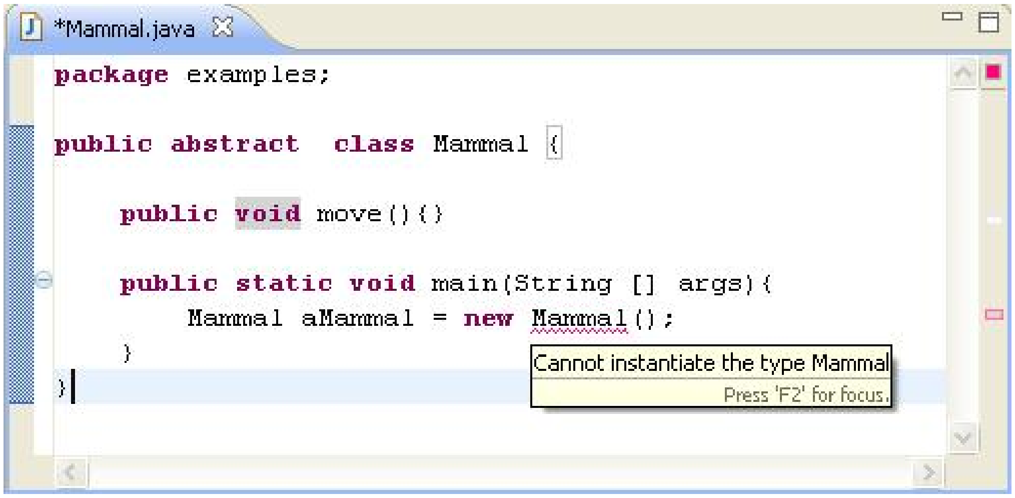
```
package examples;

public abstract class Mammal {
    //The semi-colon in the next line is RED
    public abstract void move();{ }
}
```

We've solved one problem. An abstract class is one that is declared abstract; it *may or may not* include abstract methods. So, what are the ramifications of declaring the class **abstract**? Edit **Mammal** by adding the **blue** code as shown:

CODE TO TYPE:

```
package examples;

public abstract  class Mammal {   // an abstract class

 public void move(){}          // no abstract methods

 public static void main(String [] args){
  Mammal aMammal = new Mammal();  // try to instantiate
 }
}
```

That's weird. Java didn't like that either:



```
package examples;

public abstract  class Mammal {

    public void move(){}

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

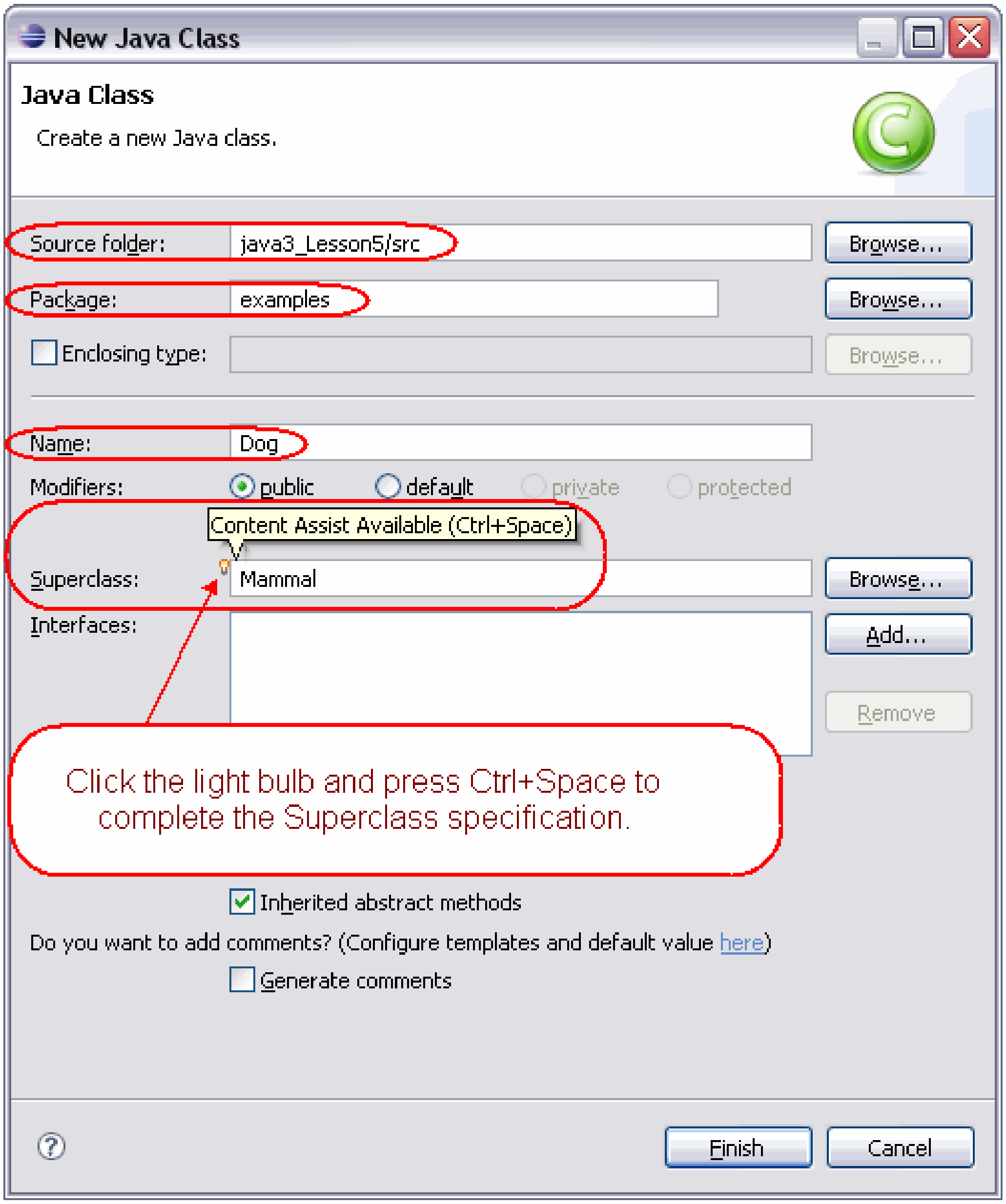
Cannot instantiate the type Mammal
Press 'F2' for focus.

Return the class to its original condition by removing the **main()** method. Remove the **red** code as shown:

CODE TO MODIFY:

```
package examples;

public abstract  class Mammal {   // an abstract class

 public void move(){}          // no abstract methods

 public static void main(String [] args){
  Mammal aMammal = new Mammal();  // try to instantiate
 }
}
```

Abstract classes cannot be instantiated, but they *can* be subclassed. Create a new class in the java3_Lesson05 project. In the New Java Class window, enter the information shown below:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson5/src

Package: examples

□ Enclosing type:

Name: Dog

Modifiers: ● public ○ default ○ private ○ protected

Content Assist Available (Ctrl+Space)

Superclass: Mammal

Interfaces:

Click the light bulb and press Ctrl+Space to complete the Superclass specification.

☑ Inherited abstract methods

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

□ Generate comments

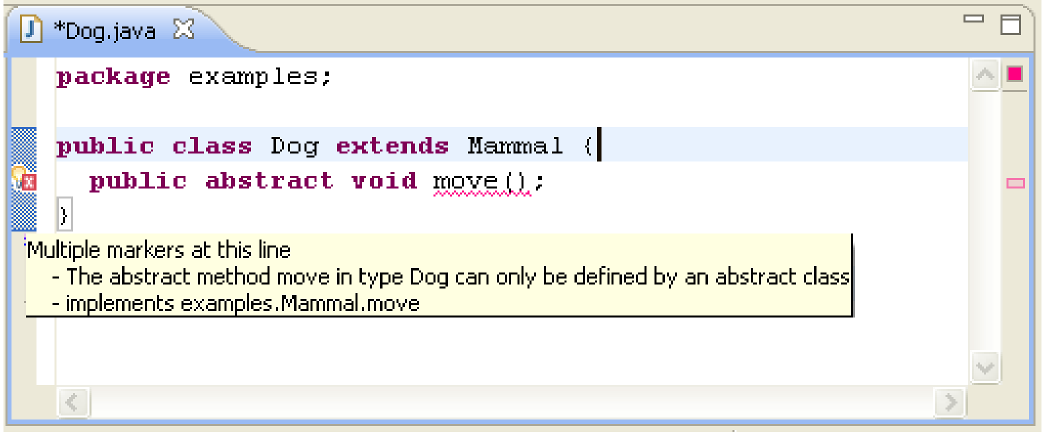
Now click **Finish**.

The Dog.java class that opens looks fine, but let's play around with it a little more. (We live to experiment!) Edit **Dog** by adding the **blue** code as shown:

CODE TO TYPE:

```
package examples;

public class Dog extends Mammal {
    public abstract void move();
}
```

Java didn't like this:



```
package examples;

public class Dog extends Mammal {
  public abstract void move();
}
```

Multiple markers at this line
- The abstract method move in type Dog can only be defined by an abstract class
- implements examples.Mammal.move

If an abstract class is *subclassed*, the subclass must either implement all of the abstract methods of its parent, *or* declare itself abstract.

When we design the abstract mammal class, we include the variables and methods that define a mammal. Determining which aspects to exclude from the mammal class depends on whether the superclass is **java.lang.Object**, or if the superclass is part of the concept hierarchy (the set of concepts arranged in the tree structure).

Let's take a look at some sample code. Edit **Mammal** by adding the **blue** code as shown:

CODE TO TYPE:

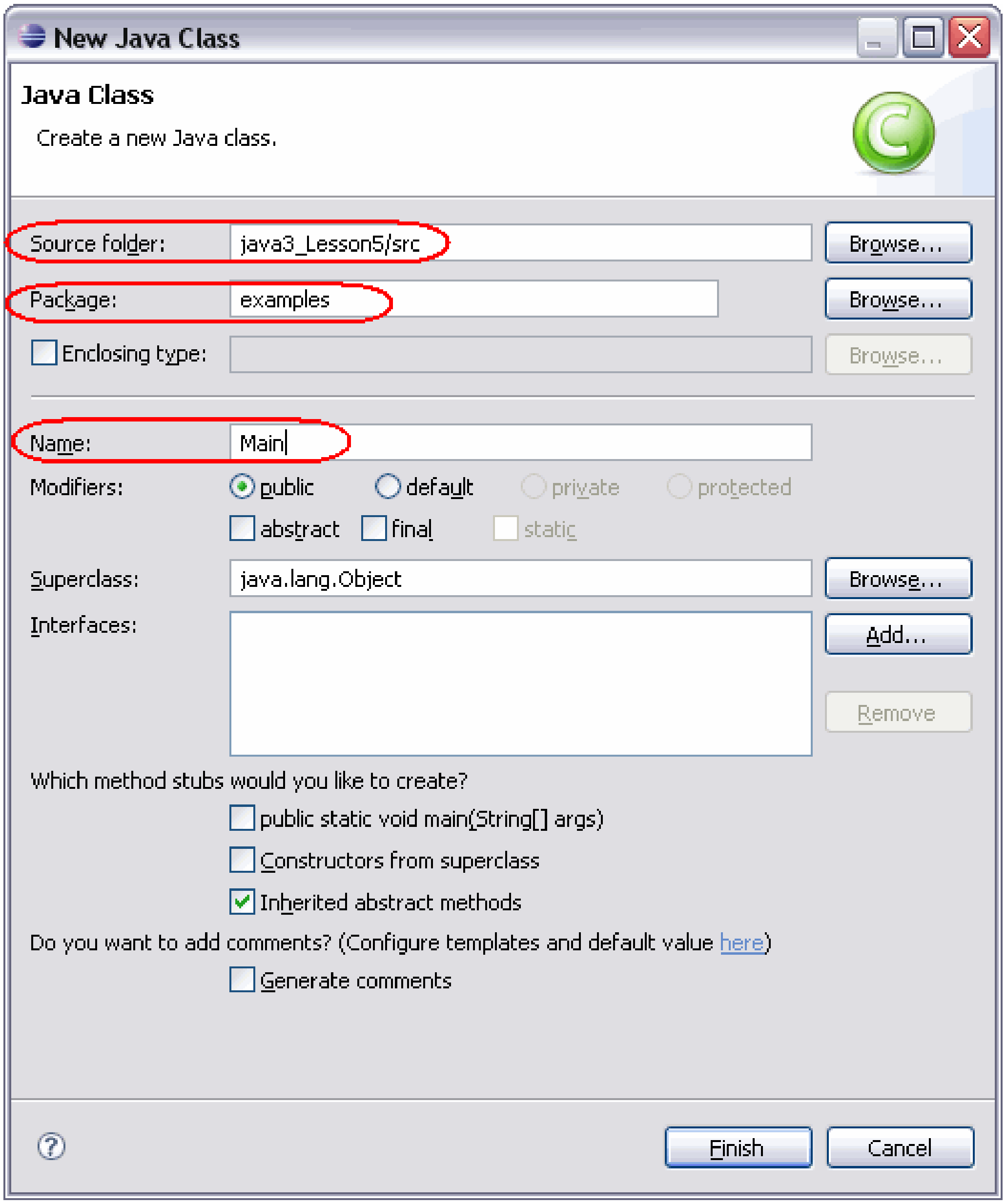
```
package examples;

public abstract class Mammal {
    boolean hasHair = true;
    String breathes = "oxygen";
    String skeletalStructure = "backbone";
    String gender;

    public Mammal(String sex){
        gender = sex;
        System.out.println("I am a " + gender + " dog");
    }

    // Depending on hierarchy, you might have this abstract method in "animal"
    // Since we show inheritance from Object, we are safe here.
    // Mammals move differently, so this is a differentiation (some 2 legs, some 4)
    public abstract void move();

    // all mammals give birth to live young, but the methods may be different.
    // Shhh, we know about the platypus. Do you want to type more?
    public abstract void liveBirth();

    public void feedYoung(){  // this one is specific to mammals
        String food = "milk";
        System.out.println("Since I am " + gender + ", ");
        if (gender =="female")
            System.out.println("I provide my young with " + food);
            // the content of the method could say how
            // depending on your level of specificity, this could be abstract too
        else
            System.out.println("I need assistance to feed my young " + food);
    }

    public boolean hasMammaryGlands(){
        return true;
    }

    public abstract void eat();
}
```

Now, let's go back to our subclass of **Mammal**. Edit **Dog** by adding the **blue** code as shown:

CODE TO TYPE:

```
package examples;

public class Dog extends Mammal {
    private boolean domesticated = true;

    public Dog(String sex){
        super(sex);
    }

    public  void move(){
        System.out.println("We move on all 4 legs.");
    }

    public boolean isDomesticated() {
        return domesticated;
    }

    public void liveBirth(){
        System.out.println("in litters, very cute");
    }

    public void eat(){
        System.out.println("With my sharp teeth. Anything I can get—except lettuce");
    }
}
```

In the examples package in the java3_Lesson05 project, create a new class and name it Main:



Java Class

Create a new Java class.

Source folder: java3_Lesson5/src [Browse...]

Package: examples [Browse...]

□ Enclosing type: [Browse...]

Name: Main

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

Edit **Main** by adding the **blue** code as shown:

CODE TO TYPE:

```
package examples;

public class Main {

    public static void main(String [] args){
        Dog myDog = new Dog("female");
        System.out.println("I am domesticated: " + myDog.isDomesticated());
        myDog.feedYoung();   // inherit from super
        if (myDog.gender == "male")
         System.out.print("My offspring come: ");
        else
         System.out.print("I give birth: ");
        myDog.liveBirth();   // implemented abstract method of super
    }
}
```

Save **Mammal**, **Dog**, and **Main**, then run **Main**.

Modify **Main** by adding the **blue** code and removing the **red** code as shown:

CODE TO TYPE:

```
package examples;

public class Main {

    public static void main(String [] args){
        Dog myDog = new Dog("male female");
        System.out.println("I am domesticated: " + myDog.isDomesticated());
        myDog.feedYoung();   // inherit from super
        if (myDog.gender == "male")
            System.out.print("My offspring come: ");
        else
            System.out.print("I give birth: ");
        myDog.liveBirth();   // implemented abstract method of super
    }
}
```

Save and run it again. Your output should change accordingly.

The API and abstract

Click the API icon under the Eclipse menu bar, go to the **java.awt** package, and scroll down to the **Graphics** class. The first line looks like this:

public abstract class Graphics extends Object

Read through the description of the Graphics class. Now scroll down to its Method Summary. The left panel beside the method signatures indicates that the methods are almost all **abstract**. Graphics methods are implemented depending on context. And, as the API's **Constructor Summary** says, "Since **Graphics** is an abstract class, applications cannot call this constructor directly. Graphics contexts are obtained from other graphics contexts or are created by calling **getGraphics** on a component."

Abstract classes provide programmers with conceptual power and control over subclasses. Classes may have multiple uses (through subclasses), but stipulations in the **abstract** class definition provide core aspects of the objects. These stipulations:

- make inheritance stronger.
- enhance each subclass of a super.
- allow designers to focus on conceptual aspects of a class.

Making Your Own Figures and Shapes

Okay, we're getting dangerously close to working on the project that you'll hand in to your instructor for this lesson. Here's an example of the project you're about to create.

Familiarize yourself with the various action buttons here. This particular example has only two graphical objects. You'll add more to your project later. Draw a figure and then move that figure around with your mouse. Each drawing is an individual **instance**.

In the previous lesson's project, we asked you to create a Shape class and some other classes that descended from it. As you make decisions about shapes you want to create, consider these questions: How would you draw a generic Shape object? What is a Shape object? Can we actually define a Shape without knowing which kind of Shape it is?

We have a few ideas to share with you about a Shape class that we envision. But we won't put them into a listing now. You'll decide later whether you want to use our design or keep your own. You might even decide to blend the two.

OBSERVE: Our Shape class

```
package bigproject.shapes;

import java.awt.Color;
import java.awt.Graphics;

public abstract class Shape{
    private int x, y;
    private Color lineColor;

    public Shape(int x, int y, Color lineColor) {
        this.x = x;
        this.y = y;
        this.lineColor = lineColor;
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }

    public Color getLineColor() {
        return lineColor;
    }

    public void setLineColor(Color lineColor) {
        this.lineColor = lineColor;
    }
}
```

In our variables, we defined a Shape as only an **x and y location** and a **lineColor**. We envision that ALL of our shapes will have a starting point and will have a line color. So, can we give a generic Shape object a width and a height? We could if *all* descendants of this class had those attributes. But our Line object will have either width or height, but not both, so we cannot give a generic Shape object a width and a height here.

In the next project, you'll decide the kind of abstract attributes and/or methods to put into your Shape class.

Note Whether it is abstract or not, a class should contain or inherit everything that ALL of its descendants will have, but should not contain anything that is not shared by ALL of its descendants.

For now, we'll show you how we envision a Rectangle and an Oval. Let's take a look at our Rectangle class first.

OBSERVE: Our Rectangle class

```
package bigproject.shapes;

import java.awt.Color;
import java.awt.Graphics;

public class Rectangle extends Shape {
    private Color fillColor;
    private int width, height;
    private boolean fill;

    public Rectangle(int x, int y, int w, int h, Color lineColor, Color fillColor, boolean fill) {
        super(x, y, lineColor);
        this.width = w;
        this.height = h;
        this.fillColor = fillColor;
        this.fill = fill;
    }

    // Getters and setters.
    public Color getFillColor() {
        return fillColor;
    }

    public void setFillColor(Color fillColor) {
        this.fillColor = fillColor;
    }

    public int getWidth() {
        return width;
    }

    public void setWidth(int width) {
        this.width = width;
    }

    public int getHeight() {
        return height;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public void setFill(boolean fill) {
        this.fill = fill;
    }

    public boolean isFill() {
        return fill;
    }

   /**
    * Returns a String representing this object.
    */
    public String toString() {
        return "Rectangle: x = " + getX() + " y = " + getY() + " w = " + getWidth() + " h = " + getHeight();
    }
}
```

How does our rectangle differ from our Shape class? We didn't repeat any information from our Shape class. In the Shape class, we provided getters and setters for our sub-classes to use in getting and setting attributes inherited from the Shape class. We could have made those attributes protected instead of private, but because we are creating highly encapsulated objects here, we'll keep them private.

In the Rectangle constructor, we pass the **x, y, and lineColor** up to the Shape class object (the parent of this object), and in doing so, set those parameters.

OBSERVE: Our Oval Class

```
package bigproject.shapes;

import java.awt.Color;
import java.awt.Graphics;

/*
 * In Java, the only difference between a Rectangle and an Oval is the drawing method.
Both are
 * represented by an x, y, width, and height. Therefore, we can save typing by just ove
rriding the
 * Rectangle's draw(Graphics g) method to draw an oval instead of a rectangle.
 */
public class Oval extends Rectangle {

    /**
     * Constructor.  Just passes the params to the Rectangle constructor.
     */
    public Oval(int x, int y, int w, int h, Color lineColor, Color fillColor, boolean f
ill) {
        super(x, y, w, h, lineColor, fillColor, fill);
    }

    /**
     * Returns a String that represents this object.
     */
    public String toString() {
        return "Oval: x = " + getX() + " y = " + getY() + " w = " + getWidth() + " h =
" + getHeight();
    }
}
```

In Java, there is no difference between an Oval and a Rectangle, except for the way they're drawn, so the Oval class needs very minimal information, passing all of the parameters to the superclasses Rectangle and Shape. We've purposely omitted the **draw()** method in these listings to give you a hint about your next project.

Design Considerations for a Graphics Tool

We'll begin by using only two-dimensional figures, but even so, we'll need to give serious consideration to the hierarchy we'll create. Since we are going to represent graphical objects, let's go over a bit of the mathematical terminology we'll be using as well:

- In an *open figure* points, lines, and curves do not start and end at the same point.
- In a *closed figure*, the geometric shape starts and ends at the same point, and there is no way into the interior of the object from outside of the object without crossing the lines that comprise it.

Let's follow the upper levels (parent classes) as they move down to the lower levels (the specific objects). This is known as a *top-down* design. Draw (use a paint program on your computer) an inheritance hierarchy. Be as thorough and specific as you can. Here's one possible hierarchy. You are not required to use this particular hierarchy, it's just an example. Moving your mouse around in the drawing tool is called a *squiggle*, although a better name might be *freeStyle*. And since the *squiggle* is located under *Open Figures* in the example inheritance tree, we do not connect the ends. In this case, even if the ends were connected, the design of the class would treat the figure as an open figure.

The alternative to *top-down* design is *bottom-up*. In bottom-up design, we begin with the Java classes already *available*, the classes that we will actually use to *draw* the figures, and work our way up. And where will we find those classes? In the API, of course!

Open the API and go to the **java.awt.Graphics** class. Look over the **methods** available for drawing **figures**. Most of them start with *draw*, but there is no **drawCircle** or **drawSquare**. Since those specific methods don't exist, are there methods available that can be used to get the effect we want?

Before you continue, consider these questions:

- Do we have a candidate or candidates for an abstract class?
- What aspects would be shared by all subclasses and so should be located within the abstract class(es)?
- What *kinds of shapes* (subclasses) do we want to create?
- Where would methods such as *getArea()* and *getCircumference()* go?

Keep these questions in mind as you work through the project for this lesson. I'm looking forward to seeing what you come up with for your project. Keep up the good work and see you in the next lesson!

Interfaces: Listeners and Adapters

Interfaces in Java

In previous lessons, we discussed classes and subclasses, and the way inheritance works within them. Java allows only single inheritance, which means a class can inherit from just one parent class. Through *interfaces* though, we can allow classes to provide capabilities beyond parental inheritance.

In this lesson as well as lesson 7, we'll continue to work on the drawing applet. We're going to enter plenty of code, so settle in and get comfortable!

Model-View-Controller Architecture

Our drawing project, like any Java project, will require lots of code. We want to make sure our code is easy to follow and more importantly, easy to maintain. Ideally, we'll incorporate the separation of functionality in our code to keep it clean and simple.

One tool that programmers use to accomplish those goals is the "Model-View-Controller" (MVC) architecture. It separates the elements that users see from the logic that controls those elements. MVC architectural design separates code functionality into three parts:

- **Model**: the business logic of the program.
- **View**: the GUI or user interface.
- **Controller**: the part of the program that tells the model what to do.

The model is the gateway. It contains information that determines the state of all elements in the program. The view asks the model what to display. The controller tells the model how to change its state. Everything goes through the model.

We'll discuss this more later, but at OST, we like to learn by **doing**, so let's start by creating a project to help illustrate the MVC architecture.

An Example: Drop-Down Lists (Choice Components)

Our example will demonstrate how to create a GUI to use in conjunction with MVC architecture. One GUI component option we have for our project is the drop-down list, or *java.awt.Choice* component. When you want the user to select one from a number of options, the drop-down menu is a good choice because it uses less space than radio buttons.

Create a new project in the Java3_Lessons working set, named **ChoiceExample**. In the **ChoiceExample** project, create a new package named **view**.

Create a new class in the view package named **ChoiceApplet** that extends **java.applet.Applet**. Add the code shown in **blue**:

CODE TO TYPE: ChoiceApplet

```
package view;

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

public class ChoiceApplet extends Applet{

    public void init() {
    }

    public void paint(Graphics g) {
        g.drawString("Message will go here.", 20,100);
    }
}
```

Save and run it. Our program doesn't do much yet.

Now we'll build a panel to put on the applet so we can add a Choice component. Then we'll be able to set a message that will be put on the applet.

In the **view** package, create a new class named **ChoicePanel** that extends **java.awt.Panel**. Add the code shown in **blue**:

CODE TO TYPE: ChoicePanel

```
package view;

import java.awt.Choice;
import java.awt.Panel;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;

public class ChoicePanel extends Panel{
    Choice selection;

    public ChoicePanel() {
        selection = new Choice();
        selection.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
            }
        });
        this.add(selection);
    }
}
```

OBSERVE: ChoicePanel

```
package view;

import java.awt.Choice;
import java.awt.Panel;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;

public class ChoicePanel extends Panel{
    Choice selection;

    public ChoicePanel() {
        selection = new Choice();
        selection.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
            }
        });
        this.add(selection);
    }
}
```

In our code, we set up a **Choice** component and gave it an **ItemListener** via an anonymous inner class. The listener method **itemStateChanged()** doesn't do anything yet. We'll get to that later. The **itemStateChanged()** method will be part of the Controller in this applet's Model, View, Controller design pattern.

Save it. Now, we'll add a ChoicePanel instance to our ChoiceApplet, as shown in **blue**:

CODE TO EDIT: ChoiceApplet

```
package view;

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

public class ChoiceApplet extends Applet{

    public void init() {
        ChoicePanel choicePanel = new ChoicePanel();
        this.add(choicePanel);
    }
    public void paint(Graphics g) {
        g.drawString("Message will go here.", 20,100);
    }
}
```

Save and run it. Now the applet has a drop-down box. Currently there are no choices there to select. We have the View of the program, which consists of the applet and its GUI components. We have the beginnings of the Controller in the ItemListener we added to the Choice component on the ChoicePanel. Now, we'll build the Model of the program and connect it to the View and the Controller.

In the **ChoiceExample** project, create a new package named **model**. In the model package, create a new class named **Model**. Add the code shown in **blue**:

CODE TO TYPE: Model

```

package model;

import java.awt.Container;
import view.ChoiceApplet;

public class Model {

    private Container view;
    private String message;

    public static String[] selections = {"The Beatles", "John", "Paul", "George"
, "Ringo"};

    public Model(Container view) {
        this.view = view;
        message = selections[0];
    }

    public void setMessage(String msg) {
        this.message = msg;
    }

    public String getMessage() {
        return this.message;
    }

    public void repaint() {
        view.repaint();
    }
}
```

Save it. Now, in your ChoicePanel, add the code shown in **blue**:

CODE TO EDIT: ChoicePanel

```
package view;

import java.awt.Choice;
import java.awt.Panel;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import model.Model;

public class ChoicePanel extends Panel{

    Model model;
    Choice selection;

    public ChoicePanel(Model mdl) {
        model = mdl;
        selection = new Choice();
        for(String msg : Model.selections) {
            selection.add(msg);
        }
        selection.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                model.setMessage(selection.getSelectedItem());
                model.repaint();
            }
        });
        this.add(selection);
    }
}
```

Save it. Now, in your ChoiceApplet, add the code shown in **blue** and remove the code in **red**:

CODE TO EDIT: ChoiceApplet

```
package view;

import java.applet.Applet;
import java.awt.Graphics;
import model.Model;


public class ChoiceApplet extends Applet{
    Model model;
    ChoicePanel choicePanel;

    public void init() {
        model = new Model(this);
        ChoicePanel choicePanel = new ChoicePanel(model);
        this.add(choicePanel);
    }

    public void paint(Graphics g) {
        g.drawString("Message will go here."model.getMessage(), 20,100);
    }
}
```

Save and run it. Now the selection from the drop-down Choice component appears in the applet.

OBSERVE: ChoicePanel

```
package view;

import java.awt.Choice;
import java.awt.Panel;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import model.Model;

public class ChoicePanel extends Panel{

    Model model;
    Choice selection;

    public ChoicePanel(Model mdl) {
        model = mdl;
        selection = new Choice();
        for(String msg : Model.selections) {
            selection.add(msg);
        }
        selection.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                model.setMessage(selection.getSelectedItem());
                model.repaint();
            }
        });
        this.add(selection);
    }
}
```

We filled the **Choice** component, **selection**, using the enhanced **for** loop. It looks at the **Model** class static variable, **selections**, and gets each item in order, placing them in the loop's local **msg** variable. That variable is added to the **selection** **Choice** component, within the loop.

Now, let's add the ability to reset the **Choice** component to its original state. We will use a facility commonly called a "callback." We'll create an interface with a method that most of our classes will implement to ensure that they all have the same method available to reset the component.

Note Unlike some other languages, Java does not have the ability to pass the memory address of a method to a method call. Instead, we implement an interface so we know that a method exists. Other classes can then "call back" to that method when it is needed. That's why the facility is referred to as a **callback**.

Create a new package named **interfaces** in the **ChoiceExample** project. In the interfaces package, create a new interface named **Resettable**. Add the code in **blue** as shown:

CODE TO TYPE: Resettable

```

package interfaces;

public interface Resettable {

    public void resetComponents();

}
```

Save it.

In the view package, create a new class named **ButtonPanel** that extends **java.awt.Panel**. Then add the code in **blue** as shown:

CODE TO TYPE: ButtonPanel

```
package view;

import java.awt.Button;
import java.awt.Panel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import model.Model;

public class ButtonPanel extends Panel {

    Model model;
    Button resetBtn = new Button("Reset");

    public ButtonPanel(Model mdl) {
        model = mdl;
        resetBtn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                model.resetComponents();
            }
        });
    this.add(resetBtn);
    }
}
```

Save it. There will be an error present because we haven't implemented the **resetComponents()** method in the Model class yet. We'll take care of that in a minute.

Edit the **ChoicePanel** to implement the Resettable interface, adding the code in **blue** as shown:

CODE TO EDIT: ChoicePanel

```
package view;

import java.awt.Choice;
import java.awt.Panel;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import model.Model;
import interfaces.Resettable;

public class ChoicePanel extends Panel implements Resettable{

    Model model;
    Choice selection;

    public ChoicePanel(Model mdl) {
    model = mdl;
    selection = new Choice();
    for(String msg : Model.selections) {
        selection.add(msg);
    }
    selection.addItemListener(new ItemListener() {
        public void itemStateChanged(ItemEvent e) {
            model.setMessage(selection.getSelectedItem());
            model.repaint();
        }
    });
    this.add(selection);
    }

    public void resetComponents() {
        selection.select(0);
        model.setMessage(selection.getSelectedItem());
    }
}
```

Save it. Go ahead and edit the **Model** to implement Resettable in **blue** as shown:

CODE TO EDIT: Model

```
package model;

import java.awt.Container;
import view.ChoiceApplet;
import interfaces.Resettable;

public class Model implements Resettable{

    private Container view;
    private String message;

    public static String[] selections = {"The Beatles", "John", "Paul", "George"
, "Ringo"};

    public Model(Container view) {
        this.view = view;
        message = selections[0];
    }

    public void setMessage(String msg) {
        this.message = msg;
    }

    public String getMessage() {
        return this.message;
    }

    public void repaint() {
        view.repaint();
    }

    public void resetComponents() {
        //cast view to a Resettable type in order to see resetComponents().
        ((Resettable)view).resetComponents();
        repaint();
    }
}
```

Save it. We get an error now because we haven't made ChoiceApplet implement the Resettable interface. Let's edit **ChoiceApplet** now to implement Resettable, as shown in **blue**:

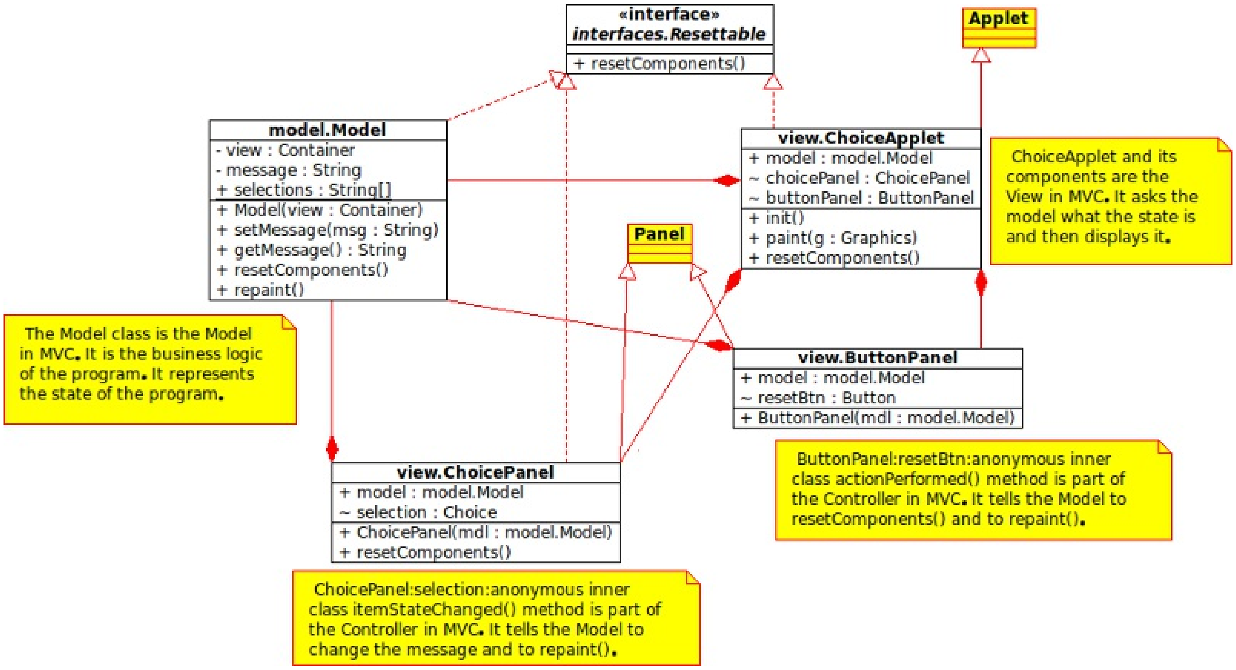
CODE TO EDIT: ChoiceApplet

```
package view;

import java.applet.Applet;
import java.awt.Graphics;
import model.Model;
import interfaces.Resettable;

public class ChoiceApplet extends Applet implements Resettable{
    Model model;
    ChoicePanel choicePanel;
    ButtonPanel buttonPanel;

    public void init() {
        model = new Model(this);
        choicePanel = new ChoicePanel(model);
        buttonPanel = new ButtonPanel(model);
        this.add(choicePanel);
        this.add(buttonPanel);
    }

    public void paint(Graphics g) {
        g.drawString(model.getMessage(), 20,100);
    }

    public void resetComponents() {
        choicePanel.resetComponents();
    }
}
```

Save and run it. Change the Choice component to some other value. Click **Reset**. The model is reset to its original state and the applet is repainted.

Chain of events: The reset button is pressed, activating the ActionListener's **actionPerformed()** method. The **actionPerformed()** method calls the Model's **resetComponent()** method, which in turn calls the ChoiceApplet's **resetComponents()** method, which in turn calls the ChoicePanel's **resetComponents()** method, setting the Choice component to its original state and telling the Model to set its message to its original state. Everything we do goes through the Model.

Let's take a look at a diagram that shows how various elements in our example are connected to one another. This is called a **UML Class Diagram**. **UML** is the **Unified Modeling Language** and the **Class Diagram** illustrates the way various classes interact.



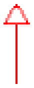
ChoiceApplet and its components are the View in MVC. It asks the model what the state is and then displays it.

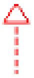
The Model class is the Model in MVC. It is the business logic of the program. It represents the state of the program.

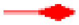
ButtonPanel:resetBtn:anonymous inner class actionPerformed() method is part of the Controller in MVC. It tells the Model to resetComponents() and to repaint().

ChoicePanel:selection:anonymous inner class itemStateChanged() method is part of the Controller in MVC. It tells the Model to change the message and to repaint().

Let's break down our diagram. First, the **+** symbol inside the class boxes means that the method or attribute is public. The **~** symbol means that the attribute or method is private.

The red lines in the diagram have specific meanings as well. A solid line with an arrow at the end of it  represents inheritance. The arrow points to a super class.

A dotted line with an arrow at the end of it  indicates implementation of an interface. The arrow points to the interface being implemented.

Finally, the solid line with a diamond at the end of it  indicates association, or "uses." The diamond is on the end of the line closest to the class that is using the other class.

So, our diagram shows us that ChoiceApplet inherits from Applet, implements Resettable, and uses Model, ChoicePanel, and ButtonPanel.

All of the View components, ChoiceApplet, ButtonPanel, and ChoicePanel, use the Model. They either get information from the Model to display it, or the Controller portion of those components tell the Model how to change its state.

Creating the Shape Drawing Project

In the **Java3_Lessons** working set, create a new Java project named **java3_Lesson06**. Add the following packages to the project:

- event
- interfaces
- model
- shapes
- ui.applet
- ui.panels

We'll need these packages to *modularize* our code and make it easy to maintain. We'll talk about this in greater detail later, but let's build something right now!

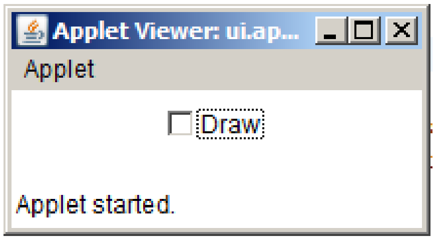
In the **ui.applet** package of your **java3_Lesson06** project, create a new class named GUIDemo. The superclass should be **java.applet.Applet**.

Edit **GUIDemo** in **blue** as shown :

CODE TO TYPE: GUIDemo

```
package ui.applet;
import java.applet.Applet;
import java.awt.*;

public class GUIDemo extends Applet {

    private final String DRAW = "Draw";

    public void init(){
        Checkbox draw = new Checkbox(DRAW);
        add(draw);
    }
}
```

Save and run it (as a Java Applet). This sweet little applet opens:



Click the checkbox so the check mark appears. Let's examine that code:

OBSERVE: GUIDemo

```
package ui.applet;
import java.applet.Applet;
import java.awt.*;

public class GUIDemo extends Applet {
    private final String DRAW = "Draw";

    public void init(){
        Checkbox draw = new Checkbox(DRAW);
        add(draw);
    }
}
```

First, we declare a **final String constant** named **DRAW** to represent action, Draw. Using the upper-case name DRAW, helps avoid confusion when we use the word "Draw" later.

Within the **init()** method, we create a **Checkbox** named **draw** as a local variable and give it the label represented by the **DRAW** constant. Then we **add** the **draw** object to the applet.

When you click in the checkbox, the check mark appears—but nothing else happens. Clicking in the box component causes an **ItemEvent**, but no one is *listening* for it. If an **ItemEvent** takes place in the program, but there is no one there to hear it, does it really happen at all? Nope. We need to delegate a *listener* to listen for that **ItemEvent** (in this case, a click in the checkbox).

Note Each Java GUI component allows a user to generate specific Events, which require relative listeners to "hear" them. The action taken in response to those events is part of the **Controller** in the MVC design pattern.

Interfaces and Listeners

Your applet is an Interface. A Java interface is similar to the face of a car radio. Car radio faces have evolved over the years:

Then:

Now:

But despite a more modern appearance, they still operate in essentially the same way they always have. Most radio interfaces still have knobs for power, volume, and tuning. Good new design incorporates interfaces that are familiar to users. While specific internal behavior that's triggered by the use of interfaces may vary, interfaces themselves usually change only slightly.

In Java, *Listeners* are interfaces that *listen* for specific types of events to occur.

API Open the API and go to the **java.awt** package. Scroll down to **Checkbox** and click it. Scroll down to its methods and you'll see **addItemListener(ItemListener l)**. Conveniently, in order to add a listener, we always use the command **add*xxx*()**, where *xxx* is the name of the listener. So, if we need an **ActionListener**, we would use the **add*ActionListener*()** method.

Let's get that **Listener** into our code! Edit **GUIDemo** as shown in **blue**:

CODE TO TYPE: GUIDemo

```
package ui.applet;

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

public class GUIDemo extends Applet implements ItemListener {
    private final String DRAW = "Draw";
    Checkbox draw;

    public void init(){
        draw = new Checkbox(DRAW);
        add(draw);
        draw.addItemListener(this);
    }

    public void itemStateChanged(ItemEvent e){
        System.out.println("I see you now!");
        if (e.getSource() == draw)
            System.out.println("I know you clicked " + e.getItem().toString());
    }
}
```

Save and run it. Now, when you click the box you see output in the console. How did we do that? Let's take a look at the code:

OBSERVE:

```
package ui.applet;

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

public class GUIDemo extends Applet implements ItemListener{
    private final String DRAW = "Draw";
    Checkbox draw;

    public void init(){
        draw = new Checkbox(DRAW);
        add(draw);
        draw.addItemListener(this);
    }

    public void itemStateChanged(ItemEvent e){
        System.out.println("I see you now!");
        if (e.getSource() == draw)
            System.out.println("I know you clicked " + e.getItem().toString());
    }
}
```

We make **draw** an instance variable so we can use it throughout the class rather than just in the **init()** method.

In the **init()** method, we add **this** instance of the GUIDemo class to the **draw** object with **addItemListener()**.

When the user changes the state of the **draw Checkbox**, the **itemStateChanged()** method is invoked, because **this** instance of GUIDemo is the **ItemListener** for the **draw Checkbox**.

The **if** statement looks at the **ItemEvent e** parameter and checks to get the object that was the source of the event. If that source is our **drawCheckbox**, we print a String to the console with the label of the **Checkbox** appended to it.

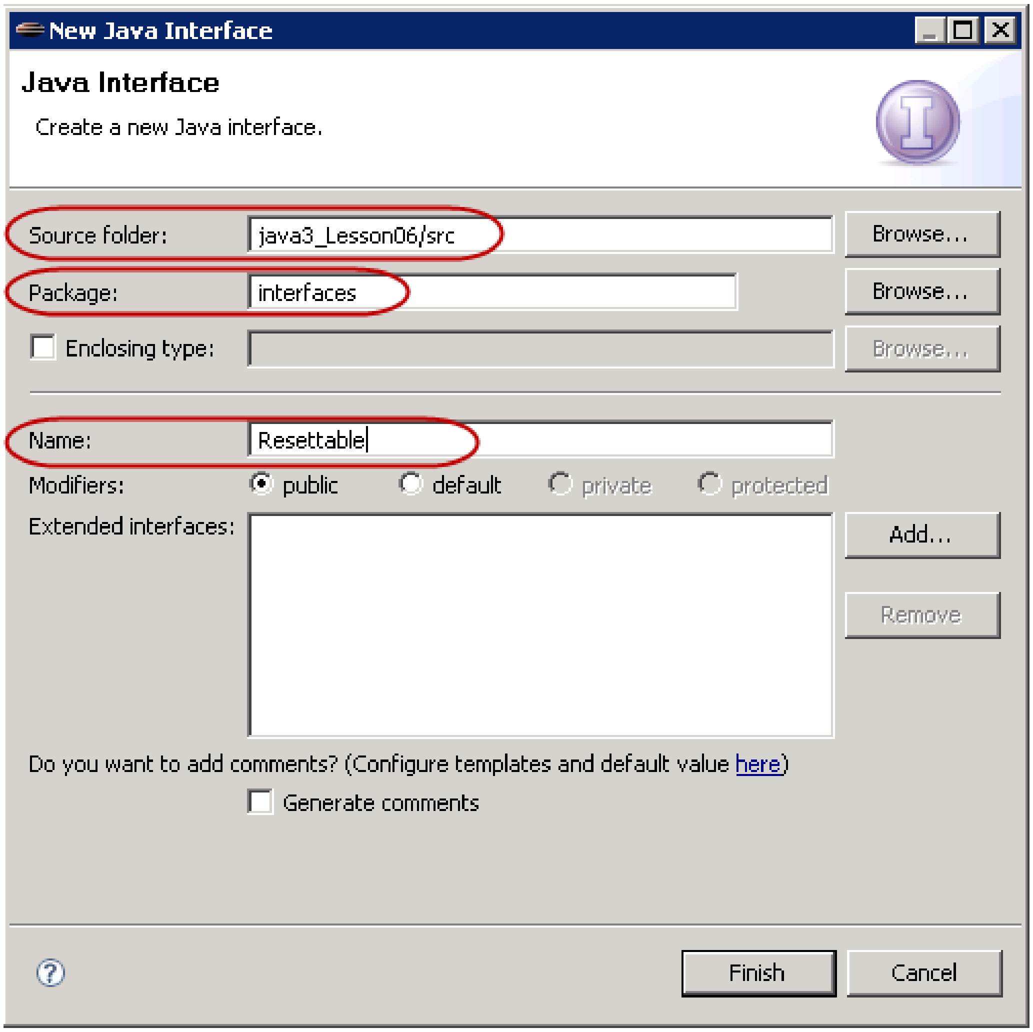
Here are a few guidelines for you to keep in mind when using a listener interface:

- Import **java.awt.event.*** and **java.awt.***.
- Be sure the class declaration **implements** the listener.
- Create the GUI component that you want to be *heard*.
- Add the GUI component to the Applet.
- Add the listener to the GUI component.
- Implement all of the methods specified by the Interface/Listener.

Building a Program

So, where is all of this going? We'll use this information to start modularizing our code.

In the **interfaces** package of your **java3_Lesson06** project, create a new interface named **Resettable**.



Java Interface

Create a new Java interface.

Source folder: java3_Lesson06/src

Package: interfaces

☐ Enclosing type:

Name: Resettable

Modifiers: ● public ○ default ○ private ○ protected

Extended interfaces:

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

☐ Generate comments

Type the **Resettable** interface as shown in **blue**:

CODE TO TYPE:

```
package interfaces;

public interface Resettable {
  public void resetComponents();
}
```

Save it. We can't run this right now, but we'll use it later.

OBSERVE: Required Method

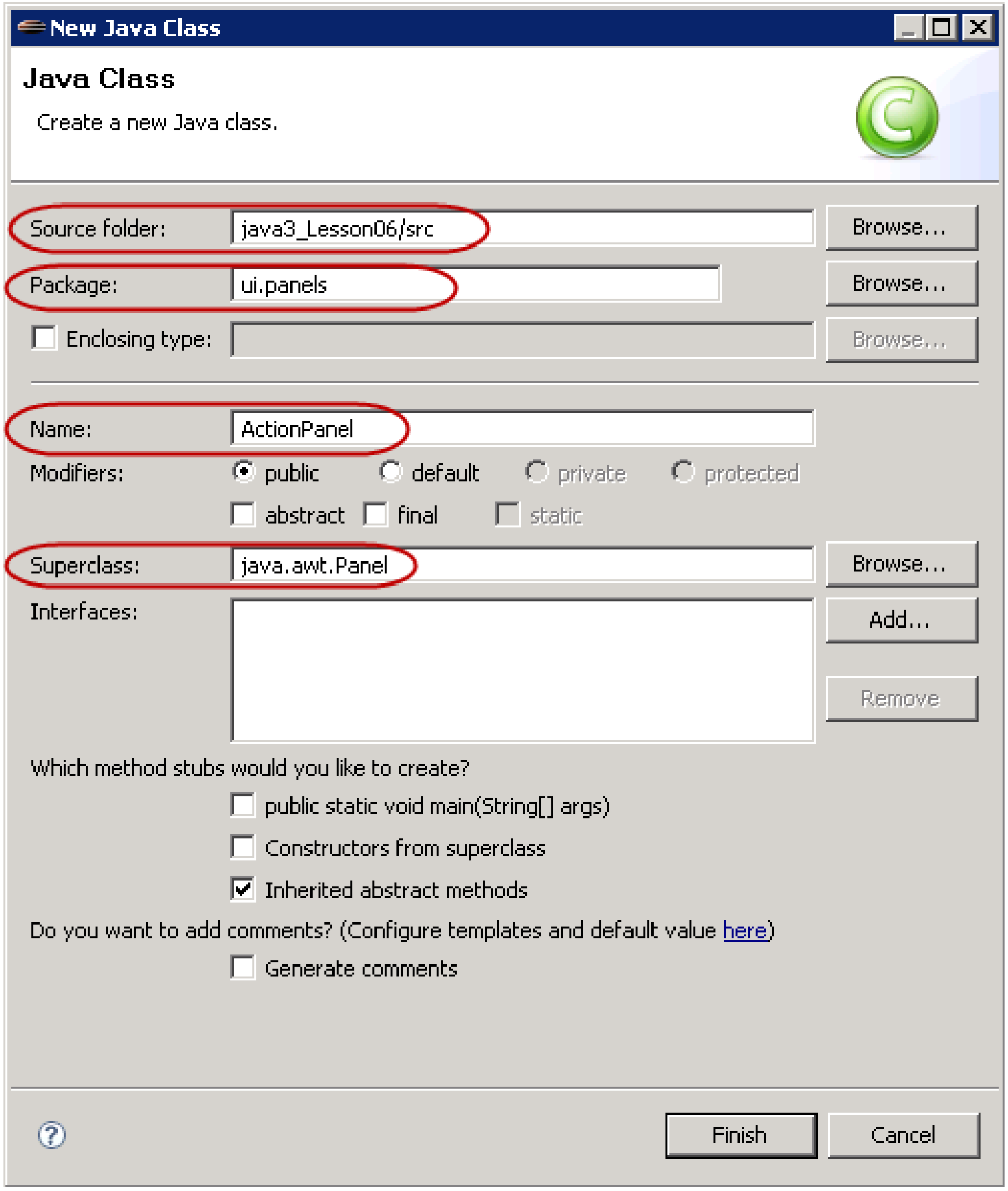
```
package interfaces;

public interface Resettable {
  public void resetComponents();
}
```

Any class that implements **Resettable** must implement **resetComponents()**. Because we know **resetComponents()** will be present in those classes, we can call the method when it's needed. As we go through the design process for our Big Project, you'll appreciate the importance of including the **resetComponents()** method.

Save and close the **Resettable** interface now; we won't need to edit it any more. Interfaces rarely need to be changed once they are finalized.

So, we have an interface that *guarantees* we have the method necessary to clear components on panels for any class that implements it. Now let's begin the division of our project by creating a panel to hold our GUI components. Usually, GUIs are built by *layering* containers and components, and applying different *Layout Managers* to the containers.

In the **ui.panels** package of your **java3_Lesson06** project, create a new class named **ActionPanel**. Its superclass will be **java.awt.Panel**.



Java Class

Create a new Java class.

Source folder: java3_Lesson06/src

Package: ui.panels

□ Enclosing type:

Name: ActionPanel

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.awt.Panel

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 **ActionPanel** class in **blue** as shown:

CODE TO TYPE: ActionPanel

```
package ui.panels;

import interfaces.Resettable;
import java.awt.Checkbox;
import java.awt.CheckboxGroup;
import java.awt.Panel;

public class ActionPanel extends Panel implements Resettable {

    private CheckboxGroup actionGroup;
    private Checkbox chkDraw, chkMove, chkResize, chkRemove, chkChange, chkFill;

}
```

Do you know why we're seeing the error message? Ignore it for now; we'll fix it soon.

OBSERVE:

```
import interfaces.Resettable;
import java.awt.Checkbox;
import java.awt.CheckboxGroup;
import java.awt.Panel;

public class ActionPanel extends Panel implements Resettable{

    private CheckboxGroup actionGroup;
    private Checkbox chkDraw, chkMove, chkResize, chkRemove, chkChange, chkFill;

}
```

See that first **import statement**? This class must implement the **Resettable** interface so that the components will be reset to their original condition when the user clicks the **Clear** button (which we'll add later).

A **CheckboxGroup** allows us to group **CheckBox** objects together, which has the effect of turning them into "radio buttons." In a group of **Checkbox** objects, only one may be in a **true** state at a time (although, at compile and run-time, none of the objects needs to be in a **true** state). Once one **Checkbox** object is set to true at run-time, one and only one of the objects in the group can be true.

The **CheckBox** objects—**chkDraw**, **chkMove**, **chkResize**, **chkRemove**, **chkChange**, and **chkFill**—will be placed on this panel.

Let's add those **Checkbox** objects to the panel now, as shown in **blue**:

CODE TO EDIT: ActionPanel

```
package ui.panels;

import interfaces.Resettable;
import java.awt.Checkbox;
import java.awt.CheckboxGroup;
import java.awt.GridLayout;
import java.awt.Panel;

public class ActionPanel extends Panel implements Resettable{

    private CheckboxGroup actionGroup;
    private Checkbox chkDraw, chkMove, chkResize, chkRemove, chkChange, chkFill;
    private final String DRAW = "Draw";
    private final String MOVE = "Move";
    private final String RESIZE = "Resize";
    private final String REMOVE = "Remove";
    private final String CHANGE = "Change";
    private final String FILL = "Fill";

    public ActionPanel(){
        actionGroup = new CheckboxGroup();
        chkDraw = new Checkbox(DRAW, actionGroup, true);
        chkMove = new Checkbox(MOVE, actionGroup, false);
        chkResize = new Checkbox(RESIZE, actionGroup, false);
        chkRemove = new Checkbox(REMOVE, actionGroup, false);
        chkChange = new Checkbox(CHANGE, actionGroup, false);
        chkFill = new Checkbox(FILL, false);
        setLayout(new GridLayout(1,6));
        add(chkDraw);
        add(chkMove);
        add(chkResize);
        add(chkRemove);
        add(chkChange);
        add(chkFill);
    }
}
```

Save it.

OBSERVE: ActionPanel Imports and Variables

```
package ui.panels;

import interfaces.Resettable;
import java.awt.Checkbox;
import java.awt.CheckboxGroup;
import java.awt.GridLayout;
import java.awt.Panel;

public class ActionPanel extends Panel implements Resettable {

    private CheckboxGroup actionGroup;
    private Checkbox chkDraw, chkMove, chkResize, chkRemove, chkChange, chkFill;
    private final String DRAW = "Draw";
    private final String MOVE = "Move";
    private final String RESIZE = "Resize";
    private final String REMOVE = "Remove";
    private final String CHANGE = "Change";
    private final String FILL = "Fill";

    public ActionPanel(){
        actionGroup = new CheckboxGroup();
        chkDraw = new Checkbox(DRAW, actionGroup, true);
        chkMove = new Checkbox(MOVE, actionGroup, false);
        chkResize = new Checkbox(RESIZE, actionGroup, false);
        chkRemove = new Checkbox(REMOVE, actionGroup, false);
        chkChange = new Checkbox(CHANGE, actionGroup, false);
        chkFill = new Checkbox(FILL, false);
        setLayout(new GridLayout(1,6));
        add(chkDraw);
        add(chkResize);
        add(chkMove);
        add(chkRemove);
        add(chkChange);
        add(chkFill);
    }
}
```

We added several **final String** fields. (We'll move them to the Model class in a bit, but we've added them here temporarily so we can see them at work with the rest of our code.)

In our example, we create and then add the **Checkbox** objects to the ActionPanel. First, we create a new **CheckboxGroup** object. The **CheckboxGroup** object is a logical container for the first five **Checkbox** objects. This container does not effect the way the objects are laid out on the screen, but it groups them into a set of radio buttons. When we create the **Checkbox** objects, we pass this group to their constructors. The constructor parameters for the first five **Checkbox** objects take the **label** of the checkbox, the **group** to which the checkbox is attached, and the initial **state** of the object—true for set, and false for unset. The last Checkbox object (**chkFill**) is not part of the group, so we omit the group parameter in its constructor call.

The **setLayout(new GridLayout(1,6))** line instructs this panel to arrange components added to it in a grid that is 1 row by 6 columns **(1,6)**. Each component added to the panel will occupy the same amount of space in the grid. Components will be arranged in a GridLayout in the order in which they are added to the panel. The grid will be filled from left to right, and from top to bottom.

Now we need to fulfill the promise we made when implementing the Resettable interface in creating the ActionPanel class. We must implement the **resetComponents()** method. Edit your code as shown in **blue**:

CODE TO EDIT: ActionPanel

```
package ui.panels;

import interfaces.Resettable;
import java.awt.Checkbox;
import java.awt.CheckboxGroup;
import java.awt.GridLayout;
import java.awt.Panel;

public class ActionPanel extends Panel implements Resettable{

    private CheckboxGroup actionGroup;
    private Checkbox chkDraw, chkMove, chkResize, chkRemove, chkChange, chkFill;
    private final String DRAW = "Draw";
    private final String MOVE = "Move";
    private final String RESIZE = "Resize";
    private final String REMOVE = "Remove";
    private final String CHANGE = "Change";
    private final String FILL = "Fill";

    public ActionPanel(){
        actionGroup = new CheckboxGroup();
        chkDraw = new Checkbox(DRAW, actionGroup, true);
        chkMove = new Checkbox(MOVE, actionGroup, false);
        chkResize = new Checkbox(RESIZE, actionGroup, false);
        chkRemove = new Checkbox(REMOVE, actionGroup, false);
        chkChange = new Checkbox(CHANGE, actionGroup, false);
        chkFill = new Checkbox(FILL, false);
        setLayout(new GridLayout(1,6));
        add(chkDraw);
        add(chkMove);
        add(chkResize);
        add(chkRemove);
        add(chkChange);
        add(chkFill);
    }

    public void resetComponents() {
        // For each component, set the state. Only one of the first five can be true.
        chkDraw.setState(true);
        chkMove.setState(false);
        chkResize.setState(false);
        chkRemove.setState(false);
        chkChange.setState(false);
        chkFill.setState(false);
    }
}
```

Save it.

This method sets appropriate default states for all of the **components** on the panel. We're getting close to being able to run our program. But first, we need to modify our GUIDemo applet so it can use this class. We have a lot more to do still, but this will allow us to see results as we go. Edit GUIDemo, adding the code shown in **blue** and removing the code shown in **red**:

CODE TO EDIT: GUIDemo

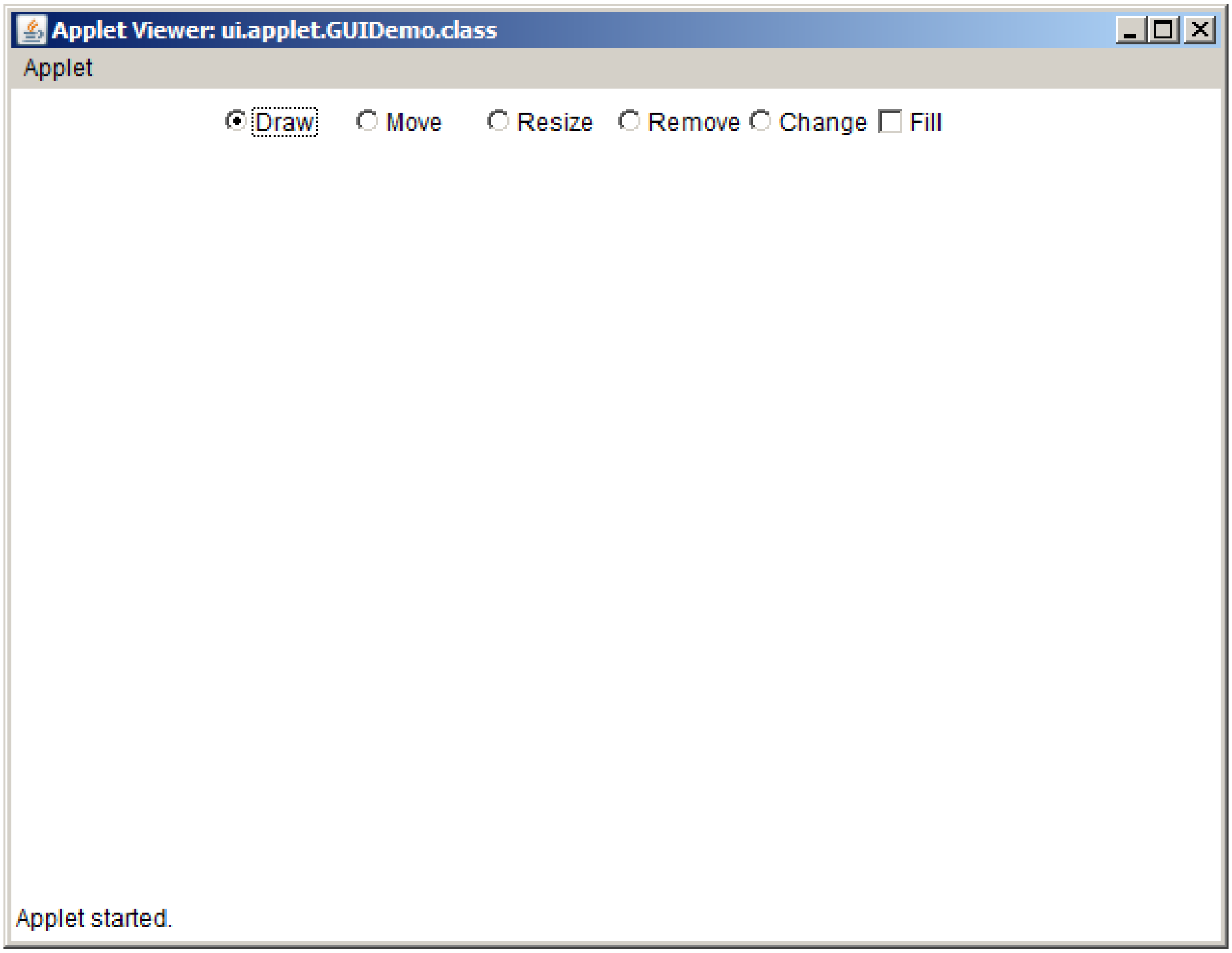
```
package ui.applet;

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import ui.panels.ActionPanel;

    public class GUIDemo extends Applet implements ItemListener{
        private final String DRAW = "Draw";
        Checkbox draw;
        ActionPanel actionPanel;

        public void init(){
            Checkbox draw = new Checkbox(DRAW);
            add(draw);
            draw.addItemListener(this);
            resize(600,400);
            actionPanel = new ActionPanel();
            add(actionPanel);
        }
        public void itemStateChanged(ItemEvent e){
            System.out.println("I see you now!");
            if (e.getSource() == draw)
            System.out.println("I know you clicked " + e.getItem().toString());
        }
}
```

OBSERVE: GUIDemo

```
package ui.applet;

import java.applet.Applet;
import ui.panels.ActionPanel;

public class GUIDemo extends Applet {
    ActionPanel actionPanel;

    public void init() {
        resize(600,400);
        actionPanel = new ActionPanel();
        add(actionPanel);
    }
}
```

Now our GUIDemo applet will look like the listing above. We created a new **ActionPanel** object, added it to the applet, and named it **actionPanel**.

Save all of your files and run the applet:

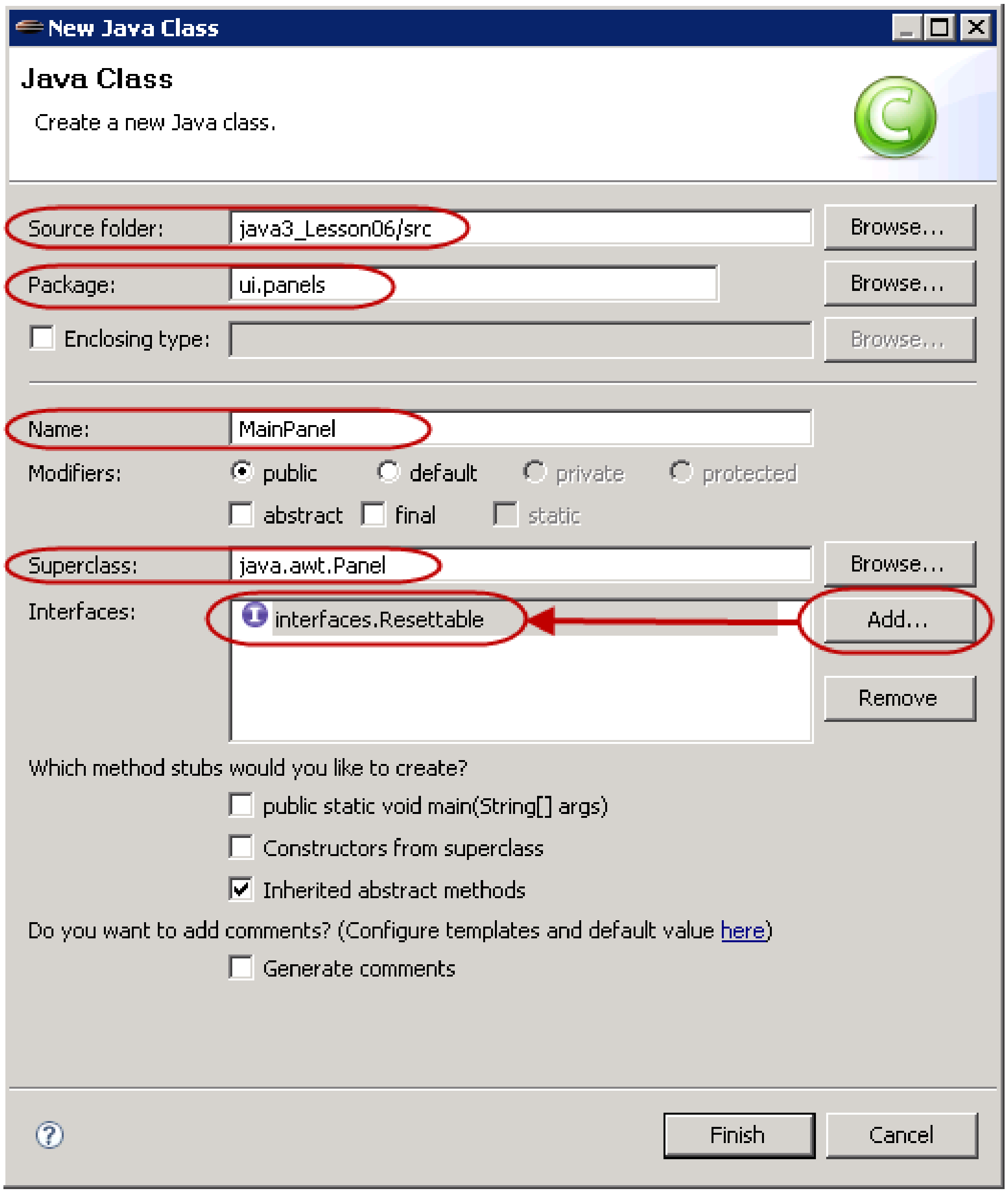


● Draw ○ Move ○ Resize ○ Remove ○ Change □ Fill

Applet started.

You can click on any of the first five checkboxes (which are configured as radio buttons), but only one of them will be selected at a time. The last checkbox (Fill) can be set and unset regardless of the other checkbox states, because we didn't add it to the **CheckboxGroup** object.

We'll incorporate a bit more modularization now, to keep as much of the actual program operation as we can, out of the applet code. Let's create a MainPanel panel to hold the ActionPanel and any other panels we want to place. Afterward, we can add this MainPanel to our applet and all of the other panels will be added automatically. Create a new class named **MainPanel** in the **ui.panels package**. It should extend **java.awt.Panel** and implement the **Resettable** interface, as shown:



Java Class

Create a new Java class.

Source folder: java3_Lesson06/src

Package: ui.panels

□ Enclosing type:

Name: MainPanel

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.awt.Panel

Interfaces: interfaces.Resettable

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

In **MainPanel** add the code in **blue** as shown:

CODE TO TYPE: MainPanel

```
package ui.panels;

import interfaces.Resettable;
import java.awt.GridLayout;
import java.awt.Panel;

public class MainPanel extends Panel implements Resettable {
    ActionPanel actionPanel;

    public MainPanel() {
        actionPanel = new ActionPanel();
        setLayout(new GridLayout(2,1));
        add(actionPanel);
    }

    public void resetComponents() {
        actionPanel.resetComponents();
    }
}
```

OBSERVE: MainPanel

```
package ui.panels;

import interfaces.Resettable;
import java.awt.GridLayout;
import java.awt.Panel;

public class MainPanel extends Panel implements Resettable {
    ActionPanel actionPanel;

    public MainPanel() {
        actionPanel = new ActionPanel();
        setLayout(new GridLayout(2,1));
        add(actionPanel);
    }

    public void resetComponents() {
        actionPanel.resetComponents();
    }
}
```

Here we create a new **ActionPanel** object named **actionPanel** and add it to this panel. We use a **GridLayout** that sets our MainPanel to two rows by one column **(2,1)** in each row. This leaves room to add another panel later.

The **resetComponents()** method is a *pass-through* method that passes the message along to the ActionPanel object. When this method is called, it in turn calls the ActionPanel's **resetComponents()** method.

Save the **MainPanel** class and then modify the **GUIDemo** class, adding the **blue** code and removing the **red** code as shown:

CODE TO EDIT: GUIDemo

```
package ui.applet;

import interfaces.Resettable;
import java.applet.Applet;
import ui.panels.ActionPanel;
import ui.panels.MainPanel;

public class GUIDemo extends Applet implements Resettable{
    ActionPanel actionPanel;
    MainPanel mainPanel;

    public void init() {
        resize(600,400);
        actionPanel = new ActionPanel();
        add(actionPanel);
        mainPanel = new MainPanel();
        add(mainPanel);
    }

    public void resetComponents(){
        mainPanel.resetComponents();
    }
}
```

Save it.

OBSERVE: GUIDemo

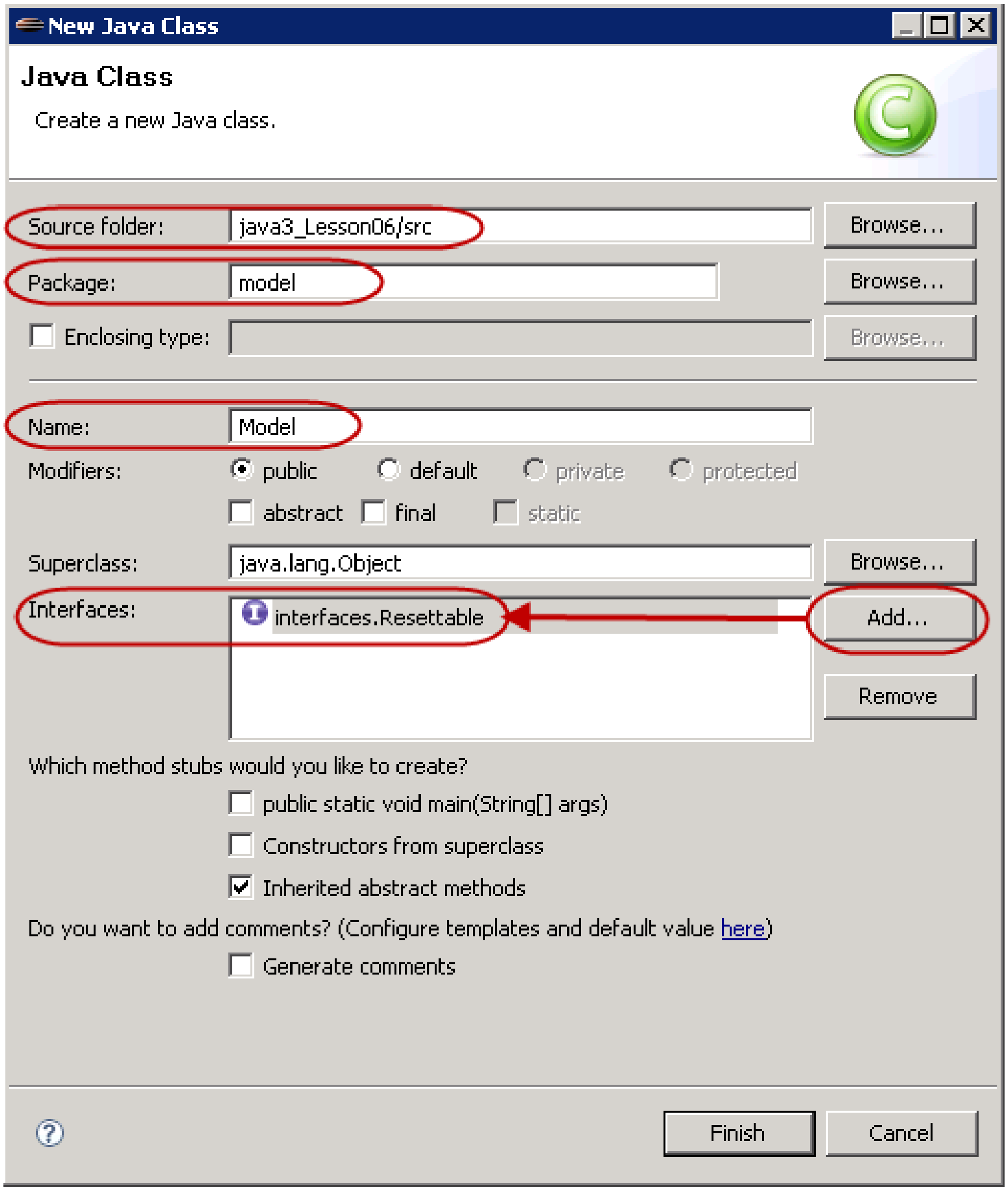
```
package ui.applet;

import interfaces.Resettable;
import java.applet.Applet;
import ui.panels.MainPanel;

public class GUIDemo extends Applet implements Resettable{
    MainPanel mainPanel;

    public void init() {
        resize(600, 400);
        mainPanel = new MainPanel();
        add(mainPanel);
    }

    public void resetComponents() {
        mainPanel.resetComponents();
    }
}
```

We replace ActionPanel with **MainPanel** and actionPanel with **mainPanel**. We implement the Resettable interface and its required method, **resetComponents()**. This gives us the ability to tell the GUIDemo applet to have the **mainPanel** reset its components whenever this method is called.

Save all of your files and run the GUIDemo applet. The applet looks the same, but our structural changes make it more modular and easier to maintain in the future.

Now let's create the Model class for this applet and start making the applet work for us!

In the **model** package, create a new class named **Model**. This class should implement the Resettable interface as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson06/src

Package: model

☐ Enclosing type:

Name: Model

Modifiers: ● public ○ default ○ private ○ protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces: interfaces.Resettable

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

When our **Clear** button is clicked, it will tell the model to call **resetComponents()** and then the model will distribute that message as needed. Type **Model** in **blue** as shown:

CODE TO TYPE: Model

```
package model;

import java.awt.Container;
import interfaces.Resettable;

public class Model implements Resettable{
    private Container container;
    // Cut and paste these from the ActionPanel class, then make them public and static

    public final static String DRAW = "Draw";
    public final static String MOVE = "Move";
    public final static String REMOVE = "Remove";
    public final static String RESIZE = "Resize";
    public final static String CHANGE = "Change";
    public final static String FILL = "Fill";

    private String action = DRAW;
    private boolean fill = false;

    public Model (Container container) {
        this.container = container;
    }

    public void repaint() {
        container.repaint();
    }

    public void resetComponents() {
        action = DRAW;
        fill = false;
        if(container instanceof Resettable) {
            ((Resettable)container).resetComponents();
        }
    }
}
```

Save it.

OBSERVE: Model

```
package model;

import java.awt.Container;
import interfaces.Resettable;

public class Model implements Resettable {
    private Container container;
    public final static String DRAW = "Draw";
    public final static String MOVE = "Move";
    public final static String REMOVE = "Remove";
    public final static String RESIZE = "Resize";
    public final static String CHANGE = "Change";
    public final static String FILL = "Fill";

    private String action = DRAW;
    private boolean fill = false;

    public Model (Container container) {
        this.container = container;
    }

    public void repaint() {
        container.repaint();
    }

    public void resetComponents() {
        action = DRAW;
        fill = false;
        if(container instanceof Resettable) {
            ((Resettable)container).resetComponents();
        }
    }
}
```

This is the beginning of the Model class. The **container** object is an instance of the **Container** class, from which Applet and Frame descend. This allows us to have a reference to either an Applet or a Frame, so we can use this class with either an applet or a GUI application class. The **Container** class includes the **repaint()** method, so we can use **repaint()** on the **container** variable.

The constructor takes in a **Container** object as a parameter and then we *persist* this reference into the instance variable **container**.

We're moving the public final String objects from the ActionPanel class into the Model class, so that any other class can access them as well.

The String variable **action** will hold the current action being performed by the program (set to **DRAW** by default) and the boolean **fill** variable will hold the current state of the shape we are about to draw or change (set **false** by default).

The **repaint()** method passes a **repaint()** message on to the **container** object, so no other class needs to know about the **Container** for the program. We pass any messages for the applet or application through the Model.

The implementation of the **resetComponents()** method required by the Resettable interface, sets the action variable to equal our **DRAW** constant first, so the default action is **Draw**. We also set the **fill** variable to its default of **false**. Then the method tests the **container** variable to find out if it's a reference to a **Resettable** object (implements **Resettable**). If so, it calls the **resetComponents()** method of the **container**.

Note

((**Resettable**)**container**).**resetComponents()**; casts the **container** to a **Resettable** object, so that we can call its **resetComponents()** method. A **Container** object does not have a **resetComponents()** method, but a **Container** subclass that implements **Resettable** does. Pay attention to the position of the parentheses. We are casting the **container** object to a **Resettable** object, which allows us to access the **container** object's **Resettable** methods for this statement in our code. Then we call the **resetComponents()** method on that object. If we did not have the outside set of parentheses, we would be trying to cast the returned value of the **resetComponents()** method and we would get an error message at compile-time because **resetComponents()** returns void. Try removing the parentheses and see what happens.

We aren't quite finished working with this class. Let's add the **setters** and **getters**, and a **toString()** method to help with debugging. Edit the code in **Model** as shown in **blue**:

CODE TO EDIT: Model

```
package model;

import java.awt.Container;
import interfaces.Resettable;

public class Model implements Resettable{
    private Container container;
    public final static String DRAW = "Draw";
    public final static String MOVE = "Move";
    public final static String REMOVE = "Remove";
    public final static String RESIZE = "Resize";
    public final static String FILL = "Fill";
    public final static String CHANGE = "Change";

    private String action = DRAW;
    private boolean fill = false;

    public Model (Container container) {
        this.container = container;
    }

    public void repaint() {
        container.repaint();
    }

    public void resetComponents() {
        action = DRAW;
        fill = false;
        if(container instanceof Resettable) {
            ((Resettable)container).resetComponents();
        }
    }

    public String getAction() {
        return action;
    }

    public void setAction(String action) {
        this.action = action;
    }

    public boolean isFill() {
        return fill;
    }

    public void setFill(boolean fill) {
        this.fill = fill;
    }

    public String toString() {
        return "Model:\n\tAction: " + action + "\n\tFill: " + fill;
    }
}
```

Save the Model class. Now, we'll modify our other classes to take advantage of this one. Edit **GUIDemo** as shown in **blue**:

CODE TO EDIT: GUIDemo

```
package ui.applet;

import interfaces.Resettable;
import java.applet.Applet;
import ui.panels.MainPanel;
import model.Model;

public class GUIDemo extends Applet implements Resettable{
    MainPanel mainPanel;
    Model model;

    public void init() {
        resize(600,400);
        model = new Model(this);
        mainPanel = new MainPanel(model);
        add(mainPanel);
    }

    public void resetComponents() {
        mainPanel.resetComponents();
    }
}
```

OBSERVE: GUIDemo

```
package ui.applet;

import interfaces.Resettable;
import java.applet.Applet;
import ui.panels.MainPanel;
import model.Model;

public class GUIDemo extends Applet implements Resettable {
    MainPanel mainPanel;
    Model model;

    public void init() {
        resize(600,400);
        model = new Model(this);
        mainPanel = new MainPanel(model);
        add(mainPanel);
    }

    public void resetComponents() {
        mainPanel.resetComponents();
    }
}
```

We create an instance of our **Model** class named **model**. Then we pass the current instance of the GUIDemo class (**this**) to it as the Container parameter required by the **Model** constructor. Thinking ahead, we'll also pass the **model** variable to the **MainPanel** so that it can pass it along to the ActionPanel that we'll modify as well.

Save GUIDemo. There are errors, but they'll be fixed as we modify the other classes.

Open the **MainPanel** class if it is not already opened. Modify it as shown in **blue** below:

CODE TO EDIT: MainPanel

```
package ui.panels;

import interfaces.Resettable;
import java.awt.GridLayout;
import java.awt.Panel;
import model.Model;

public class MainPanel extends Panel implements Resettable {
    ActionPanel actionPanel;

    public MainPanel(Model model) {
        actionPanel = new ActionPanel(model);
        setLayout(new GridLayout(2,1));
        add(actionPanel);
    }

    public void resetComponents() {
        actionPanel.resetComponents();
    }
}
```

Save the **MainPanel** class, even though it has errors—they'll vanish when we finish modifying the **ActionPanel** class:

OBSERVE: MainPanel

```
package ui.panels;

import interfaces.Resettable;
import java.awt.GridLayout;
import java.awt.Panel;
import model.Model;

public class MainPanel extends Panel implements Resettable {
    ActionPanel actionPanel;

    public MainPanel(Model model) {
        actionPanel = new ActionPanel(model);
        setLayout(new GridLayout(2,1));
        add(actionPanel);
    }

    public void resetComponents() {
        actionPanel.resetComponents();
    }
}
```

A **Model** object is being accepted in the constructor of the **MainPanel** class and that object is then passed to the **ActionPanel** instance.

Open **ActionPanel** and edit it as shown below, adding the **blue** code and removing the **red** code:

CODE TO EDIT: ActionPanel

```
package ui.panels;

import interfaces.Resettable;
import java.awt.Checkbox;
import java.awt.CheckboxGroup;
import java.awt.GridLayout;
import java.awt.Panel;
import model.Model;

public class ActionPanel extends Panel implements Resettable{
    private CheckboxGroup actionGroup;
    private Checkbox chkDraw, chkMove, chkResize, chkRemove, chkChange, chkFill;
    // You may have already cut and pasted these into the Model class.
    private final String DRAW = "Draw";
    private final String MOVE = "Move";
    private final String RESIZE = "Resize";
    private final String REMOVE = "Remove";
    private final String CHANGE = "Change";
    private final String FILL = "Fill";

    public ActionPanel(final Model model){
        actionGroup = new CheckboxGroup();
        chkDraw = new Checkbox(Model.DRAW, actionGroup, true);
        chkMove = new Checkbox(Model.MOVE, actionGroup, false);
        chkResize = new Checkbox(Model.RESIZE, actionGroup, false);
        chkRemove = new Checkbox(Model.REMOVE, actionGroup, false);
        chkChange = new Checkbox(Model.CHANGE, actionGroup, false);
        chkFill = new Checkbox(Model.FILL, false);
        setLayout(new GridLayout(1,6));
        add(chkDraw);
        add(chkMove);
        add(chkResize);
        add(chkRemove);
        add(chkChange);
        add(chkFill);
    }

    public void resetComponents() {
        chkDraw.setState(true);
        chkMove.setState(false);
        chkResize.setState(false);
        chkRemove.setState(false);
        chkChange.setState(false);
        chkFill.setState(false);
    }
}
```

In this class, we take a **Model** instance named **model** in through the constructor's parameter. We mark this parameter as **final** because later we'll be adding **ItemListener**s via anonymous inner classes, and these kinds of classes require any accessed local variables to be marked as final. Because this is the only method that needs access to **model**, we don't need to create an instance variable for it.

And speaking of anonymous inner classes, we are about to give the ActionPanel Checkbox instances the ability to effect the model. There will be many times when we use specific snippets of code over and over again. Check it out:

OBSERVE: Boilerplate code

```
chk.addItemListener(new ItemListener(){
    public void itemStateChanged(ItemEvent e){
        model.setAction(Model.REPLACE_ME);
    }
});
```

We will repeat the above code snippet six times (once for each of the checkboxes) with slight variations, in our ActionPanel class. Rather than retyping all of that code, let's copy and paste it. We'll replace the **bold** elements above with specific items after copying.

OBSERVE: About The Boilerplate

```
chkDraw.addItemListener(new ItemListener(){
    public void itemStateChanged(ItemEvent e){
        model.setAction(Model.DRAW);
    }
});
```

We modified the code snippet for the chkDraw object. We replaced the **chk** placeholder with **chkDraw** and then replaced the **REPLACE_ME** placeholder with **DRAW**, which sends a message to the **model** object to change its action variable to equal the **DRAW** action. Copy/paste the code snippet six times in the listing below, and then modify the copies for each of the objects. The **chkFill** object will need a little more modification. Edit the code as shown in **blue**:

CODE TO TYPE: ActionPanel

```
package ui.panels;

import interfaces.Resettable;
import java.awt.Checkbox;
import java.awt.CheckboxGroup;
import java.awt.GridLayout;
import java.awt.Panel;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import model.Model;

public class ActionPanel extends Panel implements Resettable {

    private CheckboxGroup actionGroup;
    private Checkbox chkDraw, chkMove, chkResize, chkRemove, chkChange, chkFill;

    public ActionPanel(final Model model) {
        actionGroup = new CheckboxGroup();
        chkDraw = new Checkbox(Model.DRAW, actionGroup, true);
        chkMove = new Checkbox(Model.MOVE, actionGroup, false);
        chkResize = new Checkbox(Model.RESIZE, actionGroup, false);
        chkRemove = new Checkbox(Model.REMOVE, actionGroup, false);
        chkChange = new Checkbox(Model.CHANGE, actionGroup, false);
        chkFill = new Checkbox(Model.FILL, false);

        chkDraw.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                model.setAction(Model.DRAW);
            }
        });
        chkMove.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                model.setAction(Model.MOVE);
            }
        });
        chkResize.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                model.setAction(Model.RESIZE);
            }
        });
        chkRemove.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                model.setAction(Model.REMOVE);
            }
        });
        chkChange.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                model.setAction(Model.CHANGE);
            }
        });
        chkFill.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                model.setFill(chkFill.getState());
            }
        });

        setLayout(new GridLayout(1, 6));
        add(chkDraw);
        add(chkMove);
        add(chkResize);
        add(chkRemove);
        add(chkChange);
        add(chkFill);
    }

    public void resetComponents() {
        chkDraw.setState(true);
        chkMove.setState(false);
        chkResize.setState(false);
        chkRemove.setState(false);
        chkChange.setState(false);
        chkFill.setState(false);
    }
}
```

The **chkFill** instance also needs an **ItemListener**, but it is changing the **fill** variable in the **model** object. We set this variable to equal the state of the **chkFill** instance. So, if the **chkFill** instance is checked, then the **fill** variable of the **model** instance will be true.

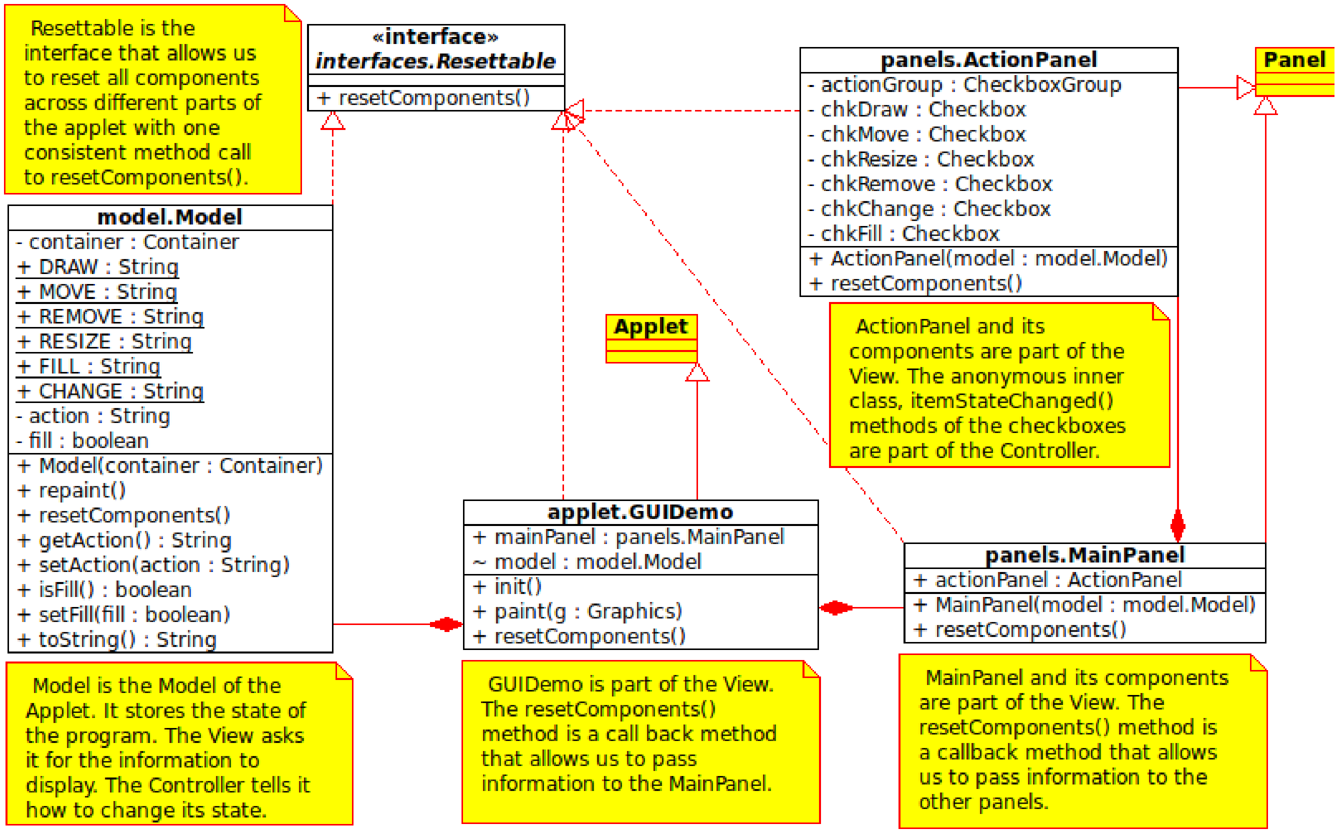
Save the **ActionPanel** class and edit the **GUIDemo** class as shown in **blue**:

CODE TO EDIT: GUIDemo

```
package ui.applet;

import interfaces.Resettable;
import java.applet.Applet;
import ui.panels.MainPanel;
import model.Model;
import java.awt.Graphics;

public class GUIDemo extends Applet implements Resettable{
    MainPanel mainPanel;
    Model model;

    public void init() {
        resize(600,400);
        model = new Model(this);
        mainPanel = new MainPanel(model);
        add(mainPanel);
    }

    public void paint(Graphics g){
        System.out.println(model);
    }

    public void resetComponents() {
        mainPanel.resetComponents();
    }
}
```

Save and run it. Change some of the checkboxes manually, then resize the applet. You can see in the console that the model has changed states. Also, dragging the appletviewer window border to manually resize the applet forces a **repaint()**.

This has been a long and lesson, so for now, just take a look at how the pieces are tied together in the UML Class Diagram for this part of the lesson:


Resettable is the interface that allows us to reset all components across different parts of the applet with one consistent method call to resetComponents().
«interface»
interfaces.Resettable
+ resetComponents()
panels.ActionPanel
- actionGroup : CheckboxGroup
- chkDraw : Checkbox
- chkMove : Checkbox
- chkResize : Checkbox
- chkRemove : Checkbox
- chkChange : Checkbox
- chkFill : Checkbox
+ ActionPanel(model : model.Model)
+ resetComponents()
Panel
model.Model
- container : Container
+ DRAW : String
+ MOVE : String
+ REMOVE : String
+ RESIZE : String
+ FILL : String
+ CHANGE : String
- action : String
- fill : boolean
+ Model(container : Container)
+ repaint()
+ resetComponents()
+ getAction() : String
+ setAction(action : String)
+ isFill() : boolean
+ setFill(fill : boolean)
+ toString() : String
Applet
ActionPanel and its components are part of the View. The anonymous inner class, itemStateChanged() methods of the checkboxes are part of the Controller.
applet.GUIDemo
+ mainPanel : panels.MainPanel
~ model : model.Model
+ init()
+ paint(g : Graphics)
+ resetComponents()
panels.MainPanel
+ actionPanel : ActionPanel
+ MainPanel(model : model.Model)
+ resetComponents()
Model is the Model of the Applet. It stores the state of the program. The View asks it for the information to display. The Controller tells it how to change its state.
GUIDemo is part of the View. The resetComponents() method is a call back method that allows us to pass information to the MainPanel.
MainPanel and its components are part of the View. The resetComponents() method is a callback method that allows us to pass information to the other panels.

It looks really similar to the example at the beginning of the lesson. That's because we're using the same MVC design pattern.

Great work so far. This was a really dense lesson, chock full of information. Good job hanging with it. You are a Java beast! We still have a lot more work to do to make this applet fully functional, but let's take a break here and congratulate ourselves on what we've accomplished! See you in the next lesson, where we will give this applet a more functionality!

Interfaces: Listeners and Adapters (continued)

Building the Shapes

In this section, we'll look again at our project from the previous lesson and trace the way our Java classes are developing. In the last homework assignment, we asked you to think about abstract classes and abstract methods for your Shape class, and then make adjustments to your classes based on what you had learned.

Take work with the **Shape** class that we modified earlier. It's now abstract. Again, you're not required to use our design for this next project, but you can if you like. If you do use your own design, make sure it includes all of the same functionality that is present in ours. This might be a bit tough, but, hey, you like a challenge, right? We encourage critical thinking, experimentation, and exploration. If you get lost along the way, you can always come back to this design.

Let's start by creating a new project named **Java3_Lesson07**. Copy the **src** folder from your **Java3_Lesson06** project into this new project.

Note Be sure to incorporate the changes you made in your last homework project.

Create or modify a Shape class in the **shapes** package. It will look something like this:

CODE TO EDIT: Shape

```
package shapes;

import java.awt.Color;
import java.awt.Graphics;

public abstract class Shape {
    private int x, y;
    private Color lineColor;

    public Shape(int x, int y, Color lineColor) {
        this.x = x;
        this.y = y;
        this.lineColor = lineColor;
    }

    public abstract void draw(Graphics g);
    public abstract boolean containsLocation(int x, int y);

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }

    public Color getLineColor() {
        return lineColor;
    }

    public void setLineColor(Color lineColor) {
        this.lineColor = lineColor;
    }
}
```

Save it.

OBSERVE: Shape

```
package shapes;

import java.awt.Color;
import java.awt.Graphics;

public abstract class Shape {
    private int x, y;
    private Color lineColor;

    public Shape(int x, int y, Color lineColor) {
        this.x = x;
        this.y = y;
        this.lineColor = lineColor;
    }

    public abstract void draw(Graphics g);
    public abstract boolean containsLocation(int x, int y);

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }

    public Color getLineColor() {
        return lineColor;
    }

    public void setLineColor(Color lineColor) {
        this.lineColor = lineColor;
    }
}
```

The class we're working on will continue to have relatively few properties and methods. All of the attributes are private, so getters and setters are especially important; we've made sure that they're all present.

The Shape class only has the attributes **x**, **y**, and **lineColor**, because *all* of its subclasses will have these attributes. Our constructor takes only these parameters, keeping the class clean.

We chose to make the **draw()** and **containsLocation()** methods abstract. The **draw()** method must be implemented in a concrete subclass of the Shape class (Rectangle, Oval, line) and will be responsible for drawing the particular shape. The **containsLocation()** method will take **x** and **y** locations as parameters and return *true* if that location is within the boundaries of the shape that is tested. This will give the concrete **Shape** subclasses the ability to tell other classes whether the **x** and **y** coordinate is within its boundaries.

This Shape class provides concrete subclasses with the minimum attributes they require. It also provides other classes with the minimum actions that they can take on *any* object that descends from the **Shape** class.

Now let's take a look at a concrete subclass. Create or modify a Rectangle class that extends Shape as shown:

CODE TO TYPE: Rectangle

```java
package shapes;

import java.awt.Color;
import java.awt.Graphics;

public class Rectangle extends Shape {
    private Color fillColor;
    private int width, height;
    private boolean fill;

    public Rectangle(int x, int y, int w, int h, Color lineColor, Color fillColor, boolean fill) {
        super(x, y, lineColor);
        this.width = w;
        this.height = h;
        this.fillColor = fillColor;
        this.fill = fill;
    }

    public void draw(Graphics g) {
        // Be nice. Save the state of the object before changing it.
        Color oldColor = g.getColor();
        if (isFill()) {
            g.setColor(getFillColor());
            g.fillRect(getX(), getY(), getWidth(), getHeight());
        }
        g.setColor(getLineColor());
        g.drawRect(getX(), getY(), getWidth(), getHeight());
        // Set the state back when done.
        g.setColor(oldColor);
    }

    // Override abstract method containsLocation in the Shape class.
    public boolean containsLocation(int x, int y) {
        if (getX() <= x && getY() <= y && getX() + getWidth() >= x && getY() + getHeight() >= y) {
            return true;
        }
        return false;
    }

    // Getters and setters.
    public Color getFillColor() {
        return fillColor;
    }

    public void setFillColor(Color fillColor) {
        this.fillColor = fillColor;
    }

    public int getWidth() {
        return width;
    }

    public void setWidth(int width) {
        this.width = width;
    }

    public int getHeight() {
        return height;
    }

    public void setHeight(int height) {
        this.height = height;
    }
    public void setFill(boolean fill) {
        this.fill = fill;
    }

    public boolean isFill() {
        return fill;
    }

    /**
    * Returns a String representing this object.
    */
    public String toString() {
        return "Rectangle: \n\tx = " + getX() + "\n\ty = " + getY() +
               "\n\tw = " + getWidth() + "\n\th = " + getHeight();
    }
}
```

Save it. Now we'll check out some aspects of this Rectangle class:

OBSERVE: Rectangle's Attributes

```
private Color fillColor;
private int width, height;
private boolean fill;
```

Because the Rectangle class is a subclass of the Shape class, we only have to define the ways a Rectangle *differs* from the Shape class. The Shape class already defines the **x**, **y**, and **lineColor** variables, so we don't need to redefine them here. We only need to add the variables for the aspects of a Rectangle that are in addition to those that define a Shape. That gives us a complete Rectangle definition with its **x**, **y**, **width**, **height**, **fill** (or not fill), **lineColor**, and **fillColor**.

OBSERVE: Rectangle's Constructor

```
public Rectangle(int x, int y, int w, int h, Color lineColor, Color fillColor,
  boolean fill) {
    super(x, y, lineColor);
    this.width = w;
    this.height = h;
    this.fillColor = fillColor;
    this.fill = fill;
}
```

Here we provide required parameters (shown in **orange**) to the constructor to define a Rectangle. Some of these are the same parameters taken in by our Shape class (**super**), so we pass those parameters up to it first. Because a Rectangle *is* a Shape, the Rectangle has access to these variables already (via the Shape class getters and setters). Next, we set our Rectangle **instance variables** equal to the parameters that were passed to it. Now we have a Rectangle object with the properties that were passed in to the constructor.

OBSERVE: Rectangle's draw(Graphics g) Method

```
public void draw(Graphics g) {
    // Be nice. Save the state of the object before changing it.
    Color oldColor = g.getColor();
    if (isFill()) {
        g.setColor(getFillColor());
        g.fillRect(getX(), getY(), getWidth(), getHeight());
    }
    g.setColor(getLineColor());
    g.drawRect(getX(), getY(), getWidth(), getHeight());
    // Set the state back when done.
    g.setColor(oldColor);
}
```

In the **draw()** method, we save the color of the **Graphics** object to **oldColor** first. Whenever possible, as a courtesy to other programmers (and ourselves!), we should return the basic state of the **Graphics** object to its original state, less the changes we want to make to this method.

Next, if the fill variable is true (via **isFill()**), we set the color of the **Graphics** object to the **fillColor**, then place a **filled rectangle** on the **Graphics** object in the location specified by this Rectangle object's state. We use the getters and setters from both Shape and Rectangle. We *must* use the getters from the Shape class, because we made its variables private. Even though it's not mandatory, in order to be consistent and to maintain the ability to copy and paste this code later in the Oval class, we use the getters and setters from the Rectangle class.

Next, we use setColor and **getLineColor** to change the color of the **Graphics** object to the lineColor and draw an unfilled rectangle on the **Graphics** object in the location specified by this Rectangle object's state. Finally, we change the **Graphics** object's color back to **oldColor**.

OBSERVE:

```
// Override abstract method containsLocation in the Shape class.
public boolean containsLocation(int x, int y) {
    if (getX() <= x && getY() <= y && getX() + getWidth() >= x
        && getY() + getHeight() >= y) {
        return true;
    }
    return false;
}
```

The **containsLocation()** method takes in **x** and **y** coordinates and compares that point to this Rectangle's **x**, **y**, **width**, and **height** values. If the **x** and **y** point is within the borders of this Rectangle (inclusive of the border itself), we return true; otherwise, we return false.

An Oval differs from a Rectangle in Java only in the way it is drawn. So we can conserve some effort when we work on the Oval class, by copying and pasting the constructor, **draw()**, and **toString()** methods from the Rectangle class, and modifying them slightly. Create or modify the Oval class as shown:

CODE TO TYPE: Oval

```
package shapes;

import java.awt.Color;
import java.awt.Graphics;

public class Oval extends Rectangle {
    /**
    * Constructor.  Just passes the params to the Rectangle constructor.
    */
    public Oval(int x, int y, int w, int h, Color lineColor, Color fillColor, boolean f
ill) {
        super(x, y, w, h, lineColor, fillColor, fill);
    }

    /*
    * Override Rectangle draw(Graphics g) method.
    */
    public void draw(Graphics g) {
        // Be nice. Save the state of the object before changing it.
        Color oldColor = g.getColor();
        if (isFill()) {
            g.setColor(getFillColor());
            g.fillOval(getX(), getY(), getWidth(), getHeight());
        }
        g.setColor(getLineColor());
        g.drawOval(getX(), getY(), getWidth(), getHeight());
        // Set the state back when done.
        g.setColor(oldColor);
    }

    /**
    * Returns a String that represents this object.
    */
    public String toString() {
        return "Oval: \n\tx = " + getX() + "\n\ty = " + getY() + "\n\tw = " + getWidth(
) + "\n\th = " + getHeight();
    }
}
```

Save it. There are only a few differences between Rectangle and Oval.

OBSERVE: Oval

```
package shapes;

import java.awt.Color;
import java.awt.Graphics;

public class Oval extends Rectangle {
    /**
    * Constructor.  Just passes the params to the Rectangle constructor.
    */
    public Oval(int x, int y, int w, int h, Color lineColor, Color fillColor, boolean f
ill) {
      super(x, y, w, h, lineColor, fillColor, fill);
    }

    /*
    * Override Rectangle draw(Graphics g) method.
    */
    public void draw(Graphics g) {
        // Be nice. Save the state of the object before changing it.
        Color oldColor = g.getColor();
        if (isFill()) {
            g.setColor(getFillColor());
            g.fillOval(getX(), getY(), getWidth(), getHeight());
        }
        g.setColor(getLineColor());
        g.drawOval(getX(), getY(), getWidth(), getHeight());
        // Set the state back when done.
        g.setColor(oldColor);
    }

    /**
    * Returns a String that represents this object.
    */
    public String toString() {
        return "Oval: \n\tx = " + getX() + "\n\ty = " + getY() + "\n\tw = " + getWidth(
) + "\n\th = " + getHeight();
    }
}
```

We modified the constructor to pass all of the parameters up to the **super** (the Rectangle constructor), and removed everything else in the constructor.

In the **draw()** method, we replaced **fillRect** with **fillOval** and **drawRect()** with **drawOval()**. We don't need to implement getters and setters in the Oval class because they already exist in the Rectangle class. In the **toString()** method, we changed the word **Rectangle** to the word **Oval** in the returned String.

It isn't necessary to change the **containsLocation()** method from the implementation in the Rectangle class, so we don't need to include it in the Oval class—it's inherited.

Note

In Java, **Oval**s are defined by a *bounding rectangle*, so any point in that bounding rectangle will register as being inside the Oval. If we needed finer control over this, we would change the **containsLocation()** method to calculate whether the location is actually within the drawn **Oval**, rather than within the bounding rectangle.

Now let's give our applet the ability to create a Shape (Rectangle or Oval) and display and size it using the mouse. You should now have usable Shape, Rectangle, and Oval classes in the **shapes** package of your **java3_Lesson07** project. (Make sure the package statement of your classes is correct.) To have a common point of reference in creating the next part of the lesson, we'll use our version of these classes.

First, we'll give our Model class the ability to keep track of a single Shape object. Later, we'll add the capability to have many different Shape objects displayed. For now, we'll illustrate the progress of program development one small step at a time.

Open your **Model** class and make the changes shown in **blue**:

CODE TO EDIT: Model

```
package model;

import java.awt.Color;
import shapes.Rectangle;
import shapes.Shape;
import java.awt.Container;
import interfaces.Resettable;

public class Model implements Resettable {
    private Container container;
    public final static String DRAW = "Draw";
    public final static String MOVE = "Move";
    public final static String REMOVE = "Remove";
    public final static String RESIZE = "Resize";
    public final static String FILL = "Fill";
    public final static String CHANGE = "Change";

    public final static String RECTANGLE = "Rectangle";
    public final static String OVAL = "Oval";

    private String action = DRAW;
    private boolean fill = false;

    private String currentShapeType = RECTANGLE;

    private Shape currentShape;

    public Shape createShape() {
        // If you changed this method in the previous homework project, you can include
 those changes here.
        if(currentShapeType == RECTANGLE){
          currentShape =  new Rectangle(0, 0, 0, 0, Color.black, Color.red, fill);
        }
        return currentShape;
    }

    public Shape getCurrentShape() {
        return currentShape;
    }

    public String getCurrentShapeType(){
      return currentShapeType;
    }

    public void setCurrentShapeType(String shapeType){
      currentShapeType = shapeType;
    }

    public Model(Container container) {
        this.container = container;
    }

    public void repaint() {
        container.repaint();
    }

    public void resetComponents() {
        action = DRAW;
        if (container instanceof Resettable) {
            ((Resettable) container).resetComponents();
        }
    }

    public String getAction() {
        return action;
    }
    public void setAction(String action) {
        this.action = action;
    }

    public boolean isFill() {
        return fill;
    }

    public void setFill(boolean fill) {
        this.fill = fill;
    }

    public String toString() {
        return "Model:\n\tAction: " + action + "\n\tFill: " + fill;
    }
}
```

💾 Save it.

OBSERVE:

```
package model;

import java.awt.Color;
import java.awt.Container;
import shapes.Rectangle;
import shapes.Shape;

import interfaces.Resettable;

public class Model implements Resettable {
    private Container container;
    public final static String DRAW = "Draw";
    public final static String MOVE = "Move";
    public final static String REMOVE = "Remove";
    public final static String RESIZE = "Resize";
    public final static String FILL = "Fill";
    public final static String CHANGE = "Change";

    public final static String RECTANGLE = "Rectangle";
    public final static String OVAL = "Oval";

    private String action = DRAW;
    private boolean fill = false;

    private String currentShapeType = RECTANGLE;

    private Shape currentShape;

    public Shape createShape() {
        // If you changed this method in the previous homework project, you can include
 those changes here.
        if(currentShapeType == RECTANGLE){
            currentShape =  new Rectangle(0, 0, 0, 0, Color.black, Color.red, fill);
        }
        return currentShape;
    }

    public Shape getCurrentShape() {
        return currentShape;
    }

    public String getCurrentShapeType(){
      return currentShapeType;
    }

    public void setCurrentShapeType(String shapeType){
      currentShapeType = shapeType;
    }

    public Model(Container container) {
        this.container = container;
    }

    public void repaint() {
        container.repaint();
    }

    public void resetComponents() {
        action = DRAW;
        if (container instanceof Resettable) {
            ((Resettable) container).resetComponents();
        }
    }

    public String getAction() {
        return action;
    }

    public void setAction(String action) {
        this.action = action;
    }

    public boolean isFill() {
        return fill;
    }

    public void setFill(boolean fill) {
        this.fill = fill;
    }

    public String toString() {
        return "Model:\n\tAction: " + action + "\n\tFill: " + fill;
    }
}
```

We created the **currentShape** variable to hold a reference to the **Shape** object that the model is currently monitoring. The **createShape()** method creates a new **Rectangle** object with no location, height, or width (we will remedy that in the mouse handler). We use the **fill** variable to tell the **Rectangle** whether to be filled or not. We assign this new object to the **currentShape** variable and then return the **currentShape** reference.

The **getCurrentShape()** method is a getter that provides a convenient way to get the **Shape** object that is being monitored by the **Model**.

Now our Model class will keep track of a single **Shape**. Right now, we don't have a way for the user to determine which **Shape** to draw. (That's part of your next homework project!)

We need a way for the applet to listen to the mouse. For now, let's just put our **MouseListener** class into the applet itself. This is an example of a *nested inner class*, which we'll discuss in depth later.

Our mouse listener is going to interact with the **Model**, not the applet, so it will need a handle to the **Model** object that the applet owns. Let's look at a basic mouse listener for this program. We could add the code to our GUIDemo applet, but we want as little *logic* in our applet as possible, so we can use the code in an application later with little or no modification. We'll add the code we need in a new file instead.

Create a new class named **ShapeMouseHandler** in the **event** package of your **java3_Lesson07** project and add the code shown in **blue**:

CODE TO TYPE: ShapeMouseHandler

```
package event;

import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import model.Model;
import shapes.Rectangle;
import shapes.Shape;

public class ShapeMouseHandler extends MouseAdapter {
    private Model model;
    //Start x and y location used to mark where the upper left corner of a
    //shape is.
    private int startX;
    private int startY;
    private Shape shape;

    /**
     * Constructor. Sets the model for this Listener.
     *
     * @param model
     */
    public ShapeMouseHandler(Model model) {
        //persist local variable model to this.model.
        this.model = model;
    }

    /*
     * Overrides MouseAdapter mousePressed method.
     */
    public void mousePressed(MouseEvent e) {
        if (model.getAction() == Model.DRAW) {
            // original upper left x and y of the shape.
            startX = e.getX();
            startY = e.getY();
            // have the model create a new shape for us.
            shape = model.createShape();
            // if the shape was created.
            if (shape != null) {
                //set its upper left x and y to where the mouse was pressed.
                shape.setX(e.getX());
                shape.setY(e.getY());
                // We should set a default width and height or ending location in
                // case the user does not drag the mouse.
                // Currently we only have instances of Rectangle or its descendants.
                if (shape instanceof Rectangle) {
                    ((Rectangle) shape).setWidth(50);
                    ((Rectangle) shape).setHeight(50);
                }
            }
        }
        // tell the model to repaint the applet or application.
        model.repaint();
    }
}
```

Note Make sure that the **ShapeMouseHandler** class is **public**.

Save it.

OBSERVE: ShapeMouseHandler

```
package event;

import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import model.Model;
import shapes.Rectangle;
import shapes.Shape;

public class ShapeMouseHandler extends MouseAdapter {
    private Model model;
    // Start x and y location used to mark where the upper left corner of a shape is.
    private int startX;
    private int startY;
    private Shape shape;

    /**
     * Constructor. Sets the model for this Listener.
     *
     * @param model
     */
    public ShapeMouseHandler(Model model) {
        //persist local variable model to this.model.
        this.model = model;
    }

    /*
     * Overrides MouseAdapter mousePressed method.
     */
    public void mousePressed(MouseEvent e) {
        if (model.getAction() == Model.DRAW) {
            // original upper left x and y of the shape.
            startX = e.getX();
            startY = e.getY();
            // have the model create a new shape for us.
            shape = model.createShape();
            // if the shape was created.
            if (shape != null) {
                //set its upper left x and y to where the mouse was pressed.
                shape.setX(e.getX());
                shape.setY(e.getY());
                // We should set a default width and height or ending location in
                // case the user does not drag the mouse.
                // Currently we only have instances of Rectangle or its descendants.
                if (shape instanceof Rectangle) {
                    ((Rectangle) shape).setWidth(50);
                    ((Rectangle) shape).setHeight(50);
                                    }
            }
        }
        // tell the model to repaint the applet or application.
        model.repaint();
    }
}
```

The **mousePressed()** method runs when the mouse is clicked on the applet. First, we test to see if we are in the **DRAW** mode in the **model**. If we are, then we get the **x** and **y** location of the mouse and set our **startX** and **startY** variables to this location. We'll need that location later in the **mouseDragged()** method.

Then we have the **model** create a **shape**. The model will track which shape to create. Right now, it will give us a new Rectangle object. If the **model** gives us a good **shape** (not null), then we set the shape's **x** and **y** location to the location where the mouse was clicked.

In case the user just clicks and does not drag the mouse, we still want the **shape** to appear, so we give it default **width** and **height** values (once we add the **mouseDragged()** method, these will be overwritten when they drag the mouse). To make sure that the object being referenced has a width and height (a **Line** would not) we test to see if the **shape** object is an instanceof a **Rectangle**. If it is, we cast it to a Rectangle and set its **width** and **height**. If we had more **Shape** types, we would add in code to test them as well and do whatever cast was necessary. For instance, for a Line shape, we would test to see if the **shape** was an instance of **Line** and then perhaps cast it to a **Line** and give it some default ending x and ending y location.

At the end of the **mousePressed()** method, we tell the **model** to **repaint()**. Remember, the **model** will pass this message on to its container—in this case, it's our applet.

Now, edit GUIDemo to enable our **ShapeMouseHandler**, as shown in **blue**:

CODE TO EDIT: GUIDemo

```
package ui.applet;

import interfaces.Resettable;
import java.applet.Applet;
import java.awt.Graphics;

import event.ShapeMouseHandler;

import shapes.Shape;

import ui.panels.MainPanel;
import model.Model;

public class GUIDemo extends Applet implements Resettable {
    MainPanel mainPanel;
    Model model;

    public void init() {
        resize(600,400);
        model = new Model(this);
        mainPanel = new MainPanel(model);
        add(mainPanel);
        ShapeMouseHandler mouseHandler = new ShapeMouseHandler(model);
        addMouseListener(mouseHandler);
        addMouseMotionListener(mouseHandler);
    }

    public void paint(Graphics g) {
        Shape shape;
        shape = model.getCurrentShape();
        if(shape != null) {
          shape.draw(g);
        }
        System.out.println(model);
        System.out.println(shape);
    }

    public void resetComponents() {
        mainPanel.resetComponents();
    }
}
```

Save and run it. Click on the applet to draw the shape. Also, try clicking on the **Fill** check box and then clicking in the applet again. We haven't added the **mouseDragged()** method yet, so when you click the mouse, the object is drawn, but nothing happens when you drag the mouse.

OBSERVE: GUIDemo with ShapeMouseHandler Enabled

```
package ui.applet;

import interfaces.Resettable;
import java.applet.Applet;
import java.awt.Graphics;

import event.ShapeMouseHandler;

import shapes.Shape;

import ui.panels.MainPanel;
import model.Model;

public class GUIDemo extends Applet implements Resettable {
    MainPanel mainPanel;
    Model model;

    public void init() {
        resize(600,400);
        model = new Model(this);
        mainPanel = new MainPanel(model);
        add(mainPanel);
        ShapeMouseHandler mouseHandler = new ShapeMouseHandler(model);
        addMouseListener(mouseHandler);
        addMouseMotionListener(mouseHandler);
    }

    public void paint(Graphics g) {
        Shape shape;
        shape = model.getCurrentShape();
        if(shape != null) {
          shape.draw(g);
        }
        System.out.println(model);
        System.out.println(shape);
    }

    public void resetComponents() {
        mainPanel.resetComponents();
    }
}
```

We created an instance of our **ShapeMouseHandler** class and added it to the applet as **MouseListener** and **MouseMotionListener**.

Note The GUIDemo applet does nothing without referring to the Model, MainPanel, or Shape class. Keeping the number of items that GUIDemo controls to a minimum in the program makes it easier to replace them later if we like.

The local Shape object, **shape**, is the Shape we will draw on the Graphics object. We get the current **shape** from the **model** and, if it is not null, we tell it to draw itself on the Graphics object.

Note We did not need to cast the **shape** object to a Rectangle. This is a good example of the potential power of abstract classes. The **Shape** class defined the **draw(Graphics g)** method as abstract. This guarantees that any concrete subclass of Shape has that method available. In Java, the object being referenced, not the type of reference, determines which method will be run. So, since the object in memory is a Rectangle object, Java runs the Rectangle's **draw()** method. Also, we did not need to call the **toString()** method of the **model** and **shape** objects in the **System.out.println()** methods. When we reference an object directly in a **System.out.println()**, the **toString()** method is automatically called. Because the base **toString()** method is defined in Object, all classes have a **toString()** method.

Let's add that **mouseDragged()** method now. Add the **blue** code to **ShapeMouseHandler** as shown:

CODE TO TYPE: ShapeMouseHandler

```
package event;

import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import model.Model;
import shapes.Rectangle;
import shapes.Shape;

public class ShapeMouseHandler extends MouseAdapter {
    private Model model;
    //Start x and y location used to mark where the upper left corner of a
    //shape is.
    private int startX;
    private int startY;
    private Shape shape;

    /**
     * Constructor. Sets the model for this Listener.
     *
     * @param model
     */
    public ShapeMouseHandler(Model model) {
        //persist local variable model to this.model.
        this.model = model;
    }

    /*
     * Overrides MouseAdapter mousePressed method.
     */
    public void mousePressed(MouseEvent e) {
        if (model.getAction() == Model.DRAW) {
            // original upper left x and y of the shape.
            startX = e.getX();
            startY = e.getY();
            // have the model create a new shape for us.
            shape = model.createShape();
            // if the shape was created.
            if (shape != null) {
                //set its upper left x and y to where the mouse was pressed.
                shape.setX(e.getX());
                shape.setY(e.getY());
                // We should set a default width and height or ending location in
                // case the user does not drag the mouse.
                // Currently we only have instances of Rectangle or its descendants.
                if (shape instanceof Rectangle) {
                    ((Rectangle) shape).setWidth(50);
                    ((Rectangle) shape).setHeight(50);
                }
            }
        }
        // tell the model to repaint the applet or application.
        model.repaint();
    }

    /*
     * Overrides MouseAdapter's mouseDragged method.
     */
    public void mouseDragged(MouseEvent e) {
        // get the current shape handled by the model.
        shape = model.getCurrentShape();
        // if there is a current shape in the model.
        if (shape != null) {
            // if we are in DRAW mode.
            if (model.getAction() == Model.DRAW) {
                // set the x and y location of the shape (allows rubber banding).
                shape.setX(Math.min(startX, e.getX()));
                shape.setY(Math.min(startY, e.getY()));
            }
            // if the shape is an instance of Rectangle or a descendant of Rectangle
            if (shape instanceof Rectangle) {
                // set its width and height.
                // allows for rubber banding.
                ((Rectangle) shape).setWidth(Math.abs(startX - e.getX()));
                ((Rectangle) shape).setHeight(Math.abs(startY - e.getY()));
            }
        }
        // tell the model to repaint the applet or application.
        model.repaint();
    }
}
```

Save it and run the **GUIDemo** applet again. Now you can draw the shape using the mouse. Try moving the mouse in all directions. Click the **Fill** check box and redraw the shape. Pretty cool, huh? Let's look at that code again:

OBSERVE: The mouseDragged(MouseEvent e) Method

```
package event;

import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import model.Model;
import shapes.Rectangle;
import shapes.Shape;

public class ShapeMouseHandler extends MouseAdapter {
    private Model model;
    //Start x and y location used to mark where the upper left corner of a
    //shape is.
    private int startX;
    private int startY;
    private Shape shape;

    /**
     * Constructor. Sets the model for this Listener.
     *
     * @param model
     */
    public ShapeMouseHandler(Model model) {
        //persist local variable model to this.model.
        this.model = model;
    }

    /*
     * Overrides MouseAdapter mousePressed method.
     */
    public void mousePressed(MouseEvent e) {
        if (model.getAction() == Model.DRAW) {
            // original upper left x and y of the shape.
            startX = e.getX();
            startY = e.getY();
            // have the model create a new shape for us.
            shape = model.createShape();
            // if the shape was created.
            if (shape != null) {
                //set its upper left x and y to where the mouse was pressed.
                shape.setX(e.getX());
                shape.setY(e.getY());
                // We should set a default width and height or ending location in
                // case the user does not drag the mouse.
                // Currently we only have instances of Rectangle or its descendants.
                if (shape instanceof Rectangle) {
                    ((Rectangle) shape).setWidth(50);
                    ((Rectangle) shape).setHeight(50);
                }
            }
        }
        // tell the model to repaint the applet or application.
        model.repaint();
    }

    /*
     * Overrides MouseAdapter's mouseDragged method.
     */
    public void mouseDragged(MouseEvent e) {
        // get the current shape handled by the model.
        shape = model.getCurrentShape();
        // if there is a current shape in the model.
        if (shape != null) {
            // if we are in DRAW mode.
            if (model.getAction() == Model.DRAW) {
                // set the x and y location of the shape (allows rubber banding).
                shape.setX(Math.min(startX, e.getX()));
                shape.setY(Math.min(startY, e.getY()));
            }
            // if the shape is an instance of Rectangle or a descendant of Rectangle
            if (shape instanceof Rectangle) {
                // set its width and height.
                // allows for rubber banding.
                ((Rectangle) shape).setWidth(Math.abs(startX - e.getX()));
                ((Rectangle) shape).setHeight(Math.abs(startY - e.getY()));
            }
        }
        // tell the model to repaint the applet or application.
        model.repaint();
    }
}
```

The **mouseDragged()** method runs when the user drags the mouse on the applet. Our program retrieves the current **shape** being tracked by the **model**. Then, it tests to make sure **shape** is not null. If it is a valid shape, the program test to see if we are in the **DRAW** action of the **model** object. If so, it does a little math to set the **x** and **y** location of the **shape**. Then we can draw the **shape** moving left, right, up, or down using the mouse (this is known as "rubber-banding" in Java). Now determine the smallest value between the **shape**'s **startX** and the **MouseEvent**'s **X** location and set the **shape**'s **x** location to that. Do the same with the **startY** and **MouseEvent**'s **Y** location.

Next, we perform an instance of test similar to the one we did in the **mousePressed()** method, to make sure we're working with a Rectangle or Rectangle subclass. If we are, we cast the **shape** object to a Rectangle and set its **width** to the absolute value of the **startX**, minus the **MouseEvent's X** location. Then we do the same with the **height**, using **startY** and the **MouseEvent's Y** location. Finally, we tell the **model** to **repaint()**, which passes that message along to the applet.

Adapters

Some **listeners** have several methods, but we don't always need them all. When we *implement* an interface though, we are also promising to implement *all* of its methods. Java provides a few options for getting this done.

A listener in Java with more than one method to be implemented always has a corresponding *adapter* class. Adapter classes can be subclassed and they implement all of their corresponding listeners' required methods. For instance, the **MouseAdapter** class implements all of the methods required by the **MouseListener** and **MouseMotionListener** interfaces. The **MouseAdapter** implements these methods as *no-op* or empty methods. They don't do anything.

The advantage in this is that we only have to implement the methods we need, rather than *all* of the listeners' methods.

The **ShapeMouseHandler** class extends **MouseAdapter**:

OBSERVE: ShapeMouseAdapter

```
public class ShapeMouseHandler extends MouseAdapter {
```

This allows us to implement only the **mousePressed()** and **mouseDragged()** methods, because those are the only methods we actually need.

Button Panel

We still have one more task to take care of in this lesson. We need to create that **Clear** Button.

Create a new class named **ButtonPanel** in the **ui.panels** package of your **java3_Lesson07** project. This class should extend **java.awt.Panel**. Add the **blue** code as shown:

CODE TO TYPE: ButtonPanel

```
package ui.panels;

import java.awt.Button;
import java.awt.Panel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import model.Model;

public class ButtonPanel extends Panel {
    private Button btnClear;

    public ButtonPanel(final Model model) {
        btnClear = new Button("Clear");
        btnClear.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                model.resetComponents();
                model.repaint();
            }
        });
        add(btnClear);
    }
}
```

Save it.

OBSERVE: ButtonPanel

```
package ui.panels;

import java.awt.Button;
import java.awt.Panel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import model.Model;

public class ButtonPanel extends Panel {
    private Button btnClear;

    public ButtonPanel(final Model model) {
        btnClear = new Button("Clear");
        btnClear.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                model.resetComponents();
                model.repaint();
            }
        });
        add(btnClear);
    }
}
```

Here, we create a panel to hold a single **Button** object named **btnClear**. We create an anonymous inner class for the **btnClear Button**. The **actionPerformed()** method calls the **model** object's **resetComponents()** method, which in turn calls the **GUIDemo** object's **resetComponents()** method, which in turn calls the **MainPanel** object's **resetComponents()** method, which then calls the **ActionPanel** object's **resetComponents()** method. Finally, we tell the **model** to **repaint()** (which tells the GUIDemo object to **repaint()**).

Controls Panel

Thinking ahead, we'll need additional controls that will allow us to select the Shape we want to draw and the colors we want to use for **lineColor** and **fillColor**. So, let's take this **ButtonPanel**, save it, and put it on another panel. Create a new class named **ControlsPanel** in the **ui.panels** package of your **java3_Lesson07** project. Add the **blue** code as shown:

CODE TO TYPE: ControlsPanel

```
package ui.panels;

import java.awt.Panel;

import interfaces.Resettable;
import model.Model;
import ui.panels.ButtonPanel;

public class ControlsPanel extends Panel implements Resettable{
    private ButtonPanel btnPanel;

    public ControlsPanel (Model model) {
        btnPanel = new ButtonPanel(model);
        add(btnPanel);
    }

    public void resetComponents() {
    }
}
```

Save it.

OBSERVE: ControlsPanel

```
package ui.panels;

import java.awt.Panel;

import interfaces.Resettable;
import model.Model;
import ui.panels.ButtonPanel;

public class ControlsPanel extends Panel implements Resettable{
    private ButtonPanel btnPanel;

    public ControlsPanel (Model model) {
        btnPanel = new ButtonPanel(model);
        add(btnPanel);
    }

    public void resetComponents() {
    }
}
```

Here in our example, we create a **ControlsPanel** class that extends Panel and implements Resettable (we don't need to implement Resettable yet, but we will later, so we might as well get it ready now).

We create and add a **ButtonPanel** object named **btnPanel** and create the constructor for the **ControlsPanel**.

Main Panel

Now, we need to add the **ControlsPanel** to our **MainPanel**. Add the **blue** code as shown:

CODE TO EDIT: MainPanel

```
package ui.panels;

import interfaces.Resettable;

import java.awt.GridLayout;
import java.awt.Panel;

import model.Model;

public class MainPanel extends Panel implements Resettable {
    ActionPanel actionPanel;
    ControlsPanel controlsPanel;

    public MainPanel(Model model) {
        actionPanel = new ActionPanel(model);
        controlsPanel = new ControlsPanel(model);
        setLayout(new GridLayout(2,1));
        add(controlsPanel);
        add(actionPanel);
    }
    public void resetComponents() {
        controlsPanel.resetComponents();
        actionPanel.resetComponents();
    }
}
```

Save it.

OBSERVE: MainPanel

```
package ui.panels;

import interfaces.Resettable;

import java.awt.GridLayout;
import java.awt.Panel;

import model.Model;

public class MainPanel extends Panel implements Resettable {
    ActionPanel actionPanel;
    ControlsPanel controlsPanel;

    public MainPanel(Model model) {
        actionPanel = new ActionPanel(model);
        controlsPanel = new ControlsPanel(model);
        setLayout(new GridLayout(2,1));
        add(controlsPanel);
        add(actionPanel);
    }
    public void resetComponents() {
        controlsPanel.resetComponents();
        actionPanel.resetComponents();
    }
}
```

Here, we create a new instance of the **ControlsPanel** class, named **controlsPanel**. We pass the **model**, which we got from **GUIDemo**, to the **ControlsPanel** constructor.

We add the **controlsPanel** object to the **MainPanel**, *then* we add the **actionPanel**. Controls go in the layout in the order that they are added to the panel. Try reversing the order. It won't effect the operation of the applet, only the way it appears.

We finalize this class by adding a call to the **controlsPanel**'s **resetComponents()**. The **ButtonPanel** itself doesn't need a call to the **controlsPanel**'s **resetComponents()**, but we will be adding other panels to the **ControlsPanel** that will.

Finally, we need to add the reset value for **currentShape** to the **Model** class. Open your **Model** class. Modify the **resetComponents()** method as shown in **blue**:

CODE TO EDIT: Model.java

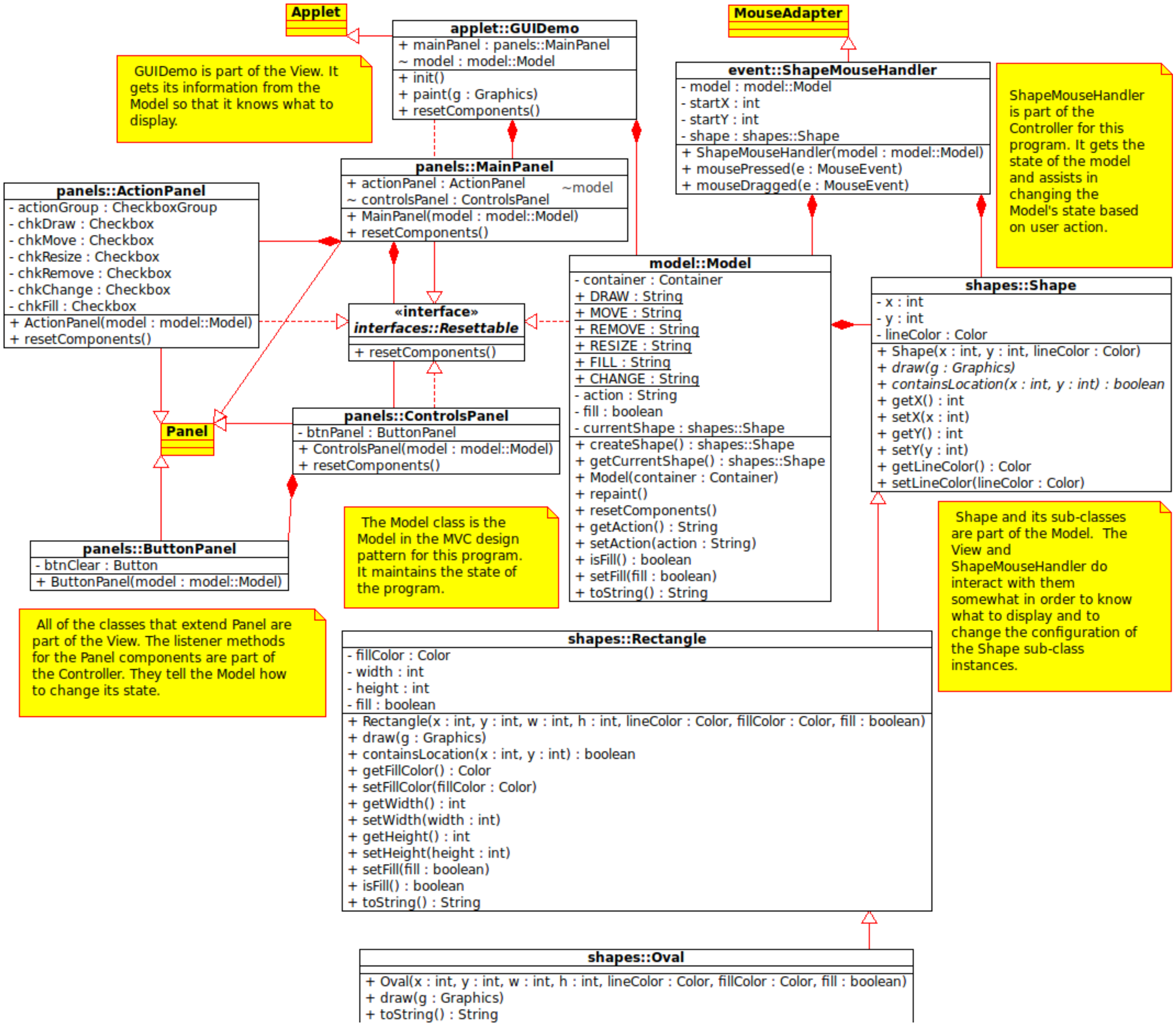
```java
package model;

import java.awt.Color;
import java.awt.Container;
import shapes.Rectangle;
import shapes.Shape;

import interfaces.Resettable;

public class Model implements Resettable {
    private Container container;
    public final static String DRAW = "Draw";
    public final static String MOVE = "Move";
    public final static String REMOVE = "Remove";
    public final static String RESIZE = "Resize";
    public final static String FILL = "Fill";
    public final static String CHANGE = "Change";

    public final static String RECTANGLE = "Rectangle";
    public final static String OVAL = "Oval";

    private String action = DRAW;
    private boolean fill = false;

    private String currentShapeType = RECTANGLE;

    private Shape currentShape;

    public Shape createShape() {
        // If you changed this method in the previous homework project, you can include
 those changes here.
        if(currentShapeType == RECTANGLE){
          currentShape =  new Rectangle(0, 0, 0, 0, Color.black, Color.red, fill);
        }
        return currentShape;
    }

    public Shape getCurrentShape() {
        return currentShape;
    }

    public String getCurrentShapeType(){
      return currentShapeType;
    }

    public void setCurrentShapeType(String shapeType){
      currentShapeType = shapeType;
    }

    public Model(Container container) {
        this.container = container;
    }

    public void repaint() {
        container.repaint();
    }

    public void resetComponents() {
        action = DRAW;
        currentShape = null;
        if (container instanceof Resettable) {
            ((Resettable) container).resetComponents();
        }
    }

    public String getAction() {
        return action;
    }

    public void setAction(String action) {
        this.action = action;
    }

    public boolean isFill() {
        return fill;
    }

    public void setFill(boolean fill) {
        this.fill = fill;
    }

    public String toString() {
        return "Model:\n\tAction: " + action + "\n\tFill: " + fill;
    }
}
```

Adding in the reset value for **currentShape** keeps the **paint()** method of the **GUIDemo** class from drawing a Rectangle when it repaints after the user clicks the **Clear** button.

Save all of your files.

Run the **GUIDemo** applet. When you press the **Clear** button, the **Fill** check box will clear if it was checked. Also, if you press any of the other radio buttons and then clear the applet, it will revert to the **Draw** button.

Okay, now let's take a look at our example's UML class diagram:



GUIDemo is part of the View. It gets its information from the Model so that it knows what to display.

ShapeMouseHandler is part of the Controller for this program. It gets the state of the model and assists in changing the Model's state based on user action.

The Model class is the Model in the MVC design pattern for this program. It maintains the state of the program.

Shape and its sub-classes are part of the Model. The View and ShapeMouseHandler do interact with them somewhat in order to know what to display and to change the configuration of the Shape sub-class instances.

All of the classes that extend Panel are part of the View. The listener methods for the Panel components are part of the Controller. They tell the Model how to change its state.

As we add new elements, the class diagram becomes more complex. Sometimes we'll break these diagrams up into many smaller diagrams to make them easier to understand.

Phew!

Wow. We've got lot to digest. As we review what we've accomplished in this lesson, pay special attention to the *structure* of our work. The design we've been working with is *model-centric*. The business logic and the state of the program are all in the model. The applet knows as little as possible and executes only the basic instantiation of the model and the GUI components. It instantiates only what it needs in order to complete its task. Everything else is within the model. Its **paint()** method does only what is necessary to draw the shapes.

This design uses a *push* strategy. The listeners **push** changes to the model. The listeners do not have any data in them, they simply tell the model to do something. The model never has to locate information, so it can concentrate on *using* its state rather than figuring out *where* and *when* to update its state.

You're doing really well so far. That was a long haul! You deserve a break and a reward, but come on back after that. See you in the next lesson!

Nested Classes

Maneuvering Around Classes

In the last lesson we used a MouseAdapter, which enabled us to exclude some of the mouse methods in the **MouseListener**. We created our own class, **ShapeMouseHandler**, to extend **MouseAdapter**, override one of the methods, and then access only the methods we needed. Then we took the **ShapeMouseHandler** out of our applet.

An alternative way to designate which classes to use is to define a *nested class*, a class inside our **Applet** subclass. Specific types of nested classes are also called *inner classes*. We've seen the anonymous inner class before, but in this lesson we'll explore it thoroughly.

Nested Classes

Nested classes are classes that are contained within another class or interface. Sometimes we want one class to be tightly associated with another class. A particular class might only be used by one other specific class, so we want to keep them connected. Or we may want a particular class to be accessible only through one other specific class. In each of these circumstances, using nesting classes is a good option.

Note In most cases, we want classes to be **loosely coupled**, meaning as little connection between classes as possible, to make reuse easier. But, in some cases, classes are so **tightly coupled**, that we use nested classes to keep the classes physically together to make maintenance easier. This is especially true with anonymous inner classes.

We have learned that classes can be defined to have *two* members: fields/variables and methods. But a class that contains one or more nested classes will have the nested classes as members as well.

Nested classes can be static or non-static:

- If a nested class is declared static, it's logically called a *static nested class*. Static nested classes are top-level classes.
- If a nested class is declared non-static, it's called an *inner class*. There are three types of inner classes:

    1. Member classes
    2. Local classes
    3. Anonymous classes

Nested Top-Level Classes

Nested classes aren't nearly as common as other constructs, so we don't see them in most classes within the API.

API Go to the **java.applet.Applet** class. Scroll down a bit. The first summary you'll see is the **Nested Class Summary**, followed by **Field Summary**, **Constructor Summary**, and **Method Summary**.

A nested top-level class (also known as a *static nested class*) is a class (or interface) defined as a static member of another class. Outer classes (like those we have already seen) may be declared only **public** or **package private**. **Package private** means there is no modifier present that provides access within the same package. Nested classes can be declared **private**, **public**, **protected**, or **package private**. The syntax for such nested classes looks like this:

OBSERVE: static member class syntax

```
class OuterClass {
 ...
   static class StaticNestedClass {
      ...
   }
}
```

A static nested class is a top-level class that is nested within another top-level class. Let's check it out!

Create a new **java3_Lesson08** project in the **Java3_Lessons** working set. If Java offers the option to "Open Associated Perspective," click **No**; we want to keep our own perspective environment. Create a new class named **NestTest** that extends **java.applet.Applet** in the new project. Then type the **blue** code as shown:

CODE TO TYPE: NestTest

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


public class NestTest extends Applet {
    private int defaultBirthYear = 1958;
    private int defaultBirthMonth = 12;
    private int defaultBirthDay = 23;

    private BirthDayClass birthDate;

    public void init() {
        birthDate = new BirthDayClass();
    }

    public void paint(Graphics g) {
        g.drawString("Default Birthdate: " + defaultBirthMonth + "/" + defaultBirthDay
+ "/"
                + defaultBirthYear, 0, 20);
        g.drawString("Birthdate from birthDate object: " + birthDate.getBirthMonth() +
"/"
                + birthDate.getBirthDay() + "/" + birthDate.getBirthYear(), 0, 40);
    }

    public static class BirthDayClass{
        private int birthYear;
        private int birthMonth;
        private int birthDay;

        public int getBirthYear() {
            return birthYear;
        }
        public void setBirthYear(int birthYear) {
            this.birthYear = birthYear;
        }
        public int getBirthMonth() {
            return birthMonth;
        }
        public void setBirthMonth(int birthMonth) {
            this.birthMonth = birthMonth;
        }
        public int getBirthDay() {
            return birthDay;
        }
        public void setBirthDay(int birthDay) {
            this.birthDay = birthDay;
        }
        public BirthDayClass() {
            birthYear = defaultBirthYear;
            birthMonth = defaultBirthMonth;
            birthDay = defaultBirthDay;
        }
    }
}
```

There are errors in the class. Nested classes do have access to the private data in the enclosing class, however, since the nested **BirthDateClass** class is marked static, it cannot access non-static data in the enclosing **NestTest** class. Nested classes marked static are top-level classes, just like **NestTest**. They must follow the same rules as a top-level class, *except* that they have direct access to their enclosing class' static data members and cannot access non-static data or methods of their enclosing class without a local instance of that class.

Give it a try. Add the **blue** code as shown:

CODE TO TYPE: NestTest

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


public class NestTest extends Applet {
    private static int defaultBirthYear = 1958;
    private static int defaultBirthMonth = 12;
    private static int defaultBirthDay = 23;

    private BirthDayClass birthDate;

    public void init() {
        birthDate = new BirthDayClass();
    }

    public void paint(Graphics g) {
        g.drawString("Default Birthdate: " + defaultBirthMonth + "/" + defaultBirthDay
+ "/"
                + defaultBirthYear, 0, 20);
        g.drawString("Birthdate from birthDate object: " + birthDate.getBirthMonth() +
"/"
                + birthDate.getBirthDay() + "/" + birthDate.getBirthYear(), 0, 40);
    }

    public static class BirthDayClass{
        private int birthYear;
        private int birthMonth;
        private int birthDay;

        public int getBirthYear() {
            return birthYear;
        }
        public void setBirthYear(int birthYear) {
            this.birthYear = birthYear;
        }
        public int getBirthMonth() {
            return birthMonth;
        }
        public void setBirthMonth(int birthMonth) {
            this.birthMonth = birthMonth;
        }
        public int getBirthDay() {
            return birthDay;
        }
        public void setBirthDay(int birthDay) {
            this.birthDay = birthDay;
        }
        public BirthDayClass() {
            birthYear = defaultBirthYear;
            birthMonth = defaultBirthMonth;
            birthDay = defaultBirthDay;
        }
    }
}
```

Save and run it.

OBSERVE: NestTest

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


public class NestTest extends Applet {
    private static int defaultBirthYear = 1958;
    private static int defaultBirthMonth = 12;
    private static int defaultBirthDay = 23;

    private BirthDayClass birthDate;

    public void init() {
        resize(400, 200);
        birthDate = new BirthDayClass();
    }

    public void paint(Graphics g) {
        g.drawString("Default Birthdate: " + defaultBirthMonth + "/" + defaultBirthDay
+ "/"
                + defaultBirthYear, 0, 20);
        g.drawString("Birthdate from birthDate object: " + birthDate.getBirthMonth() +
"/"
                + birthDate.getBirthDay() + "/" + birthDate.getBirthYear(), 0, 40);
    }

    public static class BirthDayClass{
        private int birthYear;
        private int birthMonth;
        private int birthDay;

        public int getBirthYear() {
            return birthYear;
        }
        public void setBirthYear(int birthYear) {
            this.birthYear = birthYear;
        }
        public int getBirthMonth() {
            return birthMonth;
        }
        public void setBirthMonth(int birthMonth) {
            this.birthMonth = birthMonth;
        }
        public int getBirthDay() {
            return birthDay;
        }
        public void setBirthDay(int birthDay) {
            this.birthDay = birthDay;
        }
        public BirthDayClass() {
            birthYear = defaultBirthYear;
            birthMonth = defaultBirthMonth;
            birthDay = defaultBirthDay;
        }
    }
}
```

The **BirthDayClass** instance variables, **birthYear**, **birthMonth**, and **birthDay**, are being set in the **BirthDayClass()** constructor. They are given the values of the **private static** data members, **defaultBirthYear**, **defaultBirthMonth**, and **defaultBirthDay**, of the enclosing **NestTest** class.

Create a new class named **NestTest2** in your project and type in the code below as shown:

CODE TO TYPE: NestTest2

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

public class NestTest2 extends Applet {
    NestTest.BirthDayClass birthDate;

    public void init() {
        birthDate = new NestTest.BirthDayClass();
    }

    public void paint(Graphics g) {
        g.drawString("Birthdate from birthDate object: "
                + birthDate.getBirthMonth() + "/" + birthDate.getBirthDay()
                + "/" + birthDate.getBirthYear(), 0, 40);
    }
}
```

Save and run it.

OBSERVE:

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

public class NestTest2 extends Applet {
    NestTest.BirthDayClass birthDate;

    public void init() {
        birthDate = new NestTest.BirthDayClass();
    }

    public void paint(Graphics g) {
        g.drawString("Birthdate from birthDate object: "
                + birthDate.getBirthMonth() + "/" + birthDate.getBirthDay()
                + "/" + birthDate.getBirthYear(), 0, 40);
    }
}
```

We see here that a *static nested class* is a top-level class that can be used by other classes. In this case though, our static nested class is getting its initial data from the **NestTest** class even though it is being instantiated by the **NestTest2** class. That's because it has access to the static data of the **NestTest** class.

When we wanted to access that data, we had to use the syntax **OuterClass.InnerClass** and **OuterClass.InnerConstructor()**.

OBSERVE: Static Nested Class Top-Level Access

```
NestTest.BirthDayClass birthDate;

public void init() {
    birthDate = new NestTest.BirthDayClass();
}
```

Note A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class—by having an instance of the outer class to work through.

Finally, static nested classes do not have to be public. They can have any of the normal access modifiers.

Edit the **NestTest** class by adding the **blue** code and removing the **red** code as shown:

CODE TO EDIT:

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


public class NestTest extends Applet {
    private static int defaultBirthYear = 1958;
    private static int defaultBirthMonth = 12;
    private static int defaultBirthDay = 23;

    private BirthDayClass birthDate;

    public void init() {
        resize(400, 200);
        birthDate = new BirthDayClass();
    }

    public void paint(Graphics g) {
        g.drawString("Default Birthdate: " + defaultBirthMonth + "/" + defaultBirthDay
+ "/"
                + defaultBirthYear, 0, 20);
        g.drawString("Birthdate from birthDate object: " + birthDate.getBirthMonth() +
"/"
                + birthDate.getBirthDay() + "/" + birthDate.getBirthYear(), 0, 40);
    }

    publicprivate static class BirthDayClass{
        private int birthYear;
        private int birthMonth;
        private int birthDay;

        public int getBirthYear() {
            return birthYear;
        }
        public void setBirthYear(int birthYear) {
            this.birthYear = birthYear;
        }
        public int getBirthMonth() {
            return birthMonth;
        }
        public void setBirthMonth(int birthMonth) {
            this.birthMonth = birthMonth;
        }
        public int getBirthDay() {
            return birthDay;
        }
        public void setBirthDay(int birthDay) {
            this.birthDay = birthDay;
        }
        public BirthDayClass() {
            birthYear = defaultBirthYear;
            birthMonth = defaultBirthMonth;
            birthDay = defaultBirthDay;
        }
    }
}
```

Save and run the **NestTest** class. Nothing has changed because the **NestTest** class has access to its private members, even the **BirthDayClass** class. But now the **NestTest2** class has errors, because it cannot access the private members of the **NestTest** class.

We'll get rid of the errors by changing the **private back** to **public** in **NestTest**.

Note Nested interfaces are implicitly static; however, they can also be marked as static explicitly.

Inner Classes

To reiterate, non-static nested classes are called *inner classes*; there are three types:

- Member classes
- Local classes
- Anonymous classes

Let's go over **local** and **anonymous** classes first.

Local Classes

Both *static member classes* and *member classes* are defined as **members** of a class. In contrast, *local classes* are defined inside of a block of code, typically within a method. In Java, because all blocks of code are located within classes, local classes will be nested within some outer or containing class.

The defining characteristic of a local class is that it is local to a block of code. Like a local variable, a local class is valid only within the scope defined by its enclosing block. This characteristic enables us to determine which kind of *inner* class to use. If a member class is used only within a single method of its containing class, it's usually coded as a local class, rather than a member class.

Create a new class, named **NestTest3** that extends **java.applet.Applet**, and add the **blue** code as shown:

CODE TO EDIT: NestTest3

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.List;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class NestTest3 extends Applet {
    String[] listItems = { "John Lennon", "Paul McCartney", "George Harrison", "
Ringo Starr", "Pete Best" };
    String msg = "";

    public void init() {
        List myList = new List();
        for (String item : listItems) {
            myList.add(item);
        }
        myList.addActionListener(new ListListener());
        add(myList);

        class ListListener implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                msg = e.getActionCommand();
                repaint();
            }
        }
    }

    public void paint(Graphics g) {
        if (msg != "") {
            g.drawString("Beatle " + msg + " selected.", 0, 100);
        }
    }
}
```

There's still an error in our code. As with any **local** variable, our local inner class must be defined prior to its use in this block of code. Change the code by moving the class definition of **ListListener** written in **blue** code, to the location as shown:

CODE TO EDIT: NestTest3

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.List;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class NestTest3 extends Applet {
    String[] listItems = { "John Lennon", "Paul McCartney", "George Harrison", "
Ringo Starr", "Pete Best" };
    String msg = "";

    public void init() {
        class ListListener implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                msg = e.getActionCommand();
                repaint();
            }
        }

        List myList = new List();
        for (String item : listItems) {
            myList.add(item);
        }
        myList.addActionListener(new ListListener());
        add(myList);
    }

    public void paint(Graphics g) {
        if (msg != "") {
            g.drawString("Beatle " + msg + " selected.", 0, 100);
        }
    }
}
```

Save and run it. Double-click on a list item to show the message.

Inner local classes are subject to these rules and restrictions:

- A local class is visible only within the block that defines it.
- A local class and its members can never be used outside of the block that defines it. (Outsiders cannot see in.)
- Instances of local classes, like instances of member classes, have an enclosing instance that is implicitly passed to all constructors of the local class. (It *sees out*.)
- Like member classes, local classes cannot contain **static** fields, methods, or classes, with one exception: constants that are declared both **static** and **final**.
- Interfaces cannot be defined locally. They can be *implemented* locally, but not *defined*.
- Local classes must be defined prior to their use within the block of code in which they are defined.

Anonymous Inner Classes

An anonymous class is a local class without a name. Anonymous inner classes are the most commonly used inner classes. While a local class definition is a statement within a block of Java code, an anonymous class definition is an expression which can be included as part of a larger expression, such as a method call. The most common use of Anonymous Inner Classes is to define a Listener.

Create a new class named **NestTest4** that extends **java.applet.Applet** and modify it as shown in **blue** below:

CODE TO EDIT: NestTest4

```
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;


public class NestTest4 extends Applet {
    int x, y;
    Color myColor = Color.red;
    public void init() {
        this.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                x = e.getX();
                y = e.getY();
                repaint();
            }
        });
    }
    public void paint(Graphics g) {
        g.setColor(myColor);
        g.fillOval(x, y, 25, 25);
    }
}
```

Save and run it. Now experiment by clicking various places in the applet.

OBSERVE: NestTest4

```
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;


public class NestTest4 extends Applet {
    int x, y;
    Color myColor = Color.red;
    public void init() {
        this.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                x = e.getX();
                y = e.getY();
                repaint();
            }
        });
    }
    public void paint(Graphics g) {
        g.setColor(myColor);
        g.fillOval(x, y, 25, 25);
    }
}
```

Anonymous inner classes are used most commonly to create listeners for components. This is the typical pattern used for anonymous inner class instantiation:

OBSERVE:

```
        componentVariableName.addListenerClassName(new AdapterClassOrInterfaceConstructor(){
            public void methodThatNeedsToBeImplemented(EventClassName eventVariableName){
               //what happens when the event is fired goes here.
            }
        });
```

Each Component that allows a Listener for events will have a method to add that listener to the component. Look at the Java API API and find the **java.awt.Button** class. Notice that it has an **addActionListener()** method. Now look at the **java.awt.Choice** class and see that it has an **addItemListener()** method. This is a standard Java pattern. To add a listener to a component, the word **add** will be followed by the listener class name to form the method name.

Inside the method call to add the listener, we are actually defining a class with the **new** keyword. We can instantiate a subclass of an AdapterClass or we can instantiate an instance of a class that implements an Interface. For instance, we can instantiate an anonymous (no name) subclass of the **ActionListener** interface.

Create a class named **NestTest5** that extends **java.awt.Applet** and add the **blue** code as shown:

CODE TO TYPE:

```
import java.applet.Applet;
import java.awt.Button;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class NestTest5 extends Applet {
    private static int count = 0;
    public void init() {
        Button myButton = new Button("I've been pressed " + count + " times.");
        // compare to the pattern above
        myButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                count++;
                myButton.setLabel("I've been pressed " + count + " times.");
            }
        });
        add(myButton);
    }
}
```

Whoops! There seems to be an error. But, why can't we access **myButton** from inside of the listener?

As explained in the book Java in a Nutshell, 5th edition (O'Reilly):

"a local class can use the local variables, method parameters, and even exception parameters that are in its scope, but only if those variables or parameters are declared **final**. This is because the lifetime of an instance of a local class can be much longer than the execution of the method in which the class is defined. For this reason, a local class must have a private internal copy of all local variables it uses (these copies are automatically generated by the compiler). The only way to ensure that the local variable and the private copy are always the same is to insist that the local variable is **final**."

Some of that explanation is beyond the scope of this course, but eventually it will all make sense to you.

Modify the **NestTest5** class by adding the **blue** code as shown:

CODE TO TYPE:

```
import java.applet.Applet;
import java.awt.Button;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class NestTest5 extends Applet {
    private static int count = 0;
    public void init() {
        final Button myButton = new Button("I've been pressed " + count + " time
s.");
        // compare to the pattern above
        myButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                count++;
                myButton.setLabel("I've been pressed " + count + " times.");
            }
        });
        add(myButton);
    }
}
```

Save and run it. Click on the button.

OBSERVE: NestTest5

```
import java.applet.Applet;
import java.awt.Button;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class NestTest5 extends Applet {
    private static int count = 0;
    public void init() {
        final Button myButton = new Button("I've been pressed " + count + " time
s.");
        // compare to the pattern above
        myButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                count++;
                myButton.setLabel("I've been pressed " + count + " times.");
            }
        });
        add(myButton);
    }
}
```

In our **new ActionListener()**, it seems like we are trying to instantiate an interface with a constructor. This is not the case. The compiler knows that we are creating an anonymous inner class by instantiating a new class that implements the interface **ActionListener**.

OBSERVE: NestTest4

```
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;


public class NestTest4 extends Applet {
    int x, y;
    Color myColor = Color.red;
    public void init() {
        this.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                x = e.getX();
                y = e.getY();
                repaint();
            }
        });
    }
    public void paint(Graphics g) {
        g.setColor(myColor);
        g.fillOval(x, y, 25, 25);
    }
}
```

In **NestTest4**, we are instantiating a new subclass of the Adapter Class, **MouseAdapter**.

When we create an anonymous inner class this way, we (programmers), do not have a handle to it. But in **NestTest5**, the **Button** object, **myButton** has a handle to the anonymous inner class in its listener queue. In **NestTest4**, the applet has a handle to the **MouseAdapter** subclass we created, in its listener queue. This syntax places the definition and use of the class in exactly the same place, which allows for easier maintenance.

Anonymous inner classes are preferred when creating listeners for unique components, such as a **Button** with a single, well-defined purpose.

Anonymous inner classes perform some operations automatically:

- An implicit call of the Constructor to the class's **super()**.
- Instantiation of an subclass of the indicated class, or an instantiation of a new class that implements the indicated *interface*.

Deciding When to Use Nested Classes

Every programmer has her own style and preferences. Nested classes are one of many options available in Java. Programmers may find nested classes are particularly useful when they work on High-Level Design or Low-Level Implementation.

High-Level Design Benefits

- Using a nested class is a logical way to group classes that are used in only one place; if a class is only useful to one other class, then keep them together.
- Using a nested class increases encapsulation.
- Using a nested class makes code more readable and easier to maintain because its definition is located nearer to where the class is executed.

Low-Level Implementation Benefits

- An object of an inner class can access the implementation of the object that created it—including data that would otherwise be private.
- An inner class can be hidden from other classes in the same package.
- And *anonymous* inner class is handy for defining an action "on the fly."

- Inner classes are convenient for writing event-driven programs.

As with most programming choices, there are also some **disadvantages** to using nested classes:

- Anonymous classes may make code difficult to read.
- Separation of Model-View-Controller becomes invalidated.
- Classes contain a mixture of purposes and are therefore no longer specific and easily understood.
- Inner classes may cause security concerns.

In the end it will be up to you to decide which features you need for your programs and how to achieve your goals. Now that you have a grasp on nested classes, let's return to **interfaces** and see how the two impact one another. See you in the next lesson!

Interfaces and Inheritance

Interfaces and Classes

In earlier lessons, we explored tools such as inheritance, abstract classes, interfaces, and nested classes, used to design large applications. In this lesson, we'll investigate interfaces. While interfaces aren't part of a class hierarchy, they do work in combination with classes.

Interfaces are used often by Java programmers to:

- act as listeners for graphical user interfaces.
- provide a type of multiple inheritance in Java.
- allow "callbacks."

So far we've used interfaces as listeners. Interfaces and classes are a similar. Like classes, when you define a new interface, you're defining a new data type. You can use interface names anywhere you can use data type names. An interface is like a class but with no implementation. So if you define a reference variable with a type that's an interface, any object you assign to it must be an instance of a class that implements that interface.

Java only allows *single inheritance*; a class can extend only one other class. But a class in Java can implement an unlimited number of interfaces. If a class implements more than one interface, it must implement every method of every interface it implements (or else the class must be declared abstract). We saw this in previous examples when we implemented a listener for the Buttons and the mouse.

Classes and interfaces share many of the same qualities, but they also differ significantly.

Shared Features of Classes and Interfaces

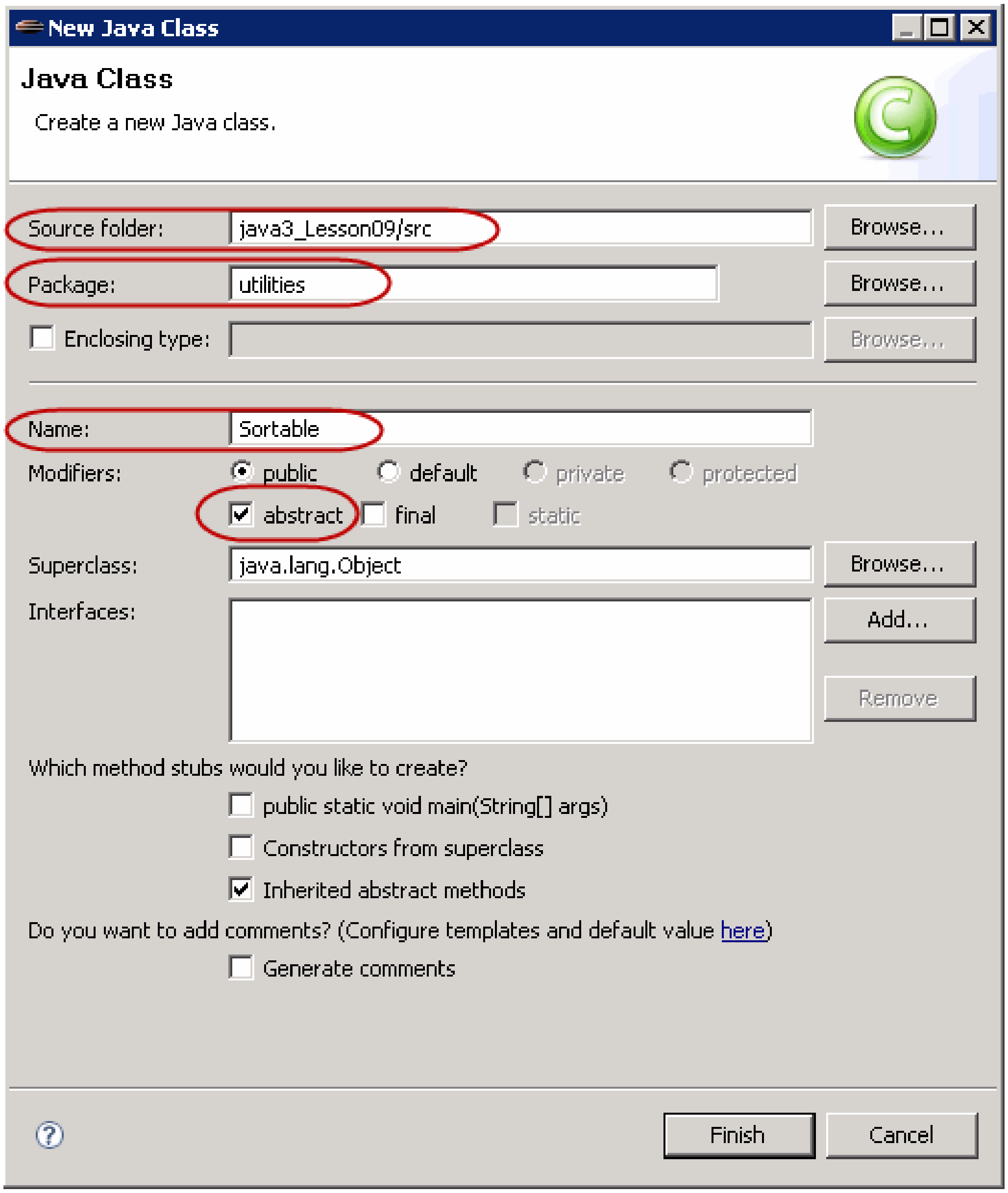
- Classes and interfaces are both types.
- Classes have fields (instance and class variables); interfaces can have fields, but **only** when they are static and final (constants without instance variables), **and** every field declaration in the body of an interface is implicitly public, static, and final.
- Classes have methods; interfaces make method declarations. (All method declarations in the body of an interface are implicitly public and abstract.)
- Classes can have subclasses; interfaces can have subinterfaces. A subinterface inherits all of the abstract methods and constants of its superinterface, and can define new abstract methods and constants.

Differences Between Classes and Interfaces

- Interfaces contain no implementation. For **all** interface methods, we use a semi-colon, but no method body.
- All methods of an interface are public. Anyone can implement the interface.
- Interfaces cannot be instantiated, so they cannot have constructors.
- Methods of an interface cannot be declared static (because static methods cannot be abstract). All of the methods of an interface must be instance methods.
- Interfaces can extend other interfaces. An interface can have an extends clause that lists more than one superinterface. All of the methods specified in the given interface **and** all of its superinterfaces must be implemented.
- A variable with an interface that is a declared type may have as its value a reference to any instance of a class that implements the specified interface. In these instances, the potential for "multiple inheritance" arises.

Interfaces and Multiple Inheritance

Let's get to work on some examples. Create a new **java3_Lesson09** project. If Java gives you the option to "Open Associated Perspective," click **No**. Now create a new **Sortable** class in this project:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson09/src

Package: utilities

□ Enclosing type:

Name: Sortable

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

In **Sortable**, type the **blue** code as shown:

CODE TO TYPE:

```
package utilities;

public abstract class Sortable {

    public abstract int compareTo(Sortable b);

    public static void shellSort(Sortable[] a){
        int n = a.length;
        int increment = n / 2;

        while (increment >= 1){
            for (int i = increment; i < n; i++){
                Sortable temp = a[i];
                int j = i;
                while (j >= increment && temp.compareTo(a[j - increment]) < 0){
                    a[j] = a[j - increment];
                    j = j - increment;
                }
                a[j] = temp;
            }
            increment = increment/2;
        }
    }
}
```

Save it.

OBSERVE:

```
package utilities;

public abstract class Sortable {

    public abstract int compareTo(Sortable b);

    public static void shellSort(Sortable[] a){
        int n = a.length;
        int increment = n / 2;

        while (increment >= 1){
            for (int i = increment; i < n; i++){
              Sortable temp = a[i];
              int j = i;
              while (j >= increment && temp.compareTo(a[j - increment]) < 0){
                  a[j] = a[j - increment];
                  j = j - increment;
              }
              a[j] = temp;
          }
          increment = increment/2;
        }
    }
}
```

In the **Sortable abstract** class, we define an **abstract** method, **compareTo()**, which takes a **Sortable** object. This method must be implemented by any concrete class that extends from the **Sortable abstract** class. We cannot implement the **compareTo()** method here, because we have no way to determine which kind of object we will need to compare.

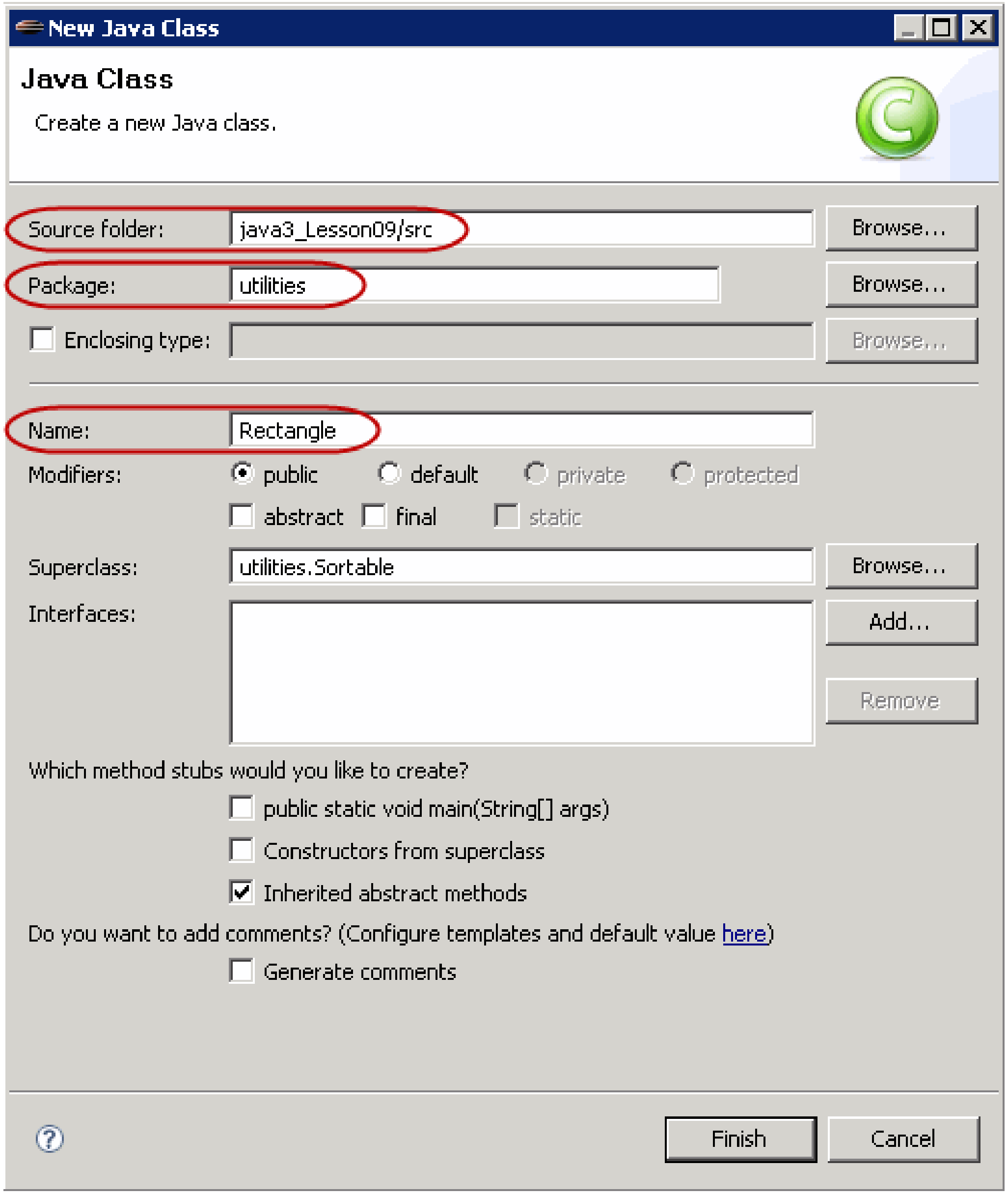
Note A concrete class is a class that can be instantiated. In the example above, the **compareTo()** method has to be implemented somewhere in the hierarchy before any subclass can become concrete.

In the **static shellSort()** method, we use the **compareTo()** method to compare a **temp Sortable** object with a **Sortable** object in the a[] array. We can implement the **compareTo()** method here because a **shell sort** doesn't require knowledge of the type of object we are comparing, as long as we have a way to compare it. We have the **compareTo()** method, which descends from **Sortable**.

Now, let's define a Rectangle class for use in our graphics drawing project. And since we already have a perfectly good **Sortable** class at our disposal, we can subclass it, and allow our users to sort the **Rectangle**s according to location.

WARNING Do **not** use the Rectangle class from previous lessons or your homework projects in this lesson. The Rectangle class we created in that lesson is not compatible with our current project!

Create a new **Rectangle** class in the java3_Lesson09 project as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson09/src

Package: utilities

□ Enclosing type:

Name: Rectangle

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: utilities.Sortable

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

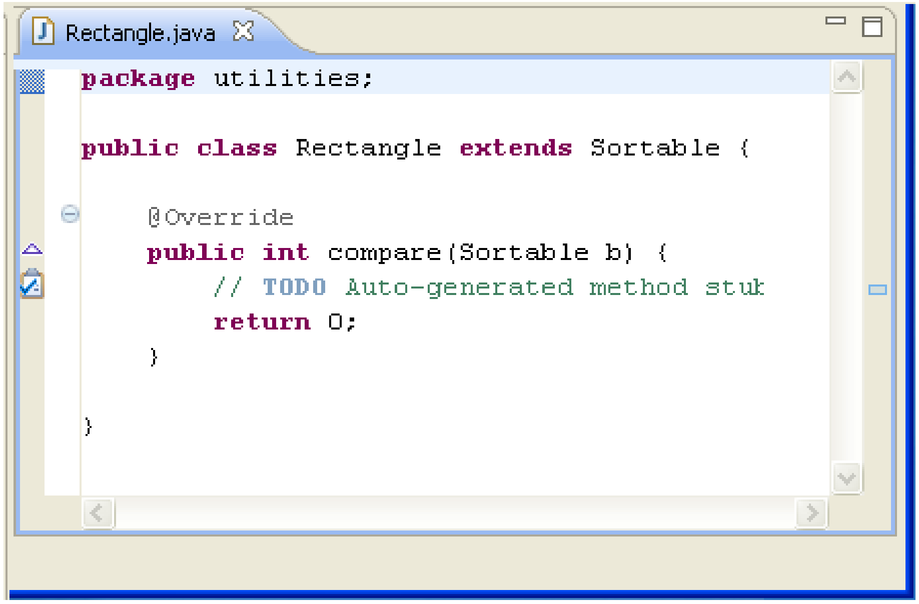
□ Constructors from superclass

☑ Inherited abstract methods

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

□ Generate comments

Because we inherited from an **abstract** class, the template that appears prompts us to implement the **abstract** method of its parent.



```
package utilities;

public class Rectangle extends Sortable {

    @Override
    public int compare(Sortable b) {
        // TODO Auto-generated method stub
        return 0;
    }

}
```

In **Rectangle**, type in the **blue** code as shown:

CODE TO TYPE:

```
package utilities;

public class Rectangle extends Sortable {
    int uLX, uLY, lRX, lRY;
    private int area;

    public Rectangle(int upperLeftX, int upperLeftY, int lowerRightX, int lowerRightY){
        uLX = upperLeftX;
        uLY = upperLeftY;
        lRX = lowerRightX;
        lRY = lowerRightY;
        setArea();
    }

    private void setArea(){
        area = (lRX - uLX) * (lRY - uLY);
    }

    public int getArea(){
        return area;
    }

    public int compareTo(Sortable b) {
        Rectangle oneToCompare = (Rectangle)b;
        if (getArea() < oneToCompare.getArea()) return -1;  // this one is smaller
        if (getArea() > oneToCompare.getArea()) return 1;   // this one is larger
        return 0;                                              // they are the same
    }
}
```

Save it.

OBSERVE:

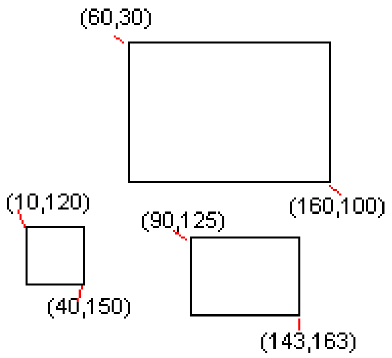
```
package utilities;

public class Rectangle extends Sortable {
    int uLX, uLY, lRX, lRY;
    private int area;

    public Rectangle(int upperLeftX, int upperLeftY, int lowerRightX, int lowerRightY){
        uLX = upperLeftX;
        uLY = upperLeftY;
        lRX = lowerRightX;
        lRY = lowerRightY;
        setArea();
    }

    private void setArea(){
        area = (lRX - uLX) * (lRY - uLY);
    }

    public int getArea(){
        return area;
    }

    public int compareTo(Sortable b) {
        Rectangle oneToCompare = (Rectangle)b;
        if (getArea() < oneToCompare.getArea()) return -1;  // this one is smaller
        if (getArea() > oneToCompare.getArea()) return 1;   // this one is larger
        return 0;                                           // they are the same
    }
}
```

We define our **Rectangle** class with two sets of coordinates: an **x** and **y** for the upper left corner and an **x** and **y** for the lower right corner. We'll be able to calculate **Rectangle**'s **area**, because these coordinates define the length and height of our rectangle. We'll also **implement** the code for the **abstract** method **compareTo()** so our rectangle will no longer be **abstract**.

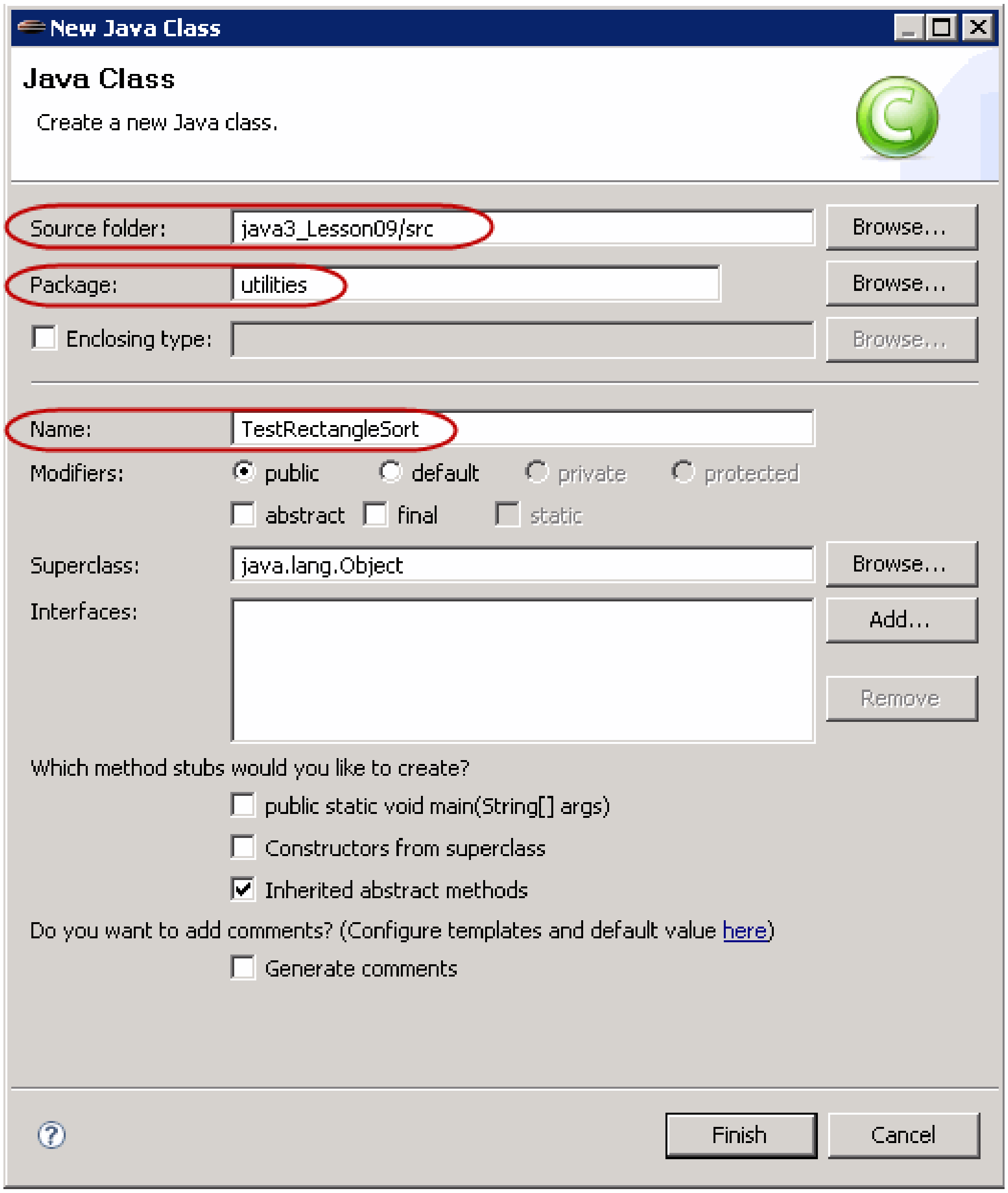
We'll focus primarily on the **compareTo()** method. We override the abstract method from the **Sortable** class. We take the parameter **b** and store it in the **oneToCompare** local variable, which is a **Rectangle** reference. We store **b** by casting it to a **Rectangle** object.

Now that our object is a **Rectangle**, we can call the **Rectangle** methods. We'll look at the area of the **Rectangle** using the **getArea()** method. If the area of this Rectangle is less than the area of **oneToCompare**, then we return a **-1**. If the Rectangle object's area is larger than that of the **oneToCompare** object, then we return a **+1**. If the two objects' areas are the same, then we return a **0**. This is the standard operation of any **compareTo()** method.

Okay, now let's sort some rectangles according to their areas:



To test our work, create a **TestRectangleSort** class in java3_Lesson09 as shown:



Java Class

Create a new Java class.

Source folder: java3_Lesson09/src

Package: utilities

□ Enclosing type:

Name: TestRectangleSort

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)

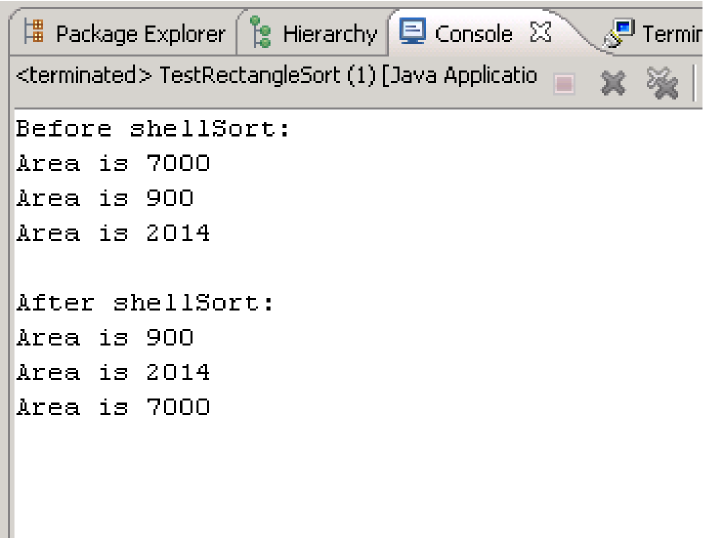
□ Generate comments

Type **TestRectangleSort** as shown in **blue**:

CODE TO TYPE:

```
package utilities;

public class TestRectangleSort {

    public static void main(String[] args){
        TestRectangleSort newExample = new TestRectangleSort();
        newExample.sortRectangles();
    }

    public void sortRectangles(){
        Rectangle[] figures = new Rectangle[3];

        figures[0] = new Rectangle(60,30,160,100);
        figures[1] = new Rectangle(10,120,40,150);
        figures[2] = new Rectangle(90,125,143,163);

        System.out.println("Before shellSort:");
        for (int i = 0; i < figures.length; i++)
            System.out.println("Area is " + figures[i].getArea());

        Sortable.shellSort(figures);

        System.out.println("\nAfter shellSort:");
        for (int i = 0; i < figures.length; i++)
            System.out.println("Area is " + figures[i].getArea());
    }
}
```

Save and run it. Your output will look like this:



<terminated> TestRectangleSort (1) [Java Applicatio

```
Before shellSort:
Area is 7000
Area is 900
Area is 2014

After shellSort:
Area is 900
Area is 2014
Area is 7000
```

OBSERVE:

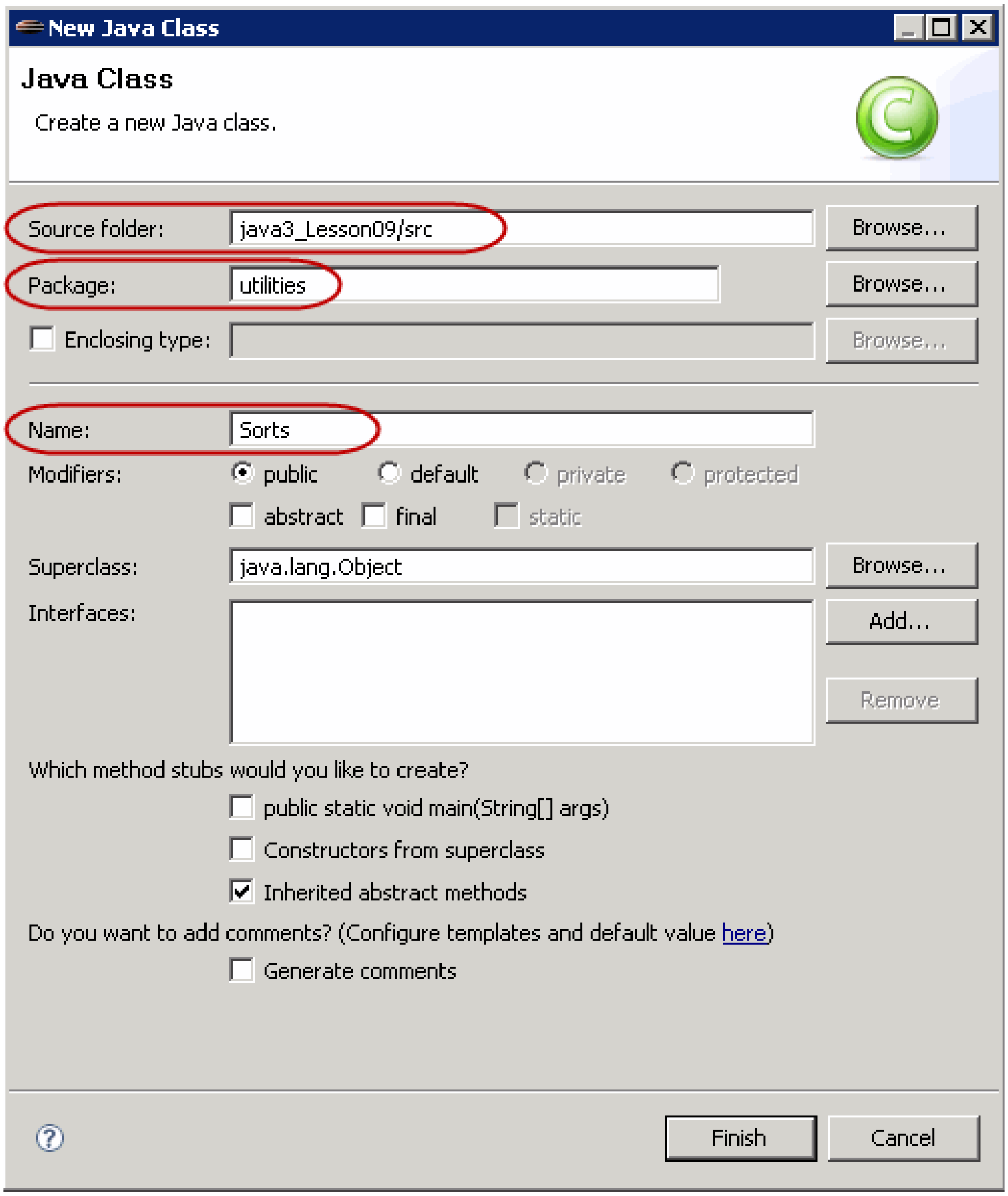
```
package utilities;

public class TestRectangleSort {

    public static void main(String[] args){
        TestRectangleSort newExample = new TestRectangleSort();
        newExample.sortRectangles();
    }

    public void sortRectangles(){
        Rectangle[] figures = new Rectangle[3];

        figures[0] = new Rectangle(60,30,160,100);
        figures[1] = new Rectangle(10,120,40,150);
        figures[2] = new Rectangle(90,125,143,163);

      System.out.println("Before shellSort:");
        for (int i = 0; i < figures.length; i++)
            System.out.println("Area is " + figures[i].getArea());

        Sortable.shellSort(figures);

     System.out.println("\nAfter shellSort:");
        for (int i = 0; i < figures.length; i++)
            System.out.println("Area is " + figures[i].getArea());
    }
}
```

In order to make this testing application, we create an array of **Rectangle**s, fill it with three **Rectangle** objects, and then display their values, in order. Once that's done, we call the **Sortable**.**shellSort()** method, passing the array and **figures** to the method. Then we display the array in order again.

It works great. It looks good. We are happy. That was a lot of work to make a single, yet important point. Because our **Rectangle** inherited from the **abstract** class **Sortable**, it cannot inherit from anything else, because Java has *single inheritance*. Now, if we wanted our Rectangle to inherit from the **abstract** Shape class in order to access (inherit) "Shape" kinds of things for our Rectangles, we couldn't do it, because Rectangle can only inherit from one class.

Interfaces specify which methods objects execute when they are implemented. Let's make our **Sortable** class an interface (also, we want Objects that **implement** the interface to be able to compare items so that they can be sorted).

Create a **Sorts** class in the java3_Lesson09 project as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson09/src

Package: utilities

□ Enclosing type:

Name: Sorts

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

Edit **Sorts** as shown in **blue**:

CODE TO TYPE:

```
package utilities;

public class Sorts {

    public static void shellSort(Sortable[] a){
        int n = a.length;
        int increment = n / 2;

        while (increment >= 1){
            for (int i = increment; i < n; i++){
                Sortable temp = a[i];
                int j = i;
                while (j >= increment && temp.compareTo(a[j - increment]) < 0){
                    a[j] = a[j - increment];
                    j = j - increment;
                }
                a[j] = temp;
            }
            increment = increment/2;
        }
    }
}
```

Look familiar? We lifted this code from the Sortable class. But we'll convert the Sortable class to an interface, so we can't have an implemented method in there.

Edit **Sortable** to make it an interface, adding the **blue** code and removing the **red** code as shown:

CODE TO EDIT:

```
package utilities;

public abstract class interface Sortable {

    public abstract int compareTo(Sortable b);

    public static void shellSort(Sortable[] a){
        int n = a.length;
        int increment = n / 2;

        while (increment >= 1){
            for (int i = increment; i < n; i++){
                Sortable temp = a[i];
                int j = i;
                while (j >= increment && temp.compareTo(a[j - increment]) < 0){
                    a[j] = a[j - increment];
                    j = j - increment;
                }
                a[j] = temp;
            }
            increment = increment/2;
        }
    }
}
```

Save it. It will look like this:

OBSERVE:

```
package utilities;

interface Sortable {
    int compareTo(Sortable b);
}
```

The Sortable class is now the Sortable *interface*, with only one method, **compareTo()**, that needs to be implemented.

Edit Rectangle to implement the interface rather than extend it, deleting the code in **red** and adding the code in **blue** as shown:

CODE TO EDIT:

```
package utilities;

public class Rectangle extends implements Sortable {
    int uLX, uLY, lRX, lRY;
    private int area;

    public Rectangle(int upperLeftX, int upperLeftY, int lowerRightX, int lowerRightY){
        uLX = upperLeftX;
        uLY = upperLeftY;
        lRX = lowerRightX;
        lRY = lowerRightY;
        setArea();
    }

    private void setArea(){
        area = (lRX - uLX) * (lRY - uLY);
    }

    public int getArea(){
        return area;
    }

    public int compareTo(Sortable b) {
        Rectangle oneToCompare = (Rectangle)b;
        if (getArea() < oneToCompare.getArea()) return -1;  // this one is smaller
        if (getArea() > oneToCompare.getArea()) return 1;   // this one is larger
        return 0;                                              // they are the same
    }

}
```

Edit TestRectangleSort to call the class method **shellSort()** from our new Sorts class rather than from Sortable, adding the **blue** code and removing the **red** code as shown:

CODE TO EDIT:

```
package utilities;

public class TestRectangleSort {

    public static void main(String[] args){
        TestRectangleSort newExample = new TestRectangleSort();
        newExample.sortRectangles();
    }

    public void sortRectangles(){
        Rectangle[] figures = new Rectangle[3];

        figures[0] = new Rectangle(60,30,160,100);
        figures[1] = new Rectangle(10,120,40,150);
        figures[2] = new Rectangle(90,125,143,163);

        System.out.println("Before shellSort:");
        for (int i = 0; i < figures.length; i++)
            System.out.println("Area is " + figures[i].getArea());

        Sortsable.shellSort(figures);

        System.out.println("\nAfter shellSort:");
        for (int i = 0; i < figures.length; i++)
            System.out.println("Area is " + figures[i].getArea());
        }
    }
}
```

Save and run it from TestRectangleSort. You should see the same result as before. Now that the Rectangle class doesn't extend anything, it can use **Shape** as a parent.

Let's go through the code in detail. We begin with the Rectangle class, which inherits from Sortable. Initially, because Rectangles are Sortable, we inherit in order to be able to sort our Rectangles. The Sortable class is abstract, so we implement its abstract **compareTo()** method.

But we'd really prefer for Rectangle to inherit from an abstract class named Shape instead, because Rectangles *are*, after all, Shapes. We want Rectangle to retain its ability to sort as well, so we change our Sortable class into an interface and then create a new Sorts class that can hold various sorting algorithms. There are many other types of Objects that we may want to sort in the future, so this maneuver is pretty cool. Now *any* class can implement the Sortable interface. And, if we include code for the **compareTo()** method, then we can sort arrays of its type using the static methods of the Sorts class.

Inheritance Design Conclusions

Using interfaces allows us to specify various capabilities, without forcing our classes to inherit methods.

When considering the use of inheritance in your design, ask yourself these questions:

- Does the subclass inherit *everything* from a parent?
- Is the class actually a subclass or is it simply sharing a common interface or common attributes with another class?

Answering these questions will help you choose the design options that best suit your purposes. Class fields present similar choices. A class may have member fields with values that serve as pointers to other class instances. This allows an instance of a class to have its own variables, as well as variables in common with and accessible from its member fields. For example, a user might have an instance variable named **myProfession**, which which points to a class **Profession**, using that user's particular professional information. Or, as we saw in the last lesson, a **Listener** (Controller) class might have an instance variable link to its **Container** (View), with its own specific members. These members do not *inherit* from each other, they simply *use* one another.

When designing a class:

- The class's member fields should be of a class type that holds additional information about some specific aspect of the class.
- When a class has capabilities (methods) that are not actions inherited from a parent, *but* are common types of actions for other types of classes, the class should implement an interface.

Note Interfaces provide a specification for a set of established action types. Each class that implements the interface writes its own code for the action. Classes can implement multiple interfaces.

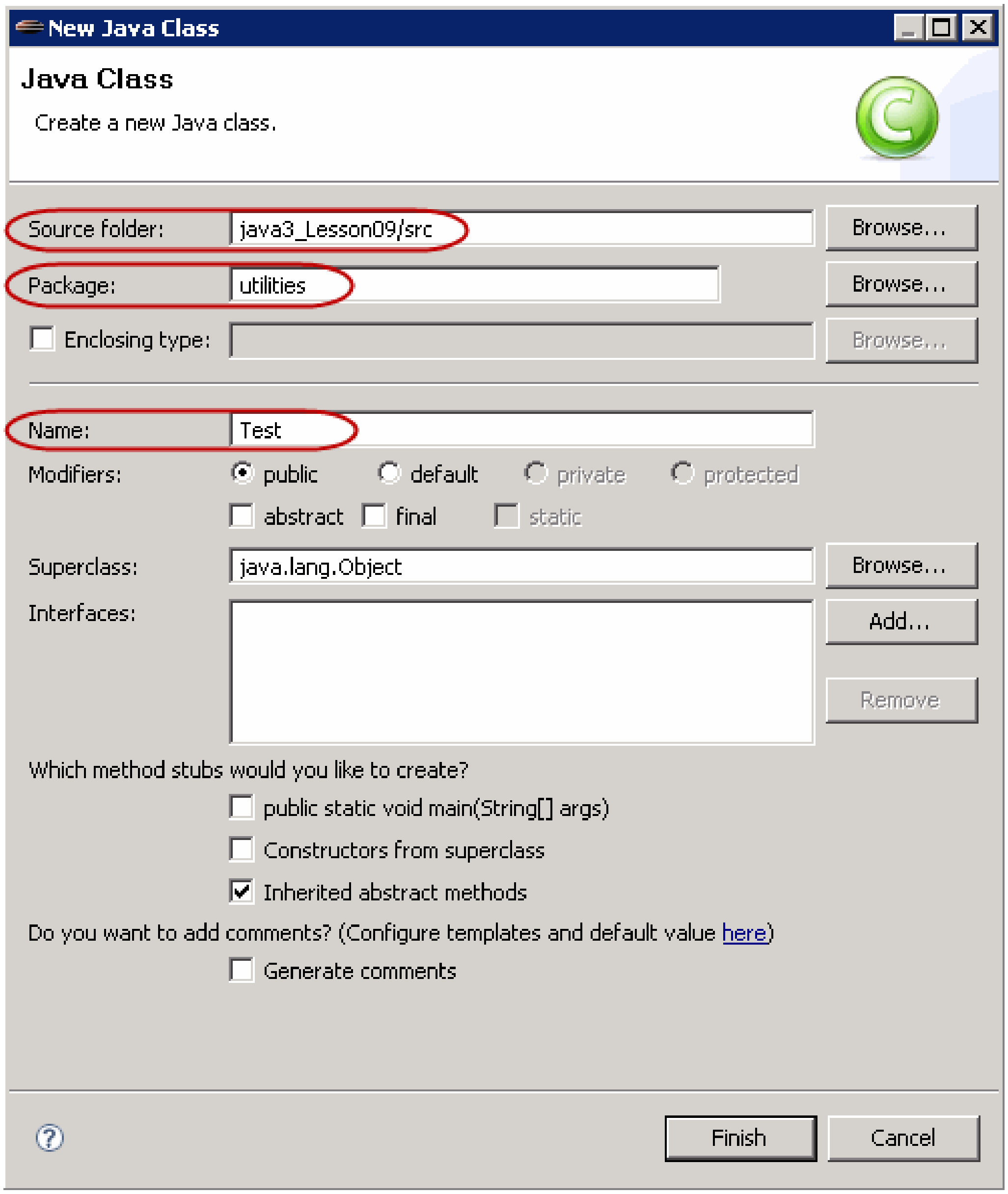
An interfaces is often written to be used by multiple classes—many different types of classes will implement that interface. Java cannot anticipate every possible use of an interface, so the amount of information the interface gets is limited to that which the interface defines. The next section will illustrate this further.

Casting

Interfaces as Types

A variable with an interface type as its declared type may have as its value, a reference to any instance of a class that implements the specified interface. When using multiple inheritance, an Object can be declared as an instance of a class type *or* something that implemented an interface type. The difference is that a declared interface type will only know about the interface methods. Let's experiment.

In the java3_Lesson09 project, create a new class named **Test**, as shown:



Java Class

Create a new Java class.

Source folder: java3_Lesson09/src

Package: utilities

□ Enclosing type:

Name: Test

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)

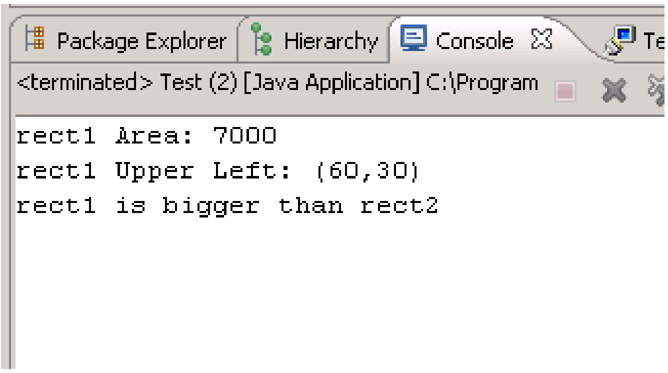
□ Generate comments

In **Test**, type the **blue** code as shown:

CODE TO TYPE:

```
package utilities;

public class Test {

    public static void main(String[] args){
        Test testMe = new Test();
        testMe.tryTypes();
    }

    public void tryTypes(){
        Rectangle rect1 = new Rectangle(60,30,160,100);
        Sortable rect2 = new Rectangle(10,120,40,150);
        Sortable [] figures = {rect1, rect2};

        System.out.println("rect1 Area: " + rect1.getArea());
        System.out.println("rect1 Upper Left:  (" + rect1.uLX + ","+ rect1.uLY +
")");

        int compareTest1 = rect1.compareTo(rect2);
        int compareTest2 = rect2.compareTo(rect1);
        if (compareTest1 > compareTest2)
            System.out.println("rect1 is bigger than rect2");
        else
            System.out.println("rect2 is bigger than rect1");
    }
}
```

Save and run it. You'll get this:



```
<terminated> Test (2) [Java Application] C:\Program
```

```
rect1 Area: 7000
rect1 Upper Left: (60,30)
rect1 is bigger than rect2
```

Let's see how we got there:

OBSERVE:

```
package utilities;

public class Test {

    public static void main(String[] args){
        Test testMe = new Test();
        testMe.tryTypes();
    }

    public void tryTypes(){
        Rectangle rect1 = new Rectangle(60,30,160,100);
        Sortable rect2 = new Rectangle(10,120,40,150);
        Sortable [] figures = {rect1, rect2};

        System.out.println("rect1 Area: " + rect1.getArea());
        System.out.println("rect1 Upper Left: (" + rect1.uLX + ","+ rect1.uLY +
")");

        int compareTest1 = rect1.compareTo(rect2);
        int compareTest2 = rect2.compareTo(rect1);
        if (compareTest1 > compareTest2)
            System.out.println("rect1 is bigger than rect2");
        else
            System.out.println("rect2 is bigger than rect1");
    }
}
```

In the **tryTypes()** method, we create **Rectangle** and **Sortable** local variables. Both of the variables, **rect1** and **rect2**, reference **Rectangle** objects. Because **Rectangle** implements **Sortable**, **Rectangle** *is a* **Sortable**. Because the **compareTo()** method is a **Sortable** method, all **Rectangle**s have that method. In Java, the object in memory that defines the method will be run, so we do not need to cast the **Rectangle** to a **Sortable** in order to call its **compareTo()** method. This is an example of *interface polymorphism*.

Try to change the constructor of rect2 to Sortable—remove the **red** code and add the **blue** code as shown:

CODE TO EDIT:

```
package utilities;

public class Test {

    public static void main(String[] args){
        Test testMe = new Test();
        testMe.tryTypes();
    }

    public void tryTypes(){
        Rectangle rect1 = new Rectangle(60,30,160,100);
        Sortable rect2 = new RectangleSortable(10,120,40,150);
        Sortable [] figures = {rect1, rect2};

        System.out.println("rect1 Area: " + rect1.getArea());
        System.out.println("rect1 Upper Left: (" + rect1.uLX + ","+ rect1.uLY +
")");

        int compareTest1 = rect1.compareTo(rect2);
        int compareTest2 = rect2.compareTo(rect1);
        if (compareTest1 > compareTest2)
            System.out.println("rect1 is bigger than rect2");
        else
            System.out.println("rect2 is bigger than rect1");
    }
}
```

We have an error: "**Cannot instantiate the type Sortable.**" Interfaces cannot be instantiated—they are not classes. Change the constructor back to **Rectangle**.

Because **rect1** is declared as a **Rectangle**, we can see all of the members of the **Rectangle** class, and it's possible to invoke its method **rect1.getArea()**, in order to get the public instance variables for its location coordinates **rect1.uLX** and **rect1.uLY**.

Since both **rect1** and **rect2** are **Sortable**, we can invoke the method(s) of the **Sortable** interface on either of them:

int **compareTest1** = **rect1.compareTo**(**rect2**);

int **compareTest2** = **rect2.compareTo**(**rect1**);

But **rect2** is a **Sortable** variable, so that's all we can see. If an Object is declared as a type of interface, then you can only access the interface members. Since we can't see all of the members of the **Rectangle** class, we can't invoke its method **rect2.getArea()**, so we can't retrieve its instance variables to find out its location coordinates **rect1.uLX** and **rect1.uLY**. Try it.

Edit **Test** as shown in **blue**:

CODE TO EDIT:

```
package utilities;

public class Test {

    public static void main(String[] args){
        Test testMe = new Test();
        testMe.tryTypes();
    }

    public void tryTypes(){
     Rectangle rect1 = new Rectangle(60,30,160,100);
        Sortable rect2 = new Rectangle(10,120,40,150);
        Sortable [] figures = {rect1, rect2};

        System.out.println("rect1 Area: " + rect1.getArea());
        System.out.println("rect1 Upper Left: (" + rect1.uLX + ","+ rect1.uLY +
")");
        System.out.println("rect2 Area: " + rect2.getArea());
        System.out.println("rect2 Upper Left: (" + rect2.uLX + ","+ rect2.uLY +
")");

        int compareTest1 = rect1.compareTo(rect2);
        int compareTest2 = rect2.compareTo(rect1);
        for (int i = 0; i < figures.length; i++)
            System.out.println("Area is " + figures[i].getArea());
        if (compareTest1 > compareTest2)
            System.out.println("rect1 is bigger than rect2");
        else
            System.out.println("rect2 is bigger than rect1");
    }
}
```

This code introduces more errors. The class members (variables and methods) of **Rectangle** are undefined for objects declared as the type **Sortable**. Interfaces do not know which type of instance an object *is*; interfaces only know which task the object is *contracted to do*. Don't change anything yet. We still have more to see.

Casting Back

By declaring an object as a type of interface, you limit the scope of the object to the declarations of that interface. The **instance** *is* still whatever its constructor was though, and can be *cast* to that class type to regain access to the class's information.

Edit **Test** as shown. Add the **blue** code and remove the **red** code:

CODE TO EDIT:

```
package utilities;

public class Test {

    public static void main(String[] args){
        Test testMe = new Test();
        testMe.tryTypes();
    }

    public void tryTypes(){
        Rectangle rect1 = new Rectangle(60,30,160,100);
        Sortable rect2 = new Rectangle(10,120,40,150);
        Rectangle rect3 = (Rectangle)rect2;
        Sortable [] figures = {rect1, rect2 rect3};

        System.out.println("rect1 Area: " + rect1.getArea());
        System.out.println("rect1 Upper Left: (" + rect1.uLX + ","+ rect1.uLY +
")");
        System.out.println("rect2rect3 Area: " + rect2rect3.getArea());
        System.out.println("rect2rect3 Upper Left: (" + rect2rect3.uLX + ","+ re
ct2 rect3.uLY + ")");

        int compareTest1 = rect1.compareTo(rect2rect3);
        int compareTest2 = rect2rect3.compareTo(rect1);

        for (int i = 0; i < figures.length; i++)
            System.out.println("Area is " + figures[i].getArea());
        if (compareTest1 > compareTest2)
            System.out.println("rect1 is bigger than rect2rect3");
        else
            System.out.println("rect2rect3 is bigger than rect1");
    }
}
```

We still seem to have the problem with the **for** loop. That's because we declared the whole array **figures** as **Sortable** so, like elements in the array, they only know **Sortable**. Change the **for** loop and cast the elements of the array. Add the **blue** code and remove the **red** code:

CODE TO EDIT:

```
package utilities;

public class Test {

  public static void main(String[] args){
    Test testMe = new Test();
    testMe.tryTypes();
  }

    public void tryTypes(){
        Rectangle rect1 = new Rectangle(60,30,160,100);
        Sortable rect2 = new Rectangle(10,120,40,150);
        Rectangle rect3 = (Rectangle)rect2;
        Sortable [] figures = {rect1, rect3};

        System.out.println("rect1 Area: " + rect1.getArea());
        System.out.println("rect1 Upper Left: (" + rect1.uLX + ","+ rect1.uLY +
")");
        System.out.println("rect3 Area: " + rect3.getArea());
        System.out.println("rect3 Upper Left: (" + rect3.uLX + ","+ rect3.uLY +
")");

        int compareTest1 = rect1.compareTo(rect3);
        int compareTest2 = rect3.compareTo(rect1);

        for (int i = 0; i < figures.length; i++) {
            Rectangle current = (Rectangle)figures[i];
            System.out.println("Area is " + figures[i]current.getArea());
        }
        if (compareTest1 > compareTest2)
            System.out.println("rect1 is bigger than rect3");
        else
            System.out.println("rect3 is bigger than rect1");
    }
}
```

Save and run it.

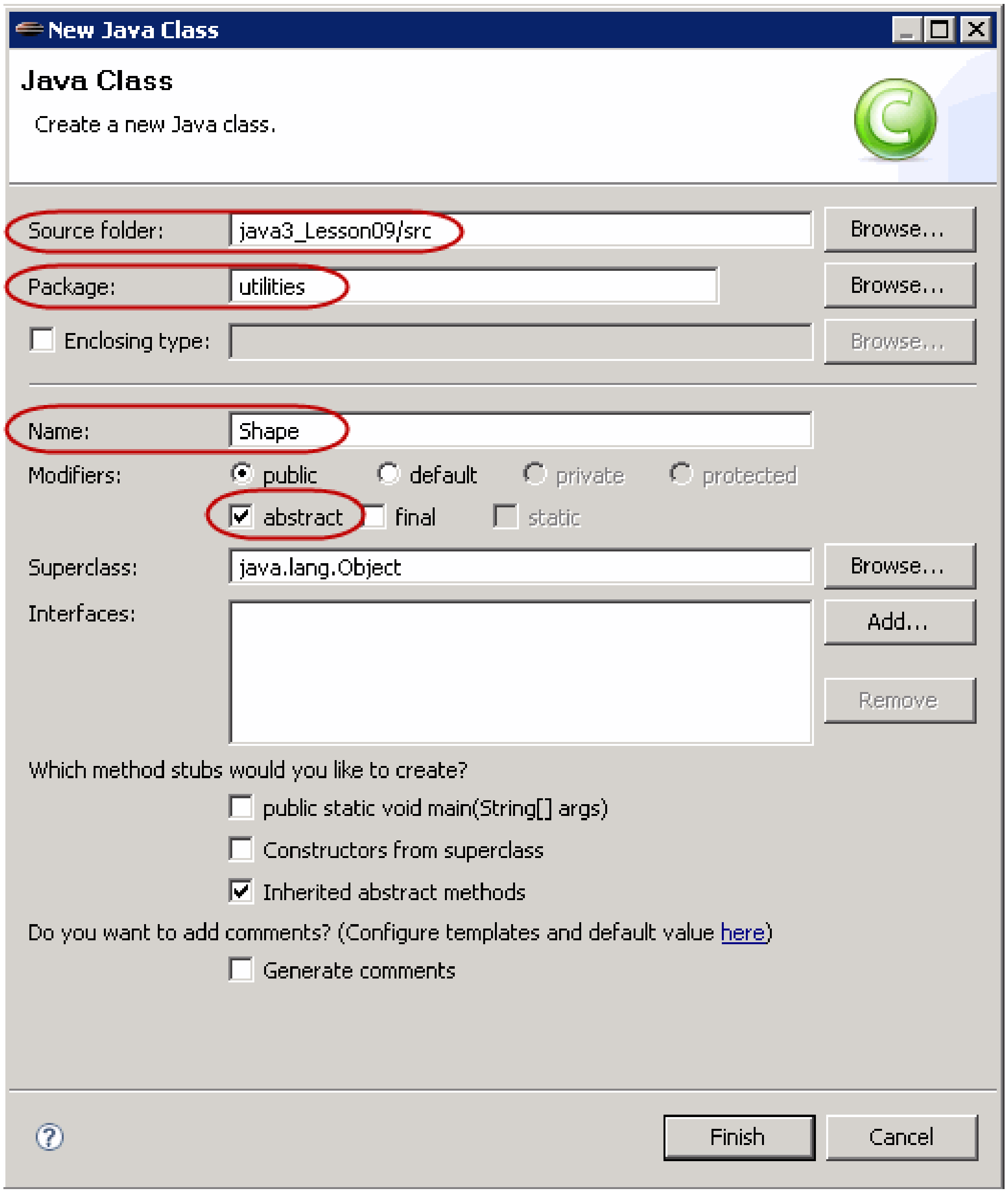
While at first glance it might seem like we've diminished the capacity of our code, we've really just limited our code's *access* to certain elements to a specific time. This technique lets Java avoid using multiple inheritance by enabling it to look at the same element in different ways. You may want Java to look at instances of a **Rectangle** *as* a **Rectangle**, and other times you might want them to be **compared** as **Sortable** items. (You can always cast the object to get all of its information again.)

Now let's try working not only with Rectangles, but with Squares, Ovals, Circles, and Triangles.

WARNING

In this lesson, *do not* use the **Shape** class from previous lessons or from your homework projects. The **Shape** class created in earlier projects is not compatible with this lesson.

In the java3_Lesson09 project, create a new class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson09/src

Package: utilities

□ Enclosing type:

Name: Shape

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

In **Shape**, add the code shown in **blue**:

CODE TO TYPE: Shape

```
package utilities;

public abstract class Shape {
    public abstract int getArea();
}
```

Our **Shape**s will be 2-D figures: they'll have **area**. We'll make sure that the items we compare in our example **have** a method for **getArea()**. And we'll want our **Rectangle** class to inherit from **Shape**.

Edit **Rectangle** as shown in **blue**:

CODE TO EDIT: Rectangle

```
package utilities;

public class Rectangle extends Shape implements Sortable {
    int uLX, uLY, lRX, lRY;
    private int area;

    public Rectangle(int upperLeftX, int upperLeftY, int lowerRightX, int lowerR
ightY){
        uLX = upperLeftX;
        uLY = upperLeftY;
        lRX = lowerRightX;
        lRY = lowerRightY;
        setArea();
    }

    private void setArea(){
        area = (lRX - uLX) * (lRY - uLY);
    }

    public int getArea(){
        return area;
    }

    public int compareTo(Sortable b) {
        Rectangle oneToCompare = (Rectangle)b;
        if (getArea() < oneToCompare.getArea()) return -1;  // this one is small
er
        if (getArea() > oneToCompare.getArea()) return 1;   // this one is large
r
        return 0;                                              // they are the s
ame
    }
}
```

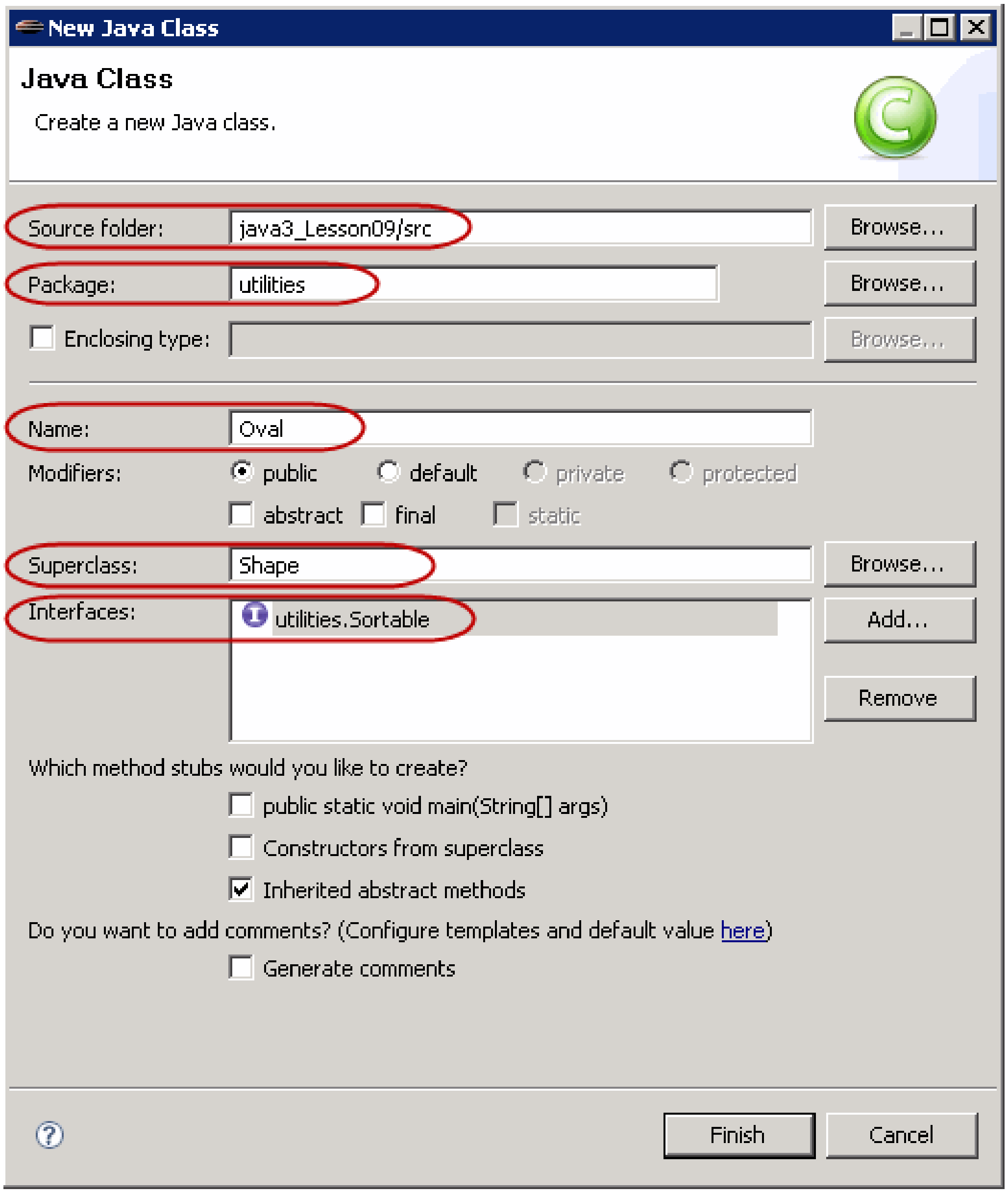
Save the **Shape** and **Rectangle** classes.

Casting: instanceof

Let's add an **Oval** class.

WARNING In this lesson, *do not* use the Oval class from previous lessons or from your homework projects! The Oval class created in earlier projects will not work in this lesson.

In the java3_Lesson09 project, create a new class as shown:



Java Class

Create a new Java class.

Source folder: java3_Lesson09/src

Package: utilities

□ Enclosing type:

Name: Oval

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: Shape

Interfaces: utilities.Sortable

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

In **Oval**, type the **blue** code as shown:

CODE TO TYPE:

```
package utilities;

public class Oval extends Shape implements Sortable {
    int uLX, uLY, lRX, lRY;
    private int area;

    public Oval(int upperLeftX, int upperLeftY, int lowerRightX, int lowerRightY
){
        uLX = upperLeftX;
        uLY = upperLeftY;
        lRX = lowerRightX;
        lRY = lowerRightY;
        setArea();
    }

    private void setArea(){
        // not necessarily a circle, so rather than PI*r*r,
        // we have for ellipses PI*a*b where a and b are half of width and heigh
t
        int width = lRX - uLX;
        int height = lRY - uLY;
        area = (int)(Math.PI*.5*width * .5*height);
    }

    public int getArea(){
        return area;
    }

    public int compareTo(Sortable b) {
        Shape oneToCompare = null;

        if (b instanceof Shape){
            oneToCompare = (Shape)b;
            if (getArea() < oneToCompare.getArea()) return -1;  // this one is s
maller
            if (getArea() > oneToCompare.getArea()) return 1;   // this one is l
arger
            return 0;                                           // they are the
same
        }
        return 0;
    }
}
```

Save it.

OBSERVE:

```
package utilities;

public class Oval extends Shape implements Sortable {
    int uLX, uLY, lRX, lRY;
    private int area;

    public Oval(int upperLeftX, int upperLeftY, int lowerRightX, int lowerRightY
){
        uLX = upperLeftX;
        uLY = upperLeftY;
        lRX = lowerRightX;
        lRY = lowerRightY;
        setArea();
    }

    private void setArea(){
        // not necessarily a circle, so rather than PI*r*r,
        // we have for ellipses PI*a*b where a and b are half of width and heigh
t
        int width = lRX - uLX;
        int height = lRY - uLY;
        area = (int)(Math.PI*.5*width * .5*height);
    }

    public int getArea(){
        return area;
    }

    public int compareTo(Sortable b) {
        Shape oneToCompare = null;

        if (b instanceof Shape){
            oneToCompare = (Shape)b;
            if (getArea() < oneToCompare.getArea()) return -1;  // this one is s
maller
            if (getArea() > oneToCompare.getArea()) return 1;   // this one is l
arger
            return 0;                                           // they are the
same
        }
        return 0;
    }
}
```

We've seen most of this before. Oval is similar to Rectangle. We extended Rectangle in previous lessons, but we aren't doing that here.

Let's examine the **compareTo()** method of the Oval class. Since Oval extends Shape, and the **getArea()** method is defined in Shape, we can compare any Shape object to any other Shape object. But first, we need to find out if the parameter **b**, is an instance of **Shape**.

Both the Rectangle and Oval classes extend Shape, which makes them both Shape objects as well, so we'll only need to cast our object to **Shape**.

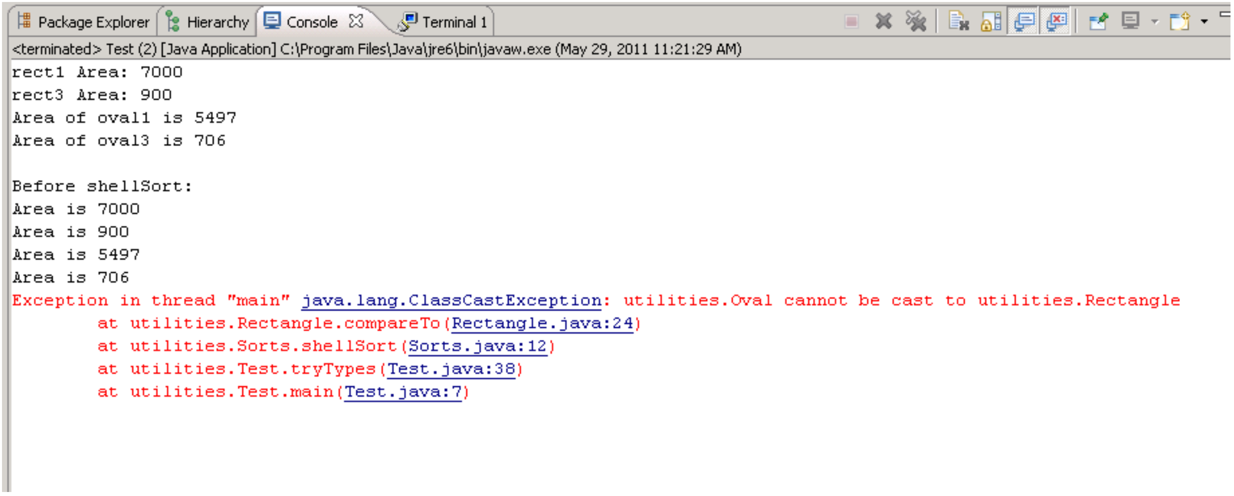
Save **Shape**, **Rectangle**, and **Oval**.

We have lots of changes to check out, so let's rewrite **Test**. Add the **blue** code and remove the **red** code as shown:

CODE TO EDIT:

```
package utilities;

public class Test{

    public static void main(String[] args){
        Test testMe = new Test();
        testMe.tryTypes();
    }

    public void tryTypes(){
        Rectangle rect1 = new Rectangle(60,30,160,100);
        Sortable rect2 = new Rectangle(10,120,40,150);
        Rectangle rect3 = (Rectangle)rect2;
        Sortable [] figures = {rect1, rect3};
        Oval oval1 = new Oval(60,30,160,100);
        Sortable oval2 = new Oval(10,120,40,150);
        Oval oval3 = (Oval)oval2;

        System.out.println("rect1 Area: " + rect1.getArea());
        System.out.println("rect1 Upper Left: (" + rect1.uLX + ","+ rect1.uLY +
")");
        System.out.println("rect3 Area: " + rect3.getArea());
        System.out.println("rect3 Upper Left: (" + rect3.uLX + ","+ rect3.uLY +
")");
        System.out.println("oval1 Area: " + oval1.getArea());
        System.out.println("oval3 Area: " + oval3.getArea());
        System.out.println();

        Sortable [] figures = {rect1, rect3, oval1, oval3};

        int compareTest1 = rect1.compareTo(rect3);
        int compareTest2 = rect3.compareTo(rect1);

        System.out.println("Before shellSort:");
        for (int i = 0; i < figures.length; i++) {
            Rectangle current = (Rectangle)figures[i];
            System.out.println("Area is " + current.getArea());
        }
        if (compareTest1 > compareTest2)
            System.out.println("rect1 is bigger than rect3");
        else
            System.out.println("rect3 is bigger than rect1");
            Shape current = null;
            if (figures[i] instanceof Rectangle)
                current = (Rectangle)figures[i];
            else
                current = (Oval)figures[i];

            System.out.println("Area is " + current.getArea());
        }

        Sorts.shellSort(figures);

        System.out.println("\nAfter shellSort:");
        for (int i = 0; i < figures.length; i++){
            Shape current = null;
            if (figures[i] instanceof Rectangle)
                current = (Rectangle)figures[i];
            else
                current = (Oval)figures[i];
            System.out.println("Area is " + current.getArea());
        }
    }
}
```

Save and run it. Oops—we forgot to allow for comparison of Shapes in the **Rectangle** class's **compareTo()** method:



```
<terminated> Test (2) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (May 29, 2011 11:21:29 AM)
rect1 Area: 7000
rect3 Area: 900
Area of oval1 is 5497
Area of oval3 is 706

Before shellSort:
Area is 7000
Area is 900
Area is 5497
Area is 706
Exception in thread "main" java.lang.ClassCastException: utilities.Oval cannot be cast to utilities.Rectangle
        at utilities.Rectangle.compareTo(Rectangle.java:24)
        at utilities.Sorts.shellSort(Sorts.java:12)
        at utilities.Test.tryTypes(Test.java:38)
        at utilities.Test.main(Test.java:7)
```

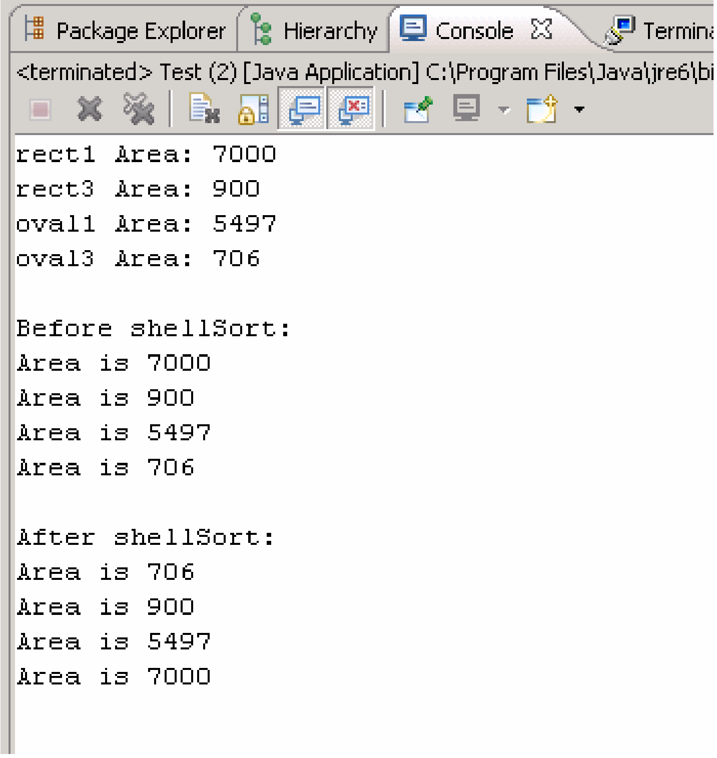
Edit the code in the Rectangle class for the **compareTo()** method. Add the **blue** code and remove the **red** code as shown:

CODE TO EDIT:

```
package utilities;

public class Rectangle extends Shape implements Sortable {
    int uLX, uLY, lRX, lRY;
    private int area;

    public Rectangle(int upperLeftX, int upperLeftY, int lowerRightX, int lowerR
ightY) {
        uLX = upperLeftX;
        uLY = upperLeftY;
        lRX = lowerRightX;
        lRY = lowerRightY;
        setArea();
    }

    private void setArea(){
        area = (lRX - uLX) * (lRY - uLY);
    }

    public int getArea(){
        return area;
    }

    public int compareTo(Sortable b) {
        Rectangle oneToCompare = (Rectangle)b;
        Shape oneToCompare = null;

        if (b instanceof Shape){
            oneToCompare = (Shape)b;
            if (getArea() < oneToCompare.getArea()) return -1;  // this one is s
maller
            if (getArea() > oneToCompare.getArea()) return 1;   // this one is l
arger
            return 0;                                           // they are the
same
        }
        return 0;
    }
}
```

Save it.

Now, go back to the **Test** class and **Run** it. Excellent! The final run will look like this:



<terminated> Test (2) [Java Application] C:\Program Files\Java\jre6\bi

```
rect1 Area: 7000
rect3 Area: 900
oval1 Area: 5497
oval3 Area: 706

Before shellSort:
Area is 7000
Area is 900
Area is 5497
Area is 706

After shellSort:
Area is 706
Area is 900
Area is 5497
Area is 7000
```

Listeners

There are several different kinds of Components that use the **ActionListener** class. The **java.awt.event.ActionEvent** instances that will be passed as the parameter, allow you to call **e.getSource()** method, but this maneuver always returns something of type **Object**. As the programmer, you need to cast it back to a **Button** (or whichever selected Component you're using) to be able to use the variables and methods of the actual instance.

Extending Interfaces

Interfaces can be *extended* as well. In fact, we can even extend interfaces with multiple interfaces.

For example, consider this example interface:

OBSERVE:

```
package utilities;

import java.awt.event.*;

public interface MoreSortables extends Sortable, ActionListener {
    int contrast();
}
```

If your class implements **MoreSortables**, you need to write code for **contrast()** for **MoreSortables** AND **compareTo()** for **Sortable** AND **actionPerformed()** for **java.awt.event.ActionListener**.

Generics

You're doing great work so far, but this lesson is getting pretty darn huge. Maybe you should take a short break and we'll continue with Generics in the next lesson. See you there!

Generics

The Dot Operator

To fully appreciate the concept of callbacks in Java, we need to understand how Java works with the dot operator. We use the dot operator in Java to access **members** of a class and to write expressions. Take a look at this code:

OBSERVE:

```
System.out.println("Hello");
```

java.lang.System has a class variable named **out**, which is of type **PrintStream**, which has a method named **println()**. The **println()** method is overloaded; one of its definitions has a parameter of **String**, so when we type **System.out.println("Hello")**, and pass the String **"Hello"**, Java goes to the **System** class, then to its **out** field, and then to the **out** variable's **println(String s)** method.

If a programmer tries to pass a method name as a parameter, the receiving method will be unable to use it, because Java will not know which class the method came from originally. We can pass *instances* though, because we know which types of objects they are and objects have locations in memory. If we pass a method call as a parameter in a method, the result of that method call gets passed, not the method itself.

Code Reuse and Flexibility

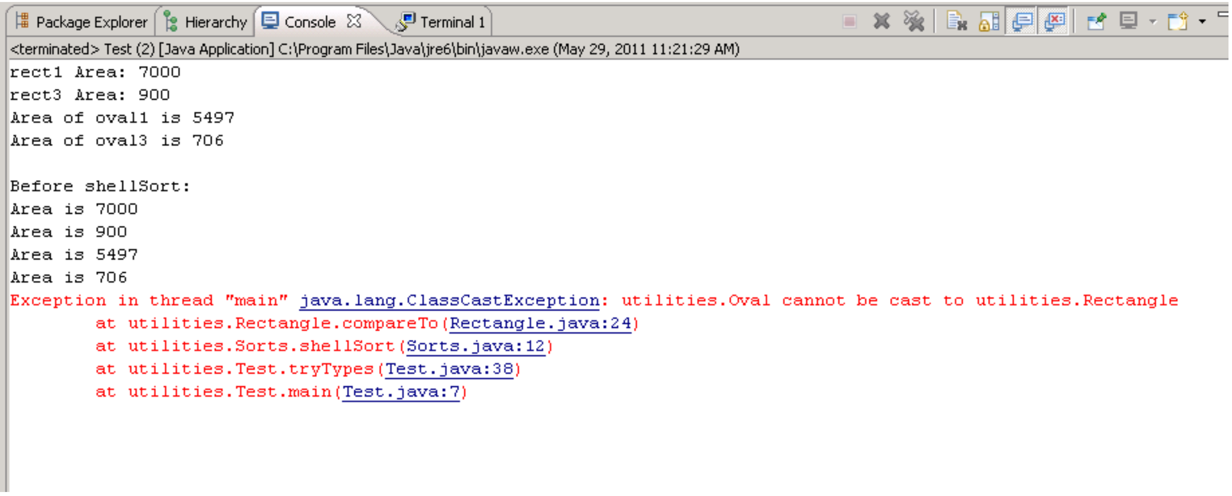
Earlier, we talked about using constructs (such as array.length) in place of supplying specific numbers for loops, so that when the numbers change for different runs, we don't have to alter the code in multiple places to accommodate those numbers that have changed. With version 1.5, Java provided a new construct, *Generics*, which also allows code reuse without requiring multiple changes.

Interfaces allow us to pass parameters (a type of interface); the objects passed are actually instances of different types of classes. Generics allow us flexibility with our interfaces and classes and at the same time, ensure accuracy. Generics also help us to write code more efficiently, and that code will be more secure and easier to use than code littered with Object variables and casts.

Checking Type

When we worked with Shapes in the past, we cast the Sortable interface type **b** to a temporary Shape (**oneToCompare**) within the **compareTo()** method of **Oval**.

When we added **Oval**s and forgot to change the code in **Rectangle**, Eclipse did not warn us of any compiler errors, because up until *runtime*, Eclipse didn't know what would be sent. At runtime, we found that we had errors because Rectangle's **compareTo()** method expected Rectangle objects.



```
<terminated> Test (2) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (May 29, 2011 11:21:29 AM)
rect1 Area: 7000
rect3 Area: 900
Area of oval1 is 5497
Area of oval3 is 706

Before shellSort:
Area is 7000
Area is 900
Area is 5497
Area is 706
Exception in thread "main" java.lang.ClassCastException: utilities.Oval cannot be cast to utilities.Rectangle
        at utilities.Rectangle.compareTo(Rectangle.java:24)
        at utilities.Sorts.shellSort(Sorts.java:12)
        at utilities.Test.tryTypes(Test.java:38)
        at utilities.Test.main(Test.java:7)
```

Generic Example

Vectors

We don't want our applications to give runtime errors. Runtime bugs can be particularly tough to catch because they aren't always readily visible near their source.

Generics provide stability to your code by making runtime bugs visible sooner—at compile time. Let's take a look at generics using a common alternative to arrays, the class **Vector**.

Often we need our code to hold multiple objects. In mathematics, we would create a *set* to hold those objects. In Java, the only mechanism we have learned for this task so far, is the **array**. Arrays have two basic limitations:

1. They must be declared with a specified size.
2. All of their elements must be of the same type.

We made the array for our **figures** of type **Sortable**, because we wanted to call the **shellSort()** method in the **Sorts** class, and **shellSort()** had a parameter of type **Sortable**.

For our graphics drawing project, we want to allow different types of **Shape**s (and icon **Image**s), but we don't know the number of figures the user will draw or add to the screen, so we can't specify a size for an array. One way to address these issues is to use a **Vector**. **Vector**s differ from arrays in that:

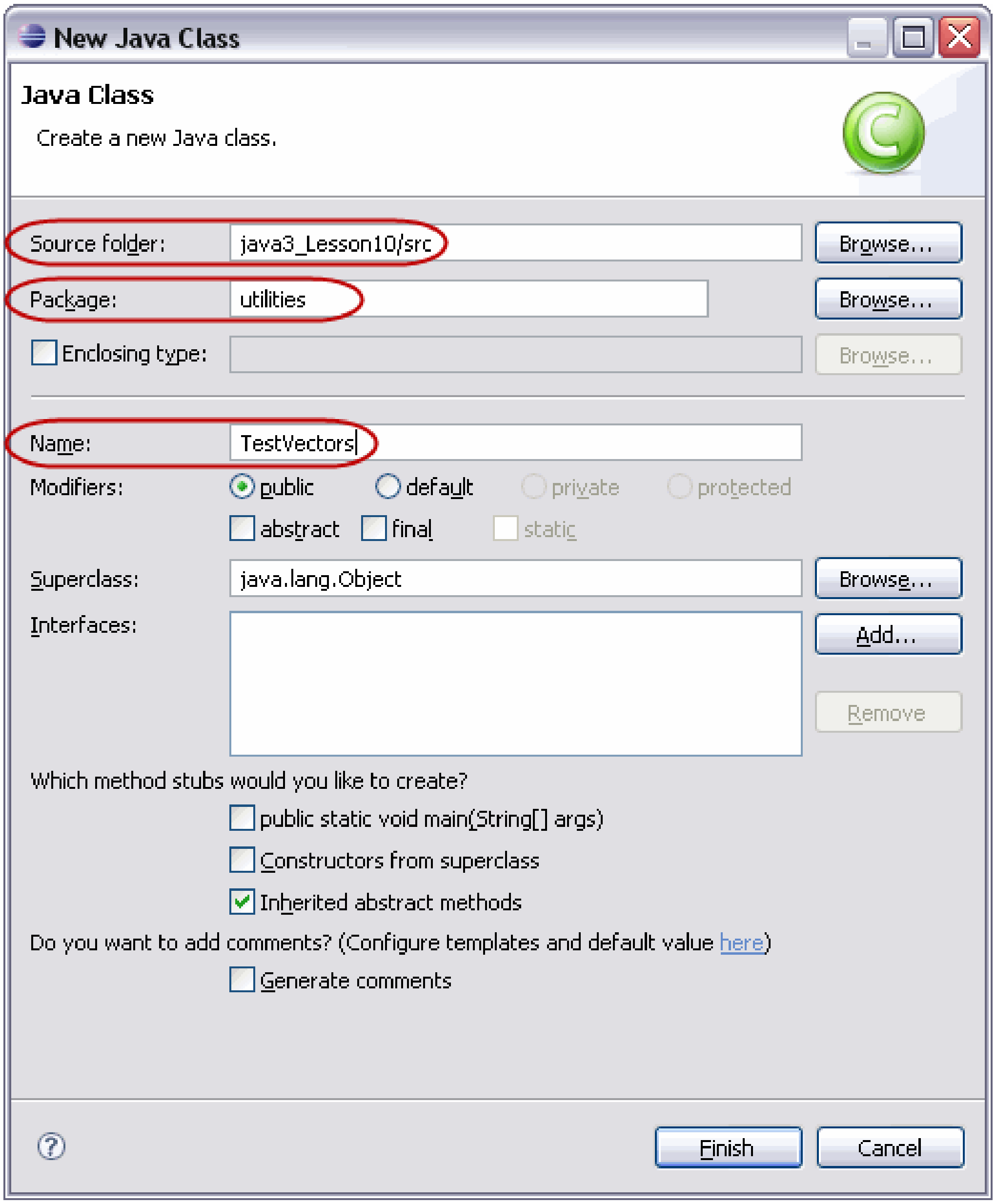
- They can grow dynamically.
- They can hold different types. That is, Java allows you to put different types into a **Vector**, but then it casts them all as **Object**.
- Primitive data types are not allowed unless *wrapped* into their wrapper classes (for example, **int** is wrapped into **java.lang.Integer**).

API In the API, you can find out if Java will wrap these data types automatically. While you're there, go to package **java.util** and read the description of the class **Vector**.

There's some information and terminology in that description that we haven't covered yet, but this part may make sense: "the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created." We'll explain the **<E>**s on this page later. But first, let's find out what can go wrong when we can put multiple types in a collection like a Vector. We'll use the classes we created in **java3_Lesson9** for our example.

Create a new **java3_Lesson10** project, and in that new project, create a **utilities** package. Copy all of the classes from **java3_Lesson9** and put them into the new utilities folder. Go to **java3_Lesson9/src/utilities**, select all of the classes there, right-click for the popup menu, and select **Copy**. Then, go to **java3_Lesson10/src/utilities** folder, right-click, and select **Paste**.

In the **java3_Lesson10** project, create a new class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson10/src

Package: utilities

□ Enclosing type:

Name: TestVectors

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

In **TestVectors**, **type** the **blue** code as shown:

Note This lesson does NOT use the Rectangle class from your projects. It uses the java.awt.Rectangle class.

CODE TO TYPE: TestVectors

```
package utilities;

import java.awt.*;
import java.util.*;

public class TestVectors {

    public static void main(String[] args){
        TestVectors testMe = new TestVectors();
        testMe.tryVectors();
    }

    public void tryVectors(){
        Rectangle rect1 = new Rectangle(60,30,160,100);
        Sortable rect2 = new Rectangle(10,120,40,150);
        Rectangle rect3 = (Rectangle)rect2;
        Oval oval1 = new Oval(60,30,160,100);
        Sortable oval2 = new Oval(10,120,40,150);
        Oval oval3 = (Oval)oval2;
        Point myPoint = new Point(55,55);
        Vector moreFigures = new Vector(2);
        moreFigures.add(rect1);
        moreFigures.add(rect2);
        moreFigures.add(rect3);
        moreFigures.add(oval1);
        moreFigures.add(oval2);
        moreFigures.add(oval3);
        moreFigures.add(myPoint);

        for (int i = 0; i < moreFigures.size(); i++)
        {
            System.out.println("Element "+ i + " is " + moreFigures.elementAt(i)
);
        }
    }
}
```

We see a few warnings, but no errors, so go ahead and **Save** and **Run** it. It seems to run fine, and the Vector does actually hold different types of Objects:

OBSERVE: TestVectors

```
package utilities;

import java.awt.*;
import java.util.*;

public class TestVectors {

    public static void main(String[] args){
        TestVectors testMe = new TestVectors();
        testMe.tryVectors();
    }

    public void tryVectors(){
        Rectangle rect1 = new Rectangle(60,30,160,100);
        Sortable rect2 = new Rectangle(10,120,40,150);
        Rectangle rect3 = (Rectangle)rect2;
        Oval oval1 = new Oval(60,30,160,100);
        Sortable oval2 = new Oval(10,120,40,150);
        Oval oval3 = (Oval)oval2;
        Point myPoint = new Point(55,55);
        Vector moreFigures = new Vector(2);
        moreFigures.add(rect1);
        moreFigures.add(rect2);
        moreFigures.add(rect3);
        moreFigures.add(oval1);
        moreFigures.add(oval2);
        moreFigures.add(oval3);
        moreFigures.add(myPoint);

        for (int i = 0; i < moreFigures.size(); i++)
        {
            System.out.println("Element "+ i + " is " + moreFigures.elementAt(i)
);
        }
    }
}
```

The **Vector** named **moreFigures** is created to hold **2** objects initially. As we add objects to the vector, it grows automatically to accommodate more objects. But this convenience comes at a price. Each time the **Vector** is called upon to grow, it doubles in size, so make sure to set the initial capacity of your vector a bit higher than the maximum number of objects you anticipate it will hold.

When we loop through the **moreFigures** **Vector**, we are no longer dealing with an array, so we can't use the **length** constant from an array. We have to use the **Vector**'s **size()** method, which gives us the number of objects in the **Vector**.

The elements at 1 and 2, and at 4 and 5, are exactly the same **Object**. Do you see why? When we *made* **rect3** and **oval3**, we did not make **new** objects. Instead, we **cast** an existing object to be *seen* differently from how it was declared earlier. They do point to the same place though, so be careful making changes to either.

Since each element in the **Vector** is an **Object**, the method **elementAt(i)** for **Vector** can print out each type. However, we didn't cast anything, so the **Vector** presents all of these elements as **Object**s, and we get only **Object** information.

Let's cast the elements *to* something.

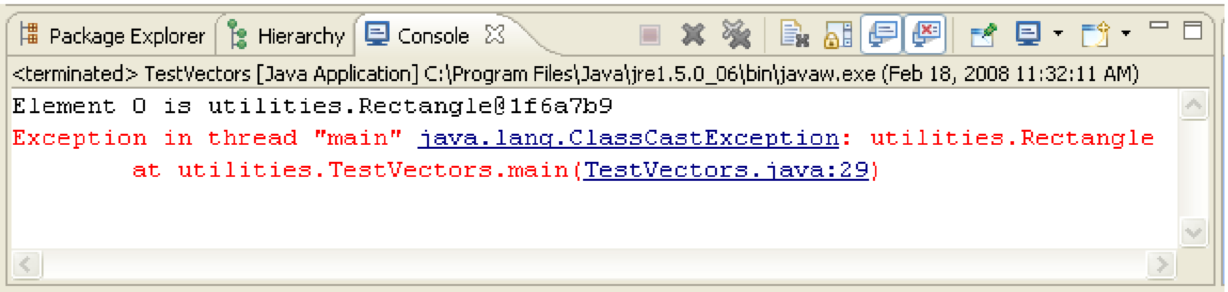
Edit **TestVectors** by adding the **blue** code as shown:

CODE TO TYPE: TestVectors

```
package utilities;

import java.awt.*;
import java.util.*;

public class TestVectors {

    public static void main(String[] args){
        TestVectors testMe = new TestVectors();
        testMe.tryVectors();
    }

    public void tryVectors(){

        Rectangle rect1 = new Rectangle(60,30,160,100);
        Sortable rect2 = new Rectangle(10,120,40,150);
        Rectangle rect3 = (Rectangle)rect2;
        Oval oval1 = new Oval(60,30,160,100);
        Sortable oval2 = new Oval(10,120,40,150);
        Oval oval3 = (Oval)oval2;
        Point myPoint = new Point(55,55);

        Vector moreFigures = new Vector(2);
        moreFigures.add(rect1);
        moreFigures.add(rect2);
        moreFigures.add(rect3);
        moreFigures.add(oval1);
        moreFigures.add(oval2);
        moreFigures.add(oval3);
        moreFigures.add(myPoint);

        for (int i = 0; i < moreFigures.size(); i++)
        {
            System.out.println("Element "+ i + " is " + moreFigures.elementAt(i)
);
            Point myBad =(Point)moreFigures.elementAt(i);
            System.out.println("Vector Element "+ myBad);
        }
    }
}
```

Note There are no compile-time errors or warnings at these new lines.

Save and **Run** it.



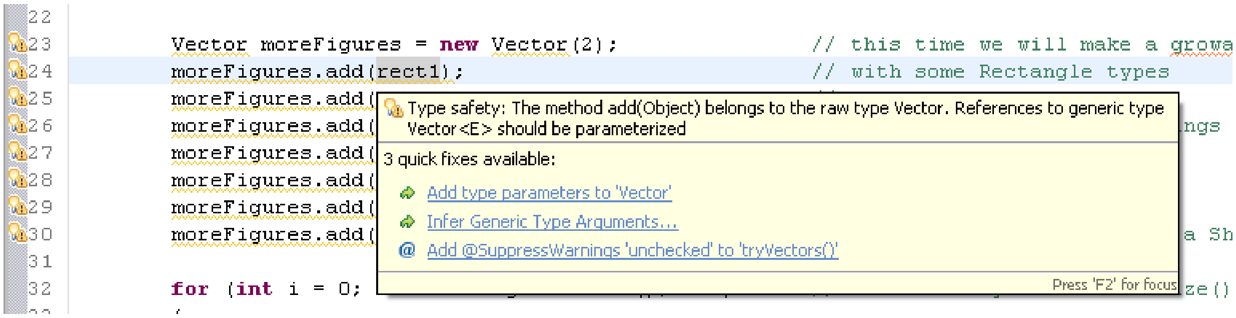
<terminated> TestVectors [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Feb 18, 2008 11:32:11 AM)

```
Element 0 is utilities.Rectangle@1f6a7b9
Exception in thread "main" java.lang.ClassCastException: utilities.Rectangle
        at utilities.TestVectors.main(TestVectors.java:29)
```

Even though we can't cast a **Rectangle** to a **Point**, we weren't given any errors. A problem like this may go unnoticed until **runtime**. Fortunately, in version 1.5, Java made **Vectors** a Generic class, so if we use the generics framework properly, we **cannot** cast incorrectly.

Vectors Using Generics

There are cautions on each line of our **TestVectors** class, located wherever we try to add elements to the **Vector**:



```
        Vector moreFigures = new Vector(2);             // this time we will make a growa
        moreFigures.add(rect1);                         // with some Rectangle types
        moreFigures.add(
        moreFigures.add(                                                            ngs
        moreFigures.add(
        moreFigures.add(
        moreFigures.add(
        moreFigures.add(                                                            a Sh

        for (int i = 0;                                                             ze()
```

Type safety: The method add(Object) belongs to the raw type Vector. References to generic type Vector<E> should be parameterized

3 quick fixes available:

Add type parameters to 'Vector'
Infer Generic Type Arguments...
Add @SuppressWarnings 'unchecked' to 'tryVectors()'
Press 'F2' for focus

Generic classes force us to specify a type by **parameterizing** the class. We specify a type that is expected within the **Vector**. In the past, Java used the class **Object** for parameterization by default, because **all** classes in Java inherit from **Object**. This allowed programmers to remove elements and then cast them incorrectly without being aware of potential errors until runtime. But now, Java demands that we specify a type, so casting can be checked. We see these warnings in our code because we did not parameterize:

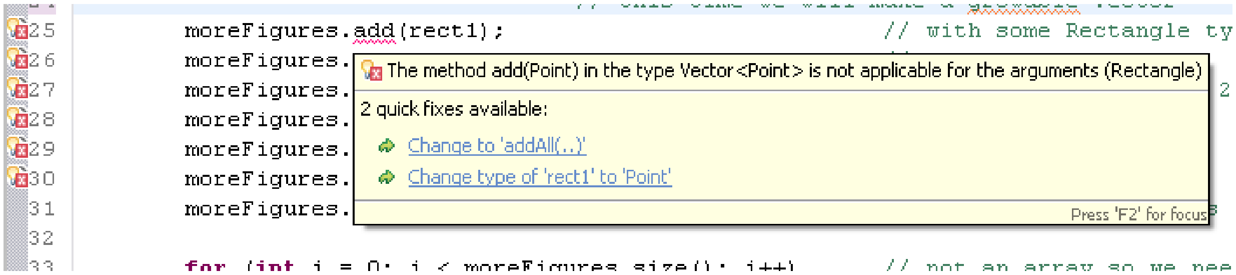
- **Type safety:** we may have a problem with the *safety* of our types when casting.
- **The method add(Object) belongs to the raw type Vector:** we did not specify a type, so the compiler will use the **raw type**, which by default is **Object**.
- **References to generic type Vector <E> should be parameterized:** We should use the generic type Vector <E> so checks can be done safely. **Object** is not specific enough.

Edit **TestVectors** as shown below (we are casting to **Point**, so we will parameterize it accordingly):

CODE TO TYPE: TestVectors

```
package utilities;

import java.awt.*;
import java.util.*;

public class TestVectors {

    public static void main(String[] args){
        TestVectors testMe = new TestVectors();
        testMe.tryVectors();
    }

    public void tryVectors(){
        Rectangle rect1 = new Rectangle(60,30,160,100);
        Sortable rect2 = new Rectangle(10,120,40,150);
        Rectangle rect3 = (Rectangle)rect2;
        Oval oval1 = new Oval(60,30,160,100);
        Sortable oval2 = new Oval(10,120,40,150);
        Oval oval3 = (Oval)oval2;
        Point myPoint = new Point(55,55);

        Vector <Point> moreFigures = new Vector <Point>(2);
        moreFigures.add(rect1);
        moreFigures.add(rect2);
        moreFigures.add(rect3);
        moreFigures.add(oval1);
        moreFigures.add(oval2);
        moreFigures.add(oval3);
        moreFigures.add(myPoint);

        for (int i = 0; i < moreFigures.size(); i++)
        {
            System.out.println("Element "+ i + " is " + moreFigures.elementAt(i)
);
            Point myBad =(Point)moreFigures.elementAt(i);
            System.out.println("Vector Element "+ myBad);
        }
    }
}
```

Well, that didn't help. Now we have lots of **errors** where we added rectangles and ovals:



```
        moreFigures.add(rect1);                         // with some Rectangle ty
        moreFigures.
        moreFigures.
        moreFigures.
        moreFigures.
        moreFigures.
        moreFigures.
```

The method add(Point) in the type Vector<Point> is not applicable for the arguments (Rectangle)

2 quick fixes available:

- Change to 'addAll(..)'
- Change type of 'rect1' to 'Point'

Press 'F2' for focus

API Go back to the API and look at **Vector**. See all of those **<E>**s? When we use Vector **<Point>** moreFigures = new Vector **<Point>**(), we are telling the compiler that in this instance of **Vector**, we are parameterizing **<E>** to **<Point>**. So, according to the API, each instance of **<E>** in this particular piece of code, becomes **<Point>**. The method **add(E e)** will become **add(Point e)**, the method **elementAt** will return an element of type **Point**, and we will get errors.

Let's fix those pesky errors. **Edit TestVectors** as shown. Remove the **red** code:

CODE TO TYPE: TestVectors

```
package utilities;

import java.awt.*;
import java.util.*;

public class TestVectors {

    public static void main(String[] args){
        TestVectors testMe = new TestVectors();
        testMe.tryVectors();
    }

    public void tryVectors(){
        Rectangle rect1 = new Rectangle(60,30,160,100);
        Sortable rect2 = new Rectangle(10,120,40,150);
        Rectangle rect3 = (Rectangle)rect2;
        Oval oval1 = new Oval(60,30,160,100);
        Sortable oval2 = new Oval(10,120,40,150);
        Oval oval3 = (Oval)oval2;
        Point myPoint = new Point(55,55);

        Vector <Point> moreFigures = new Vector <Point> (2);
        moreFigures.add(rect1);
        moreFigures.add(rect2);
        moreFigures.add(rect3);
        moreFigures.add(oval1);
        moreFigures.add(oval2);
        moreFigures.add(oval3);
        moreFigures.add(myPoint);

        for (int i = 0; i < moreFigures.size(); i++)
        {
            System.out.println("Element "+ i + " is " + moreFigures.elementAt(i)
);
            Point myBad =(Point)moreFigures.elementAt(i);
            System.out.println("Vector Element "+ myBad);
        }
    }
}
```

That clears the compile-time errors. **Save** and **Run** it. Our code should be free of runtime errors too.

Now, we'll get rid of all of the cautions. They showed up because we didn't use the variables. So we'll use them now—but not in the Vector:

Edit TestVectors. Add the **blue** code and remove the **red** code as shown:

CODE TO EDIT: TestVectors

```
package utilities;

import java.awt.*;
import java.util.*;

public class TestVectors {

    public static void main(String[] args){
        TestVectors testMe = new TestVectors();
        testMe.tryVectors();
    }

    public void tryVectors(){
        Rectangle rect1 = new Rectangle(60,30,160,100);
        Sortable rect2 = new Rectangle(10,120,40,150);
        Rectangle rect3 = (Rectangle)rect2;
        Oval oval1 = new Oval(60,30,160,100);
        Sortable oval2 = new Oval(10,120,40,150);
        Oval oval3 = (Oval)oval2;
        Point myPoint = new Point(55,55);

        Sortable [] figures = {rect1, rect3, oval1, oval3};
        Vector <Point> moreFigures = new Vector <Point>(2);
        moreFigures.add(myPoint);

        for (int i = 0; i < moreFigures.size(); i++)
        {
            System.out.println("Element " + i + " is " + moreFigures.elementAt(i
));
            Point myBad = (Point)moreFigures.elementAt(i);
            System.out.println("Vector Element " + myBad);
        }

        System.out.println();
        for (int i = 0; i < figures.length; i++)
        {
            System.out.println("Array Element " + i + " is " + figures[i]);
            Shape myBad =(Shape)figures[i];
            System.out.println("Array Element " + myBad);
        }
    }
}
```

Nice! No warnings and no errors. We were also able to remove the cast to **Point** because our **Vector**, **moreFigures**, can only hold **Point** objects now.

Save and **Run** it.

Generals on Generics

Now that you know what the **<E>**s represent, you'll probably take more notice of them in the Java API. By convention, we use lowercase letters for variables for Class **members**. Parameter names for **Type**s are comprised of one uppercase letter. We differentiate parameters for **Class Types** using various uppercase letters. The most commonly used type parameter names are:

- **E:** Element (used extensively by the Java Collections Framework—we'll go over it in the next lesson.)
- **K:** Key
- **N:** Number
- **T:** Type
- **V:** Value
- **S,U,V, and so on:** 2nd, 3rd, 4th types (methods can have multiple parameters of different types.)

Because the API always illustrates the most general usage, we commonly see parameters of **<T>** for **Type** and **<E>** for **Element**. And there are additional parameter type limitations in generics including:

- bounds (for example, **<T** extends Shape **>**)
- wildcards (for example, **<?** extends Shape **>**)

There's still a **lot** more to the Generic Framework, but much of it is beyond the scope of this class. Our focus is on **using** the classes that Java provides, so we won't be writing our own generic classes, or experience their full potential just yet. Although in the next lesson, we will look at the Collection Framework which is probably the most extensive applied use of generics. Here are some links to more information on generics:

- Oracle's generics page with a link to the Generics guide (as a pdf), also available as html.
- **Learning the Java Language**, the Generics Tutorial.

Don't wrestle with this too much now. Like most new skills, the generics tool is best learned through many examples and lots of practice over time.

The Collection Framework

Collections

Computers hold and sort through collections of information continually, looking for specific items. We need efficient ways to program and manipulate those collections. Java provides a **Collection Framework** that contains numerous types to hold, access, and manipulate our collections. This Collection Framework saves programmers lots of time and effort because it allows them to avoid writing and rewriting code for tasks the Collection Framework already manages. Collections are also a part of the Generic Framework. Since we usually think of things in collections as **elements**, the common parameter variable used to specify our elements' types is **<E>**.

So far, we have seen two constructs that hold collections: arrays (which are **not** classes) and **Vector**s. In this lesson, we'll look at some additional constructs. We'll check out the similarities that cause a class to be included within the framework, as well as some of the "concrete collections" that have been sitting in the API just waiting for us to find them.

Empowered by Collections

Using collections, we can:

- determine whether anything is in the collection.
- count the items in the collection.
- search for specific items.
- order (sort) the items.
- store elements.
- retrieve elements.
- empty (clear) the collection.

Interfaces specify which tasks a collection is able to perform. Java provides multiple *core* collection interfaces:

- **Collection**: the root of the collection hierarchy. It represents a group of objects we call *elements*.
- **Set**: a collection that cannot contain duplicate elements.
- **List**: an ordered collection. Lists may contain duplicates.
- **Queue**: a collection that holds multiple elements awaiting processing (like a line at a bank). Queues typically operate in a FIFO (First In First Out) order.
- **Map**: an object that maps keys to values (for example, common keys for the IRS would be social security numbers). A Map cannot contain duplicate keys and each key can map at most, to one value.

Classes in the collections framework **implement** one or more of those **interfaces**. In addition, Collections are also a part of the Generic Framework, and since we usually think of things in collections as **elements**, the common parameter variable used to specify our elements' types is **<E>**.

ArrayList

The **ArrayList** class is similar to our **Vector** class. **ArrayList** allows the collection to grow dynamically, in the same way that the **Vector** class does. You must specify a length for your arrays, and they must be large in order to avoid "array out of bounds" errors. Empty array locations take up lots of space as well.

API Go to the API package **java.util**. Read about the interfaces and their use of generics. Scroll down to the classes, then click on **ArrayList**:

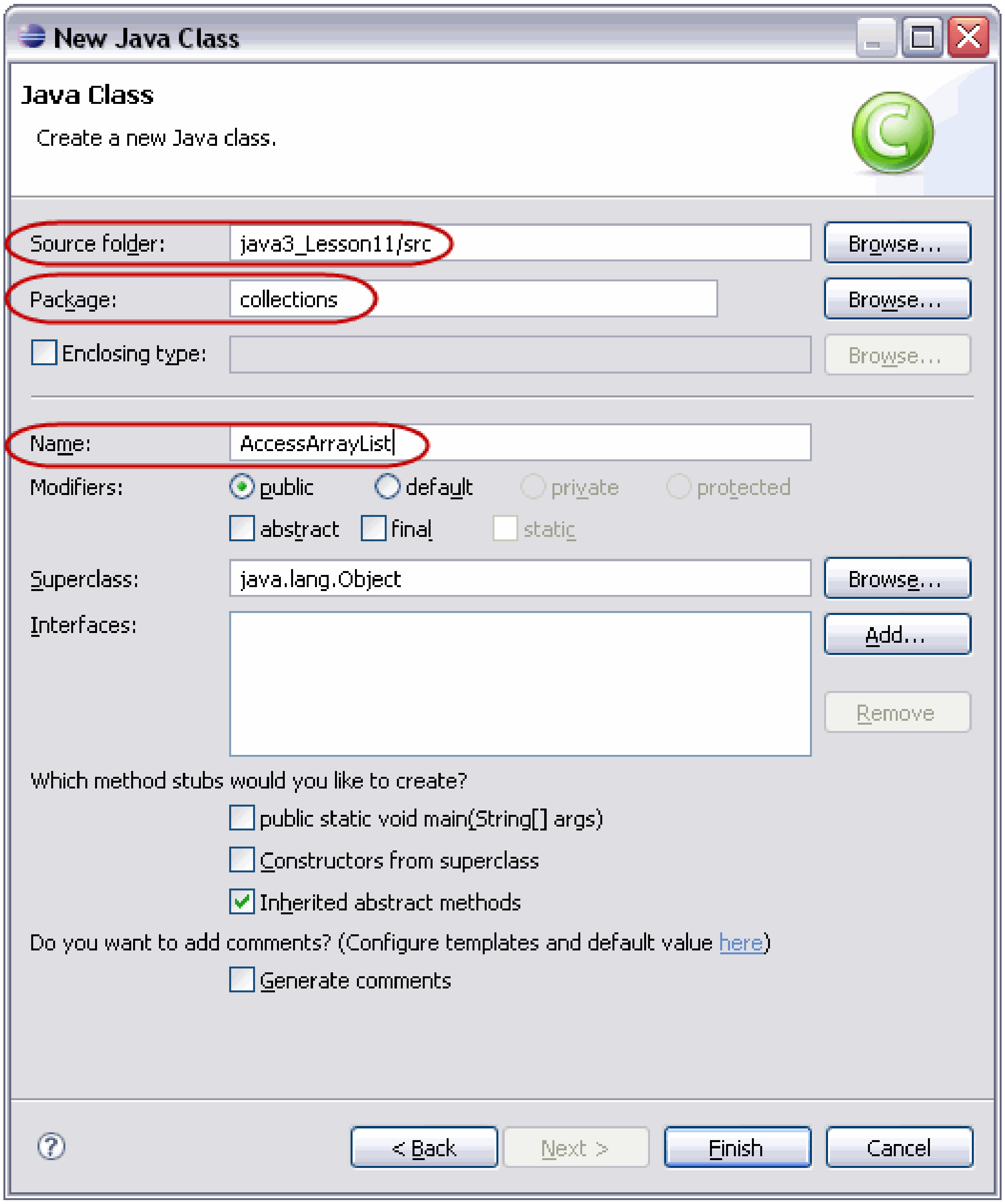
- **ArrayList** allows collections of different types of objects by implementing the generics framework.
- **ArrayList implements** the **interfaces** of **Iterable**, **Collection**, **List**, and **RandomAccess**.

API Now take a look at each of those interfaces in the API. Use the Back button to return to the **ArrayList** API page each time.

1. **Iterable** means that you can iterate, that is, you can go through the list, one item at a time.
2. **Collection** has useful methods such as **add()**, **remove()**, **isEmpty()**, and **size()**.
3. **List** (which is a subinterface of **Collection** and **Iterable**), has methods of **add()**, **get()**, **remove()**, and **toArray()**.
4. **RandomAccess** files permit nonsequential, or random, access to a file's contents.

The classes in the collections framework have implemented the ArrayList interfaces for us already! Let's write a couple of examples to demonstrate the typical access methods available in collections.

Create a new **java3_Lesson11** project. If you're given the option to "Open Associated Perspective", click **No**. In this project, create a new Class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson11/src

Package: collections

□ Enclosing type:

Name: AccessArrayList

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 AccessArrayList as shown below in **blue**:

CODE TO TYPE: AccessArrayList

```
package collections;

import java.util.ArrayList;

public class AccessArrayList{

    public static void main (String[] args){
        AccessArrayList testing = new AccessArrayList();
        testing.tryThis();
    }

    public void tryThis(){
        ArrayList <String> beatles = new ArrayList<String>();

        System.out.println ("Size of beatles at start: " + beatles.size());
        beatles.add ("John");
        beatles.add ("Paul");
        beatles.add ("George");
        beatles.add ("Ringo");
        beatles.add ("MetamorphosisGuy");

        System.out.println (beatles);
        System.out.println ("Size of beatles after adding: " + beatles.size());

        int location = beatles.indexOf ("MetamorphosisGuy");
        beatles.remove (location);

        System.out.println ("After removing location "
              + location + "\n    beatles are " + beatles);
        System.out.println ("At index 1 is " + beatles.get(1));

        beatles.add (2, "Mick");

        System.out.println ("After adding Mick at location 2 \n    "
              + beatles);
        System.out.println ("Size of beatles: " + beatles.size());
    }
}
```

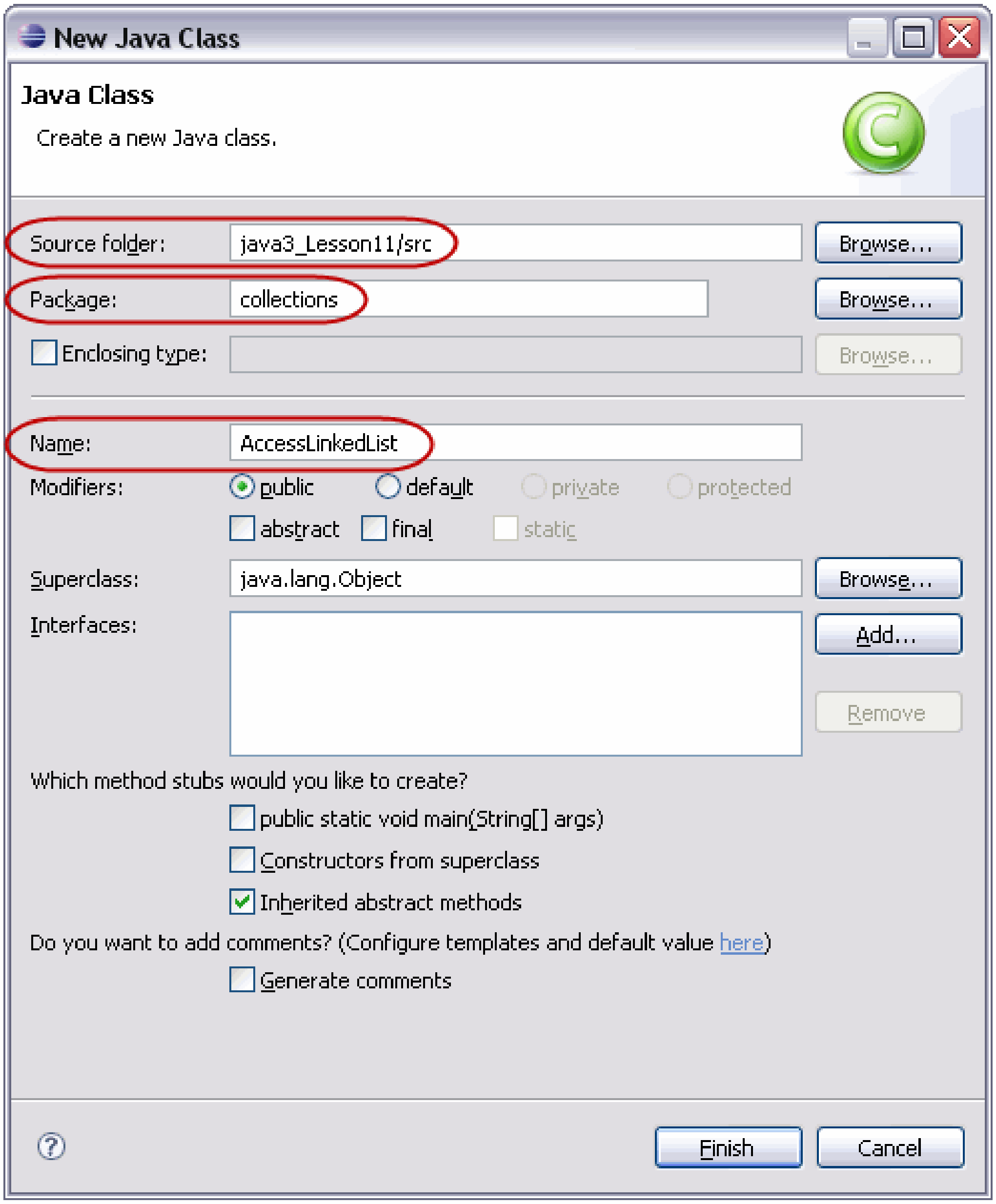
Save and **Run** it. Compare the results in the console with the code to see how the methods worked. Everything works pretty much as expected. Nice.

LinkedList

Let's try another similar example of a collection. Because it's common to look through lists (and in general, collections), the classes in the Collections Framework **implement** the **interface** **Iterable**, which means the method **iterator()** has been **implement**ed. Using the **Iterable** interface is much easier than writing lots of **for** loops to go through our lists and collections.

The tools we use most often to wade through collections are **iterators** and the **for-each** construct. We'll demonstrate both.

In the java3_Lesson11 project, create another new class, as shown:



Source folder: java3_Lesson11/src

Package: collections

□ Enclosing type:

Name: AccessLinkedList

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)

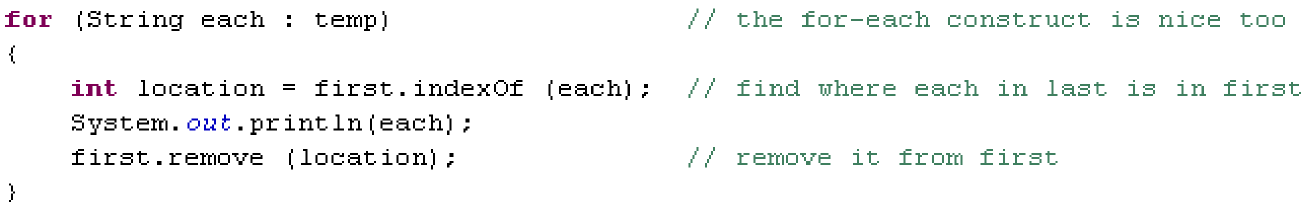
□ Generate comments

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

CODE TO TYPE: AccessLinkedList

```
package collections;

import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;

public class AccessLinkedList {

    public static void main (String[] args){
        AccessLinkedList testing = new AccessLinkedList();
        testing.tryThis();
    }

    public void tryThis(){
        List<String> first = new LinkedList<String>();
        first.add ("Mick");
        first.add ("Keith");
        first.add ("Charlie");
        first.add ("Bill");
        first.add ("Ron");
        System.out.println ("First:  " + first);

        List<String> last = new LinkedList<String>();
        last.add ("Jagger");
        last.add ("Richards");
        last.add ("Watts");
        last.add ("Wyman");
        last.add ("Wood");
        System.out.println ("Last: " + last);

        ListIterator<String> firstIter = first.listIterator();
        Iterator<String> lastIter = last.iterator();

        while (lastIter.hasNext()){
            if (firstIter.hasNext())
                firstIter.next();
            firstIter.add(lastIter.next());
        }
        System.out.println("\nMerged all into first:\n");
        System.out.println(first);

        List <String> temp = new LinkedList<String>();

        lastIter = last.iterator();
        while (lastIter.hasNext()){
            lastIter.next();
            if (lastIter.hasNext()){
                temp.add(lastIter.next());
                lastIter.remove();
            }
        }

        System.out.println("\nRemoved every other element in last\n");
        System.out.println("Last has become: " + last);

        first.removeAll(last);

        System.out.println("First is now: " + first);

        for (String each : temp)
        {
         int location = first.indexOf (each);
         System.out.println(each);
            first.remove (location);
        }
        System.out.println("First is back to: " + first);
    }
}
```

Save and **Run** it. Compare the results in the console with the code. The method **next()** is used to iterate through the set/collection. The last instance of the **for-each** construct could have been performed using a collections bulk operation, specifically, this code:



```
for (String each : temp)                    // the for-each construct is nice too
{
    int location = first.indexOf (each);    // find where each in last is in first
    System.out.println(each);
    first.remove (location);                // remove it from first
}
```

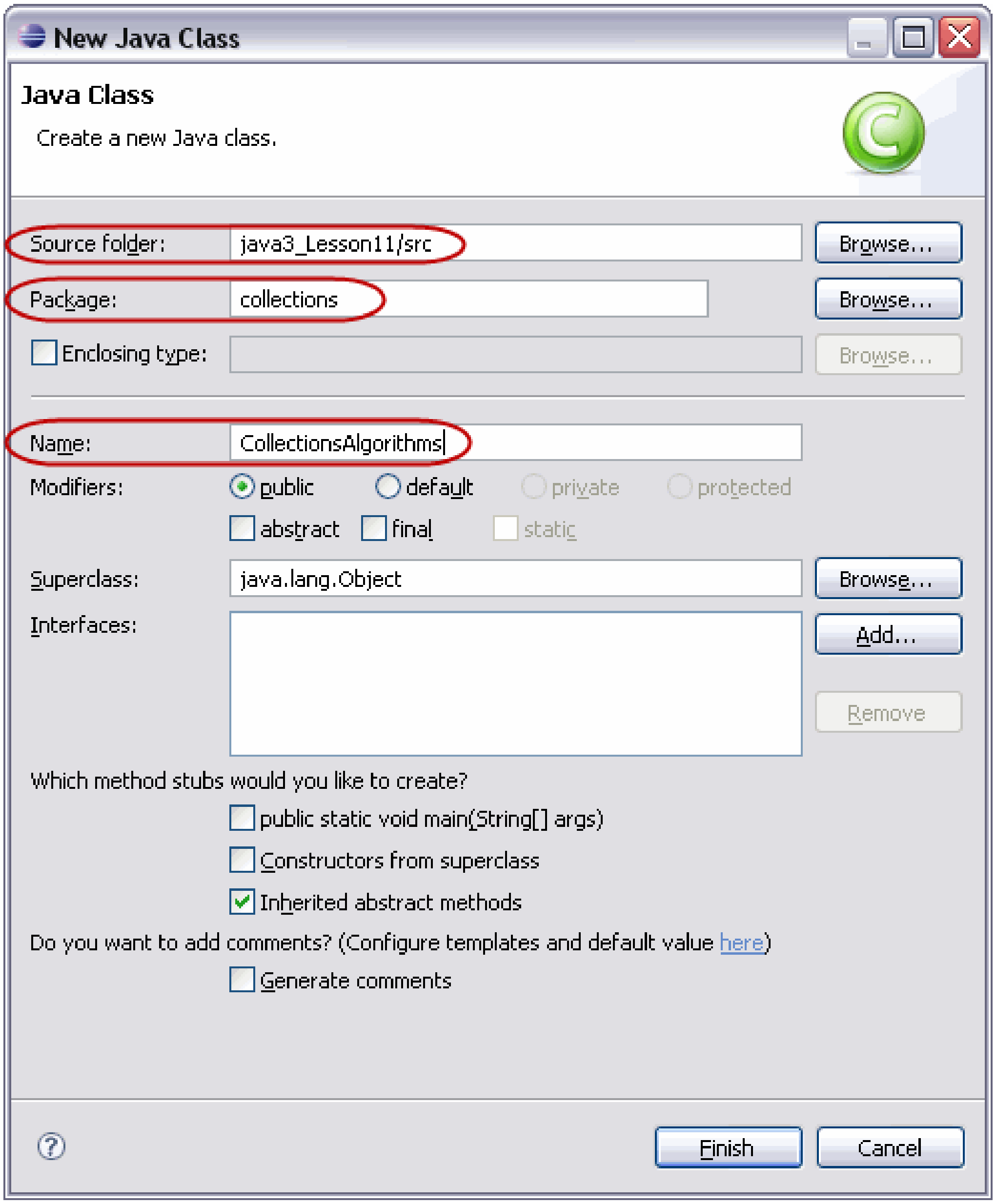
With the exception of the **println** command, this code does exactly the same thing as **first.removeAll(temp)**.

Collections: Things Java Has Already Written for Us

In the last couple of lessons we sorted elements in **array**s and **Vector**s by writing the sorts ourselves. The classes **ArrayList** and **LinkedList** used the **interface methods** to access their members. Sometimes we'll want to arrange our collections in a specific order as well. Sorting items in a collection is almost as common as searching for elements in a collection. The collection framework not only provides methods to access elements, but it provides algorithms to manipulate entire collections. These algorithms are made available through **static** methods in the **Collections** class. Be aware that this is a **class** (that **s** on the end of **Collections** is significant).

Let's see what we can find in the API. Go to the **java.util** package. Scroll down to the **Class Summary** and choose **Collections**. Take a look at its methods. They're all **static**, so we can access them using **Collections.methodName()**. Also, they almost all have parameters of **List**. Most of the classes in the collections framework **implement** the **List** interface, so they're relatively easy to use.

In the java3_Lesson11 project, create a new class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson11/src

Package: collections

□ Enclosing type:

Name: CollectionsAlgorithms

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 CollectionsAlgorithms as shown below in **blue** (we're not actually writing any **Collections** code ourselves, except to generate the elements in a **List**. The **main()** method makes calls to the **Collections** class and then prints results):

CODE TO TYPE: CollectionsAlgorithms

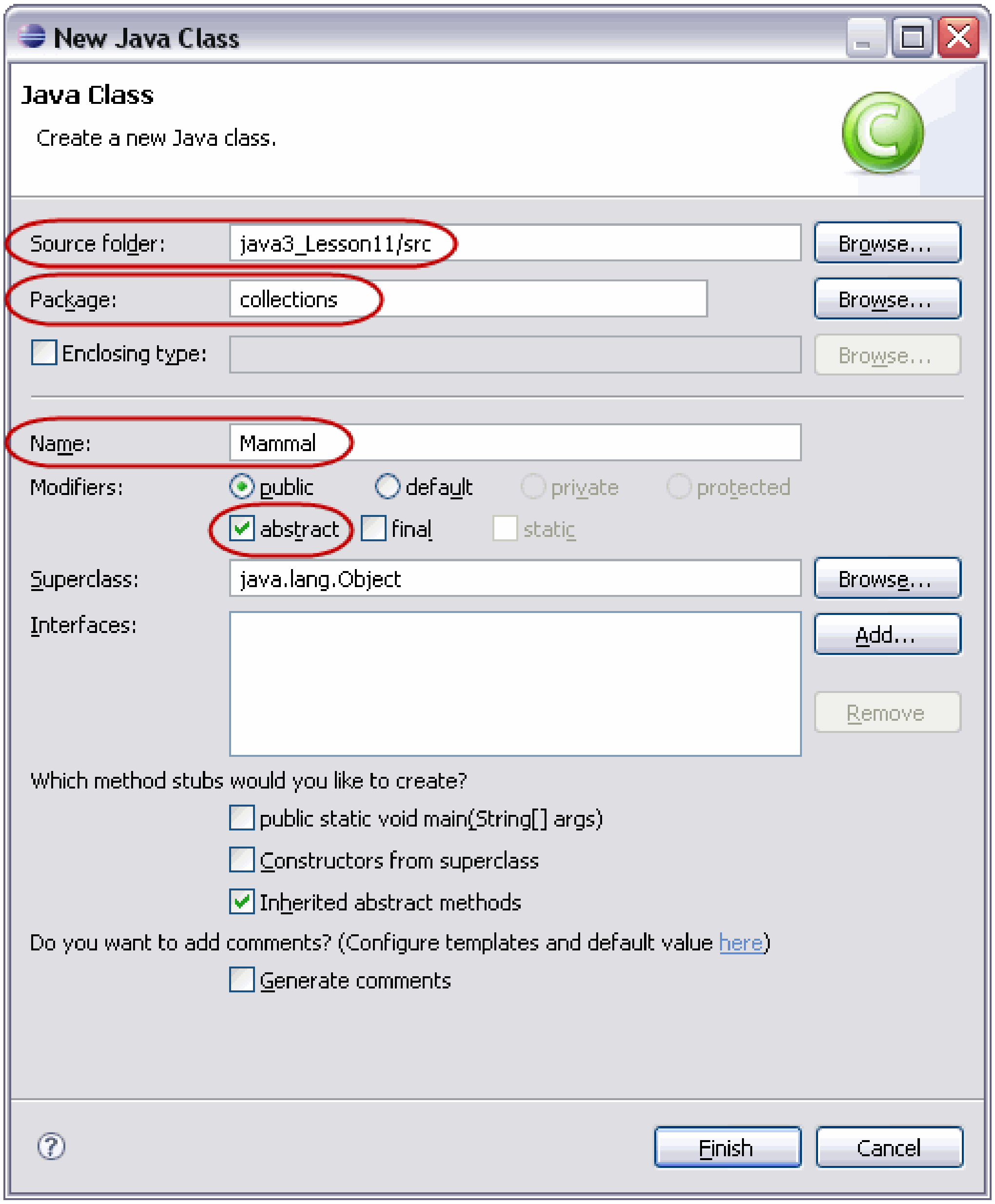
```
package collections;

import java.util.*;

public class CollectionsAlgorithms {

    private Integer numberGenerator(){
        int randomInt = (int)(Math.random() * 100);
        return Integer.valueOf(randomInt);
    }

    public List<Integer> createAList(int howMany){
        List <Integer> createdList = new ArrayList<Integer>(howMany);
        for(int i =0; i < howMany; i++)
            createdList.add(numberGenerator());
        return createdList;
    }

    public static void main (String[] args){
        CollectionsAlgorithms testMe = new CollectionsAlgorithms();
        testMe.tryThis();
    }

    public void tryThis(){
        List<Integer> myList = createAList(7);
        System.out.println("Created list: " + myList);

        List<Integer> myCopy = createAList(7);
        System.out.println("Second list:  " + myCopy);

        Collections.fill(myCopy, Integer.valueOf(0));
        System.out.println("Second list with 0s: " + myCopy);

        Collections.copy(myCopy, myList);
        System.out.println("Copied first into second list so "
                   + "we can mess with it: \n       " + myCopy);

        System.out.println();
        Collections.sort(myCopy);
        System.out.println("Sorted list: " +myCopy);

        int foundFirst = Collections.binarySearch(myCopy, myList.get(0));
        System.out.println("Found first in original list at index "
                   + foundFirst + " in sorted list " );

        Collections.reverse(myCopy);
        System.out.println("Reversed order of list: " +myCopy);

        Collections.shuffle(myCopy);
        System.out.println("Shuffled list: " +myCopy);

        Integer min = Collections.min(myCopy);
        System.out.println("Min value is: " + min.intValue()
                   + ", Max value is: " + Collections.max(myCopy).intValue());

        myCopy = Collections.emptyList();
        System.out.println("Emptied list: " +myCopy);

        System.out.println("Still have original created list: " + myList);
    }
}
```

Save and **Run** it. Compare the results in the console. Also, check out **java.util.Collections** in the API to see the specifications of the methods used.

Comparator

Since we can specify our own way to compare elements by **implement**ing the **java.util.Comparator** interface, we have unlimited potential for sorting different types of elements, using different criteria. Suppose, for example, we want to compare various types of mammals using certain criteria.

In the java3_Lesson11 project, create a new class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson11/src

Package: collections

☐ Enclosing type:

Name: Mammal

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)

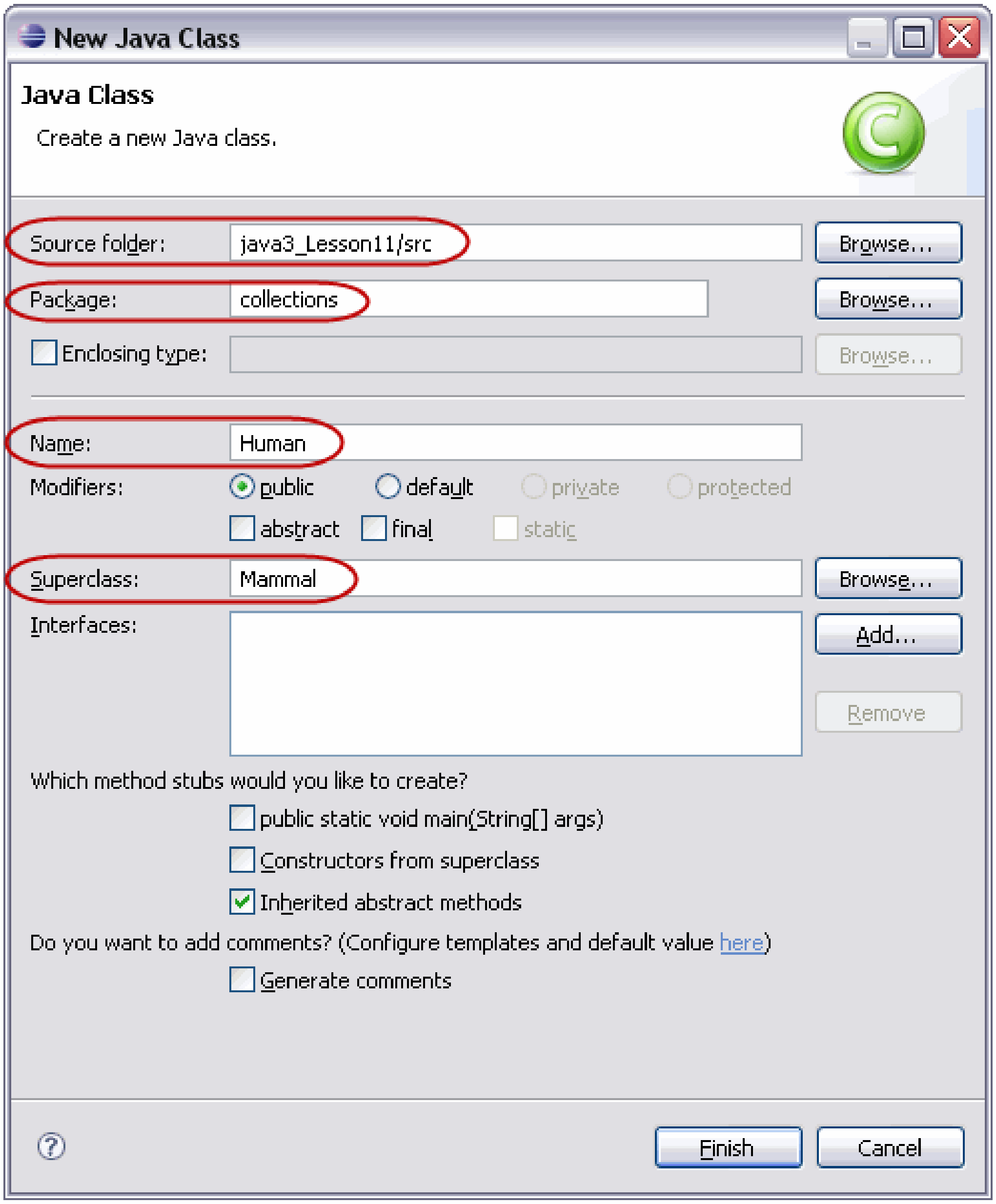
☐ Generate comments

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

CODE TO TYPE: Mammal

```
package collections;

public abstract class Mammal {
    protected String name;

    public Mammal(String who){
        name = who;
    }

    public String getName(){
        return name;
    }

    public abstract double getHeight();
    public abstract double getSpeed();
}
```

Create another new class in the java3_Lesson11 project as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson11/src

Package: collections

□ Enclosing type:

Name: Human

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: Mammal

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)

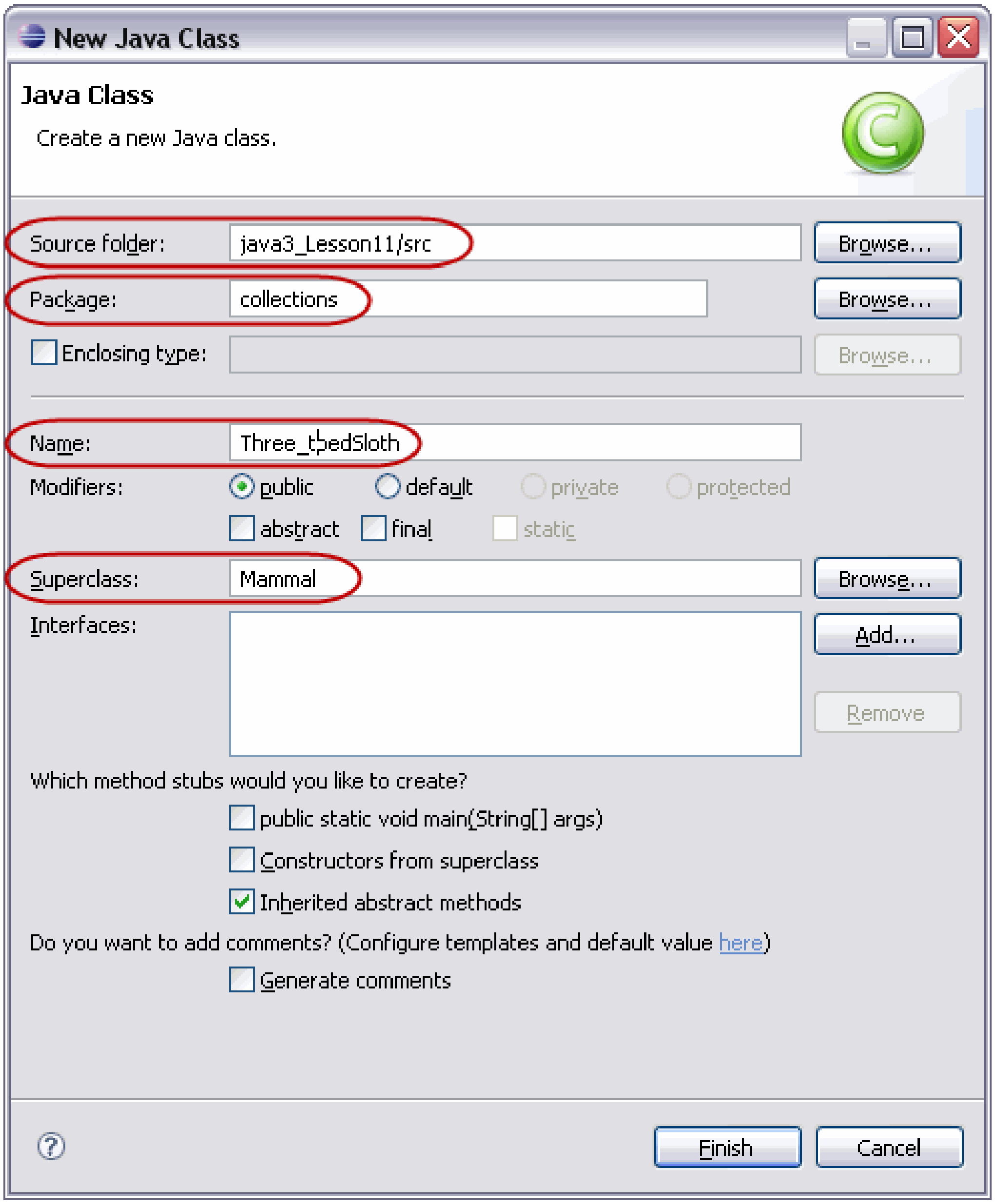
□ Generate comments

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

CODE TO TYPE: Human

```
package collections;

public class Human extends Mammal{
    private double runningSpeed = 10.00;
    private double height = 1.6;

    public Human(String who){
        super(who);
    }

    public double getHeight(){
       return height;
    }
    public double getSpeed(){
       return runningSpeed;
    }

}
```

Create another new class in the java3_Lesson11 project as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson11/src

Package: collections

□ Enclosing type:

Name: Three_toedSloth

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: Mammal

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)

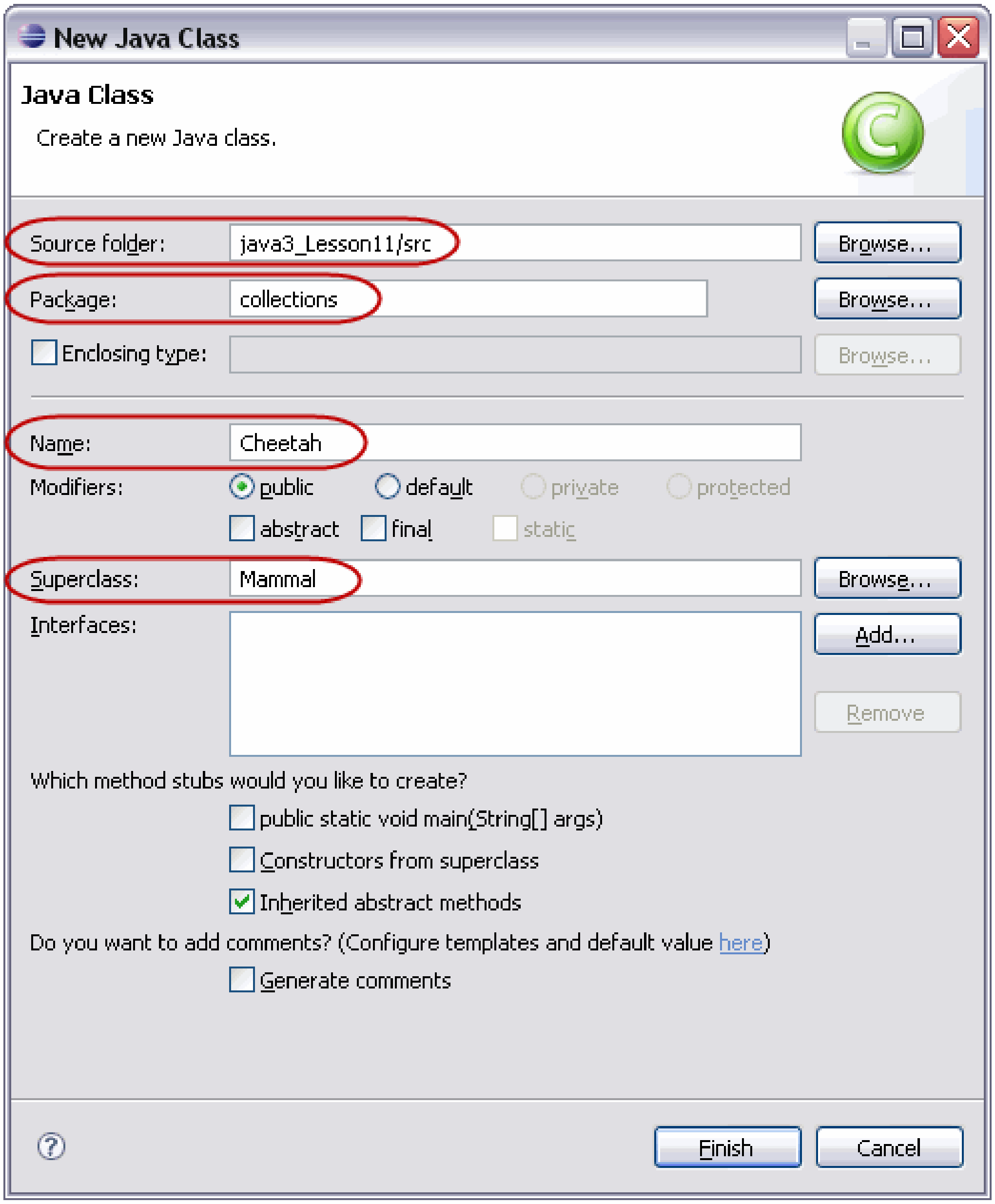
□ Generate comments

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

CODE TO TYPE: Three_toedSloth

```
package collections;

public class Three_toedSloth extends Mammal{
    private double runningSpeed = 0.15;
    private double height = 0.58;

    public Three_toedSloth(String who){
        super(who);
    }

   public double getHeight(){
      return height;
   }
   public double getSpeed(){
      return runningSpeed;
   }

}
```

Create another new class in the java3_Lesson11 project as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson11/src

Package: collections

□ Enclosing type:

Name: Cheetah

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: Mammal

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)

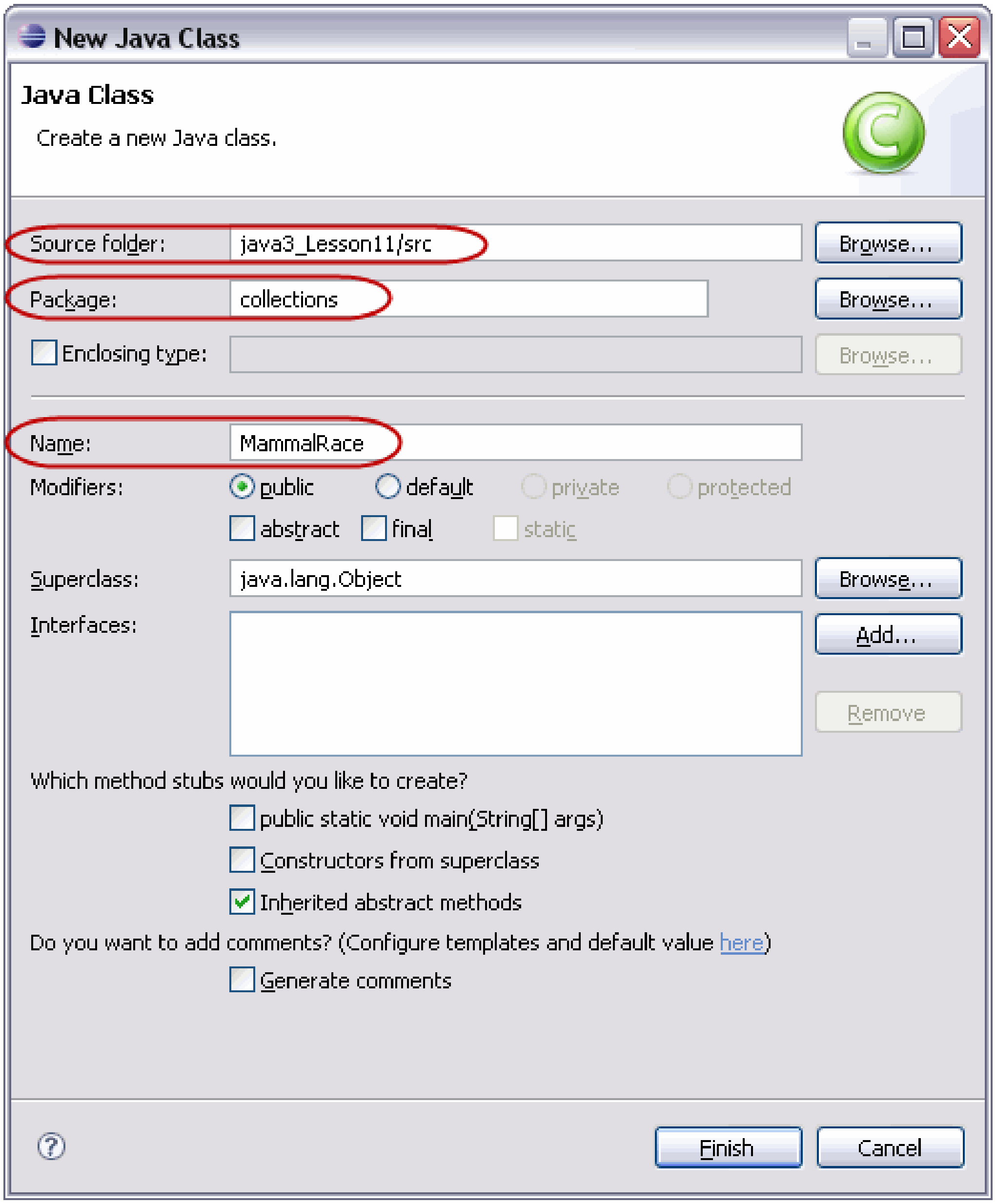
□ Generate comments

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

CODE TO TYPE: Cheetah

```
package collections;

public class Cheetah extends Mammal {
    private double runningSpeed = 70.00;     // in mph
    private double height = 1.25;  // shoulder height in meters

    public Cheetah(String who){
        super(who);
    }

    public double getHeight(){
        return height;
    }

    public double getSpeed(){
       return runningSpeed;
    }
}
```

Now, let's create a class to compare these three mammals. In the java3_Lesson11 project, create a new class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson11/src

Package: collections

□ Enclosing type:

Name: MammalRace

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 **MammalRace** as shown below in **blue**:

CODE TO TYPE: MammalRace

```
package collections;

import java.util.*;

public class MammalRace {

    public static void main (String[] args){
        MammalRace testing = new MammalRace();
     testing.race();
    }

    public void race(){
        Human me = new Human("me(Human)");
        Three_toedSloth frank = new Three_toedSloth("frank sloth");
        Cheetah chester = new Cheetah("chester cheetah");

        List<Mammal> critters = new ArrayList<Mammal>();
        critters.add(me);
        critters.add(frank);
        critters.add(chester);
        System.out.println("Original Objects: ");
        System.out.println(critters);

        ListIterator<? extends Mammal> crittersIter = critters.listIterator();
        System.out.println("Elements of the list by their names: ");
        while (crittersIter.hasNext()){
            System.out.print(crittersIter.next().getName() + ", ");
        }
        System.out.println();

        Collections.sort(critters, new Comparator<Mammal>() {
            public int compare(Mammal a, Mammal b ){
                if (a.getSpeed() < b.getSpeed()) return -1;
                if (a.getSpeed() > b.getSpeed()) return 1;
                return 0;
      }});

        System.out.println("\nSorted from slowest to fastest, "
                   + "with speed information:");

        for (Mammal each : critters){
            System.out.println("Name: " + each.getName() + " Speed: "
                   + each.getSpeed() + " mph");
        }

        Collections.sort(critters, new Comparator<Mammal>() {
            public int compare(Mammal a, Mammal b ){
               if (a.getHeight() < b.getHeight()) return -1;
               if (a.getHeight() > b.getHeight()) return 1;
               return 0;
        }});
        System.out.println("\nSorted from shortest to tallest, "
                   + "with height information:");
        for (Mammal each : critters){
            System.out.println("Name: " + each.getName()
                   + " Height: " + each.getHeight()+ " m");
        }
    }
}
```

Save all of the classes: **Mammal**, **Human**, **Cheetah**, **Three_toedSloth**, and **MammalRace**.

Run **MammalRace**. Compare the results in the console with the code and observe how the methods worked. We have two different ways to sort and compare the elements. By creating an interface **Comparator** within a Generic Framework, the Collections Framework allows us to sort various ways. Good stuff!

Wrapping Up the Collections Framework

Using the Collections Framework can save you a lot of time. It may well be worthwhile for you to look at Java's tutorial and become familiar with all of the capabilities these interfaces and classes provide. Why reinvent the wheel, right?

Here are just some of the benefits provided by the Java Collections Framework, as described in that tutorial we just mentioned:

- **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework allows you to concentrate on the important aspects of your own program, rather than reinventing things that have already been written (by experts). Two major contributions these collections have made in the ongoing quest to simplify the writing of code can be seen in:
    1. Enhanced **for** loops
    2. Autoboxing
- **Increases program speed and quality:** The Collections Framework provides high-performance, high-quality implementations of useful constructs and algorithms. The various implementations of each interface are interchangeable, so programs can be tuned easily by switching collection implementations, depending on your needs.
- **Allows interoperability among unrelated APIs:** The collection interfaces define how to pass collections back and forth. If one individual's code furnishes a collection of node names and another's toolkit expects a collection of certain column headings, the Collection APIs will interoperate seamlessly, even though they were written independently.
- **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections as input and furnish them as output. With a set of standard collection interfaces, collection use is uniform.
- **Reduces effort to design new APIs:** Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
- **Fosters software reuse:** If you use standard types and conform to the framework, others can use your code without major changes. They will love you for this.

All in all, Java provides some really handy tools!

Enumeration and enum

Enumeration

Enumerate: To count off or name one by one; list. This is the definition one finds in the *American Heritage Dictionary*. Java's new reserved word **enum** comes from the use of *enumeration*.

Earlier in this series of courses, we learned that Java has two main **types**—**primitive data types** and **classes**. In our current course, we learned that **interfaces** are a **types** as well. Programming languages evolve all the time. Since version 1.5, Java has provided a new **types** of object, the **Enum**. It's actually pretty cool that you're involved right now, during this evolution of a programming language! If you go into the API from Java Version 1.4 in **java.lang**, you won't find the new **Enum** class.

Click on that link to the API from Java Version 1.4 and scroll down. Look at the **Class Summary** between **Double** and **Float**. There's nothing that starts with an **E**.

API Now, click our link to the newest API version and go to **java.lang**. **Enum<E> extends Enum <E>** .

All the *enum*s that you define, by default, will inherit from **java.lang.Enum**. Let's take a closer look at this new **type**.

Constants

In object-oriented programming, **everything** is in a **class**. There are pieces of information that should be readily accessible to everyone, such as:

PI (π)
E
the speed of light (c in E = mc^2)
Avogadro's constant

In Java, we make these pieces of information **constants** by declaring them with **public static final**:

- **public** makes them accessible to everyone.
- **static** makes them accessible by using the class name (for example, **Math.PI**).
- **final** makes sure that no one can change them.

API Go to the **java.lang** package. Scroll down to the **Math** class. Scroll to the **Field Summary** and click on **E**.

The convention in Java is to name constants with all capital letters; that's why you see **PI** and **E**.

Enum Types

An enumerated type is a special kind of class. An **enum type** is a **type** with **variables** (fields) that consist of a fixed set of constants. Common examples include: days of the week, months of the year, seasons of the year (values of WINTER, SUMMER, FALL, and SPRING), compass directions (NORTH, SOUTH, EAST, and WEST), and static Color or Action choices on a menu. Because they are constants, the names enum types' fields conventionally use uppercase letters as well.

The values of the enumerated type are a fixed set of constants (by default) and are objects. The values are, in fact, instances of their own enumeration type.

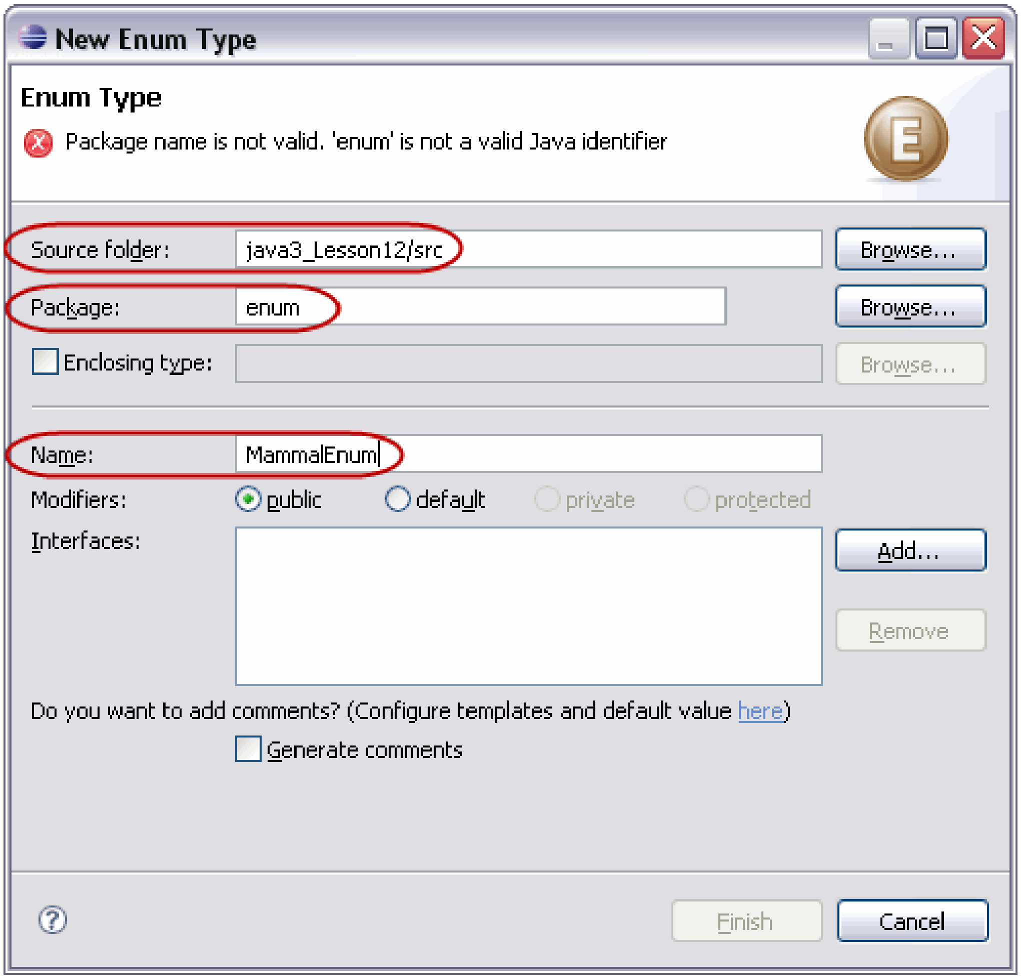
In our upcoming example, **CHEETAH** is an object/instance of the **MammalEnum** class. Specifically, the **enum** declaration defines a **class** (called an *enum type*). The enum class body can include methods and other fields. Interestingly, they have a static **values()** method that returns an array containing all of the values of the enum in the order that they are declared—so you can iterate through the enumeration.

Enum Example

The animal of type **Mammal** and the subclasses we used in the last lesson's example are usually classes of their own. But we'll make them of the class **enum** here, in order to demonstrate the use of **enum**.

Create a new **java3_Lesson12** project. If you're given the option to "Open Associated Perspective," click **No**. Click on java3_Lesson12 and then right-click for the popup menu. Select **New | Enum** (if no Enum option appears in this popup, select **New | Other | Java | Enum**). **Enum** should appear in the popup from then on.

In the **New Enum Type** window that opens, enter the circled information:



Enum Type

Package name is not valid. 'enum' is not a valid Java identifier

Source folder: java3_Lesson12/src

Package: enum

□ Enclosing type:

Name: MammalEnum

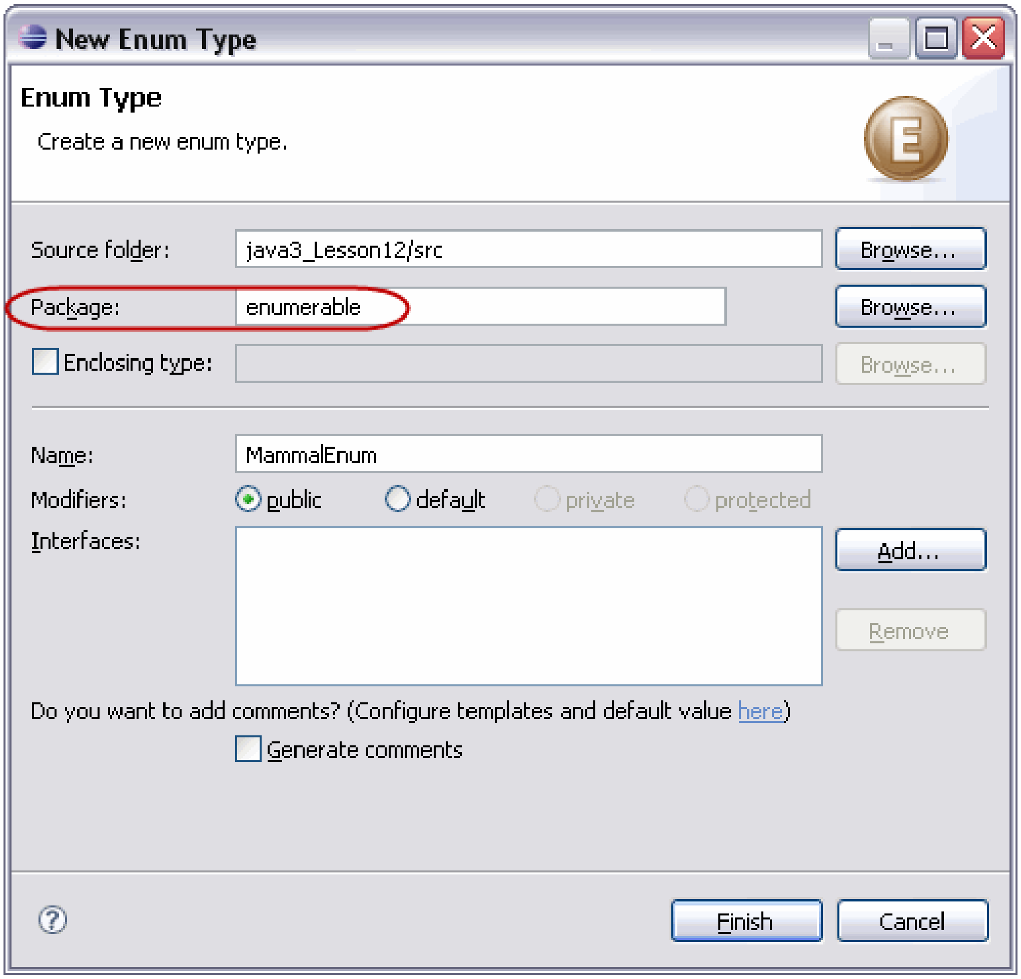
Modifiers: ● public ○ default ○ private ○ protected

Interfaces:

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

□ Generate comments

Hey, what's going on here? Why can't you click Finish? Sorry about that—I wanted to demonstrate that **enum** is a reserved word. Change the name of the package to **enumerable**:



New Enum Type

Enum Type

Create a new enum type.

Source folder: java3_Lesson12/src

Package: enumerable

□ Enclosing type:

Name: MammalEnum

Modifiers: ● public ○ default ○ private ○ protected

Interfaces:

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

□ Generate comments

Create the **MammalEnum** class as shown in **blue**:

CODE TO TYPE: MammalEnum

```
package enumerable;

public enum MammalEnum {
    CHEETAH,
    HUMAN,
    THREETOED_SLOTH;

    public static void main(String[] args){
        for (MammalEnum each : MammalEnum.values())
            System.out.println(each);
    }
}
```

OBSERVE: MammalEnum

```
package enumerable;

public enum MammalEnum {
  CHEETAH,
  HUMAN,
  THREETOED_SLOTH;

  public static void main(String[] args){
    for (MammalEnum each : MammalEnum.values())
      System.out.println(each);
  }
}
```

In the **MammalEnum** class, we create three constant objects: **CHEETAH**, **HUMAN**, and **THREETOED_SLOTH**. In the **main()** method, we output those constants to the console.

Save and **Run** it. It's good, but we can make it better. Remember, each of these are **Objects** of their own.

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

CODE TO TYPE: MammalEnum

```
package enumerable;

public enum MammalEnum {
    CHEETAH (70.00, 1.25),
    HUMAN (27.89, 1.6),
    THREETOED_SLOTH (0.15, 0.58);

    private double speed;
    private double height;

    MammalEnum(double howFast, double howTall){
        speed = howFast;
        height = howTall;
    }

    public double getSpeed(){
        return speed;
    }

    public double getHeight(){
        return height;
    }

    public static void main(String[] args){
        for (MammalEnum each : MammalEnum.values())
            System.out.println("Mammal " + each + ": Speed " + each.getSpeed() + " and
 Height " + each.getHeight());
    }
}
```

OBSERVE: MammalEnum

```
package enumerable;

public enum MammalEnum {
    CHEETAH (70.00, 1.25),
    HUMAN (27.89, 1.6),
    THREETOED_SLOTH (0.15, 0.58);

    private double speed;
    private double height;

    MammalEnum(double howFast, double howTall){
        speed = howFast;
        height = howTall;
    }

    public double getSpeed(){
        return speed;
    }

    public double getHeight(){
        return height;
    }

    public static void main(String[] args){
        for (MammalEnum each : MammalEnum.values())
            System.out.println("Mammal " + each + ": Speed " + each.getSpeed()
                   + " and  Height " + each.getHeight());
    }
}
```

Here, we take the three constant objects and give them data. Each of the three constants, **CHEETAH** (**70.00**, **1.25**), **HUMAN** (**27.89**, **1.6**), and **THREETOED_SLOTH** (**0.15**, **0.58**), now take in parameters that will be passed to the enum constructor. This single **MammalEnum** now represents three separate objects. The **parameters** of these objects, represent the **speed** and **height** of the object.

Each of the **MammalEnum** objects have private variables named **speed** and **height**, as well as **getters** for those values. Did you notice that there are no **setters**? That's because the objects in an **enum** are implicitly **public static final**.

Save and **Run** it. Pretty cool, huh?

Of course, it might be better to use things that would actually be **constants**. Each of the things in our current example is a class that we would want to instantiate with individuals from the class. The classes of **Cheetah** and **Human** are certainly not **final**.

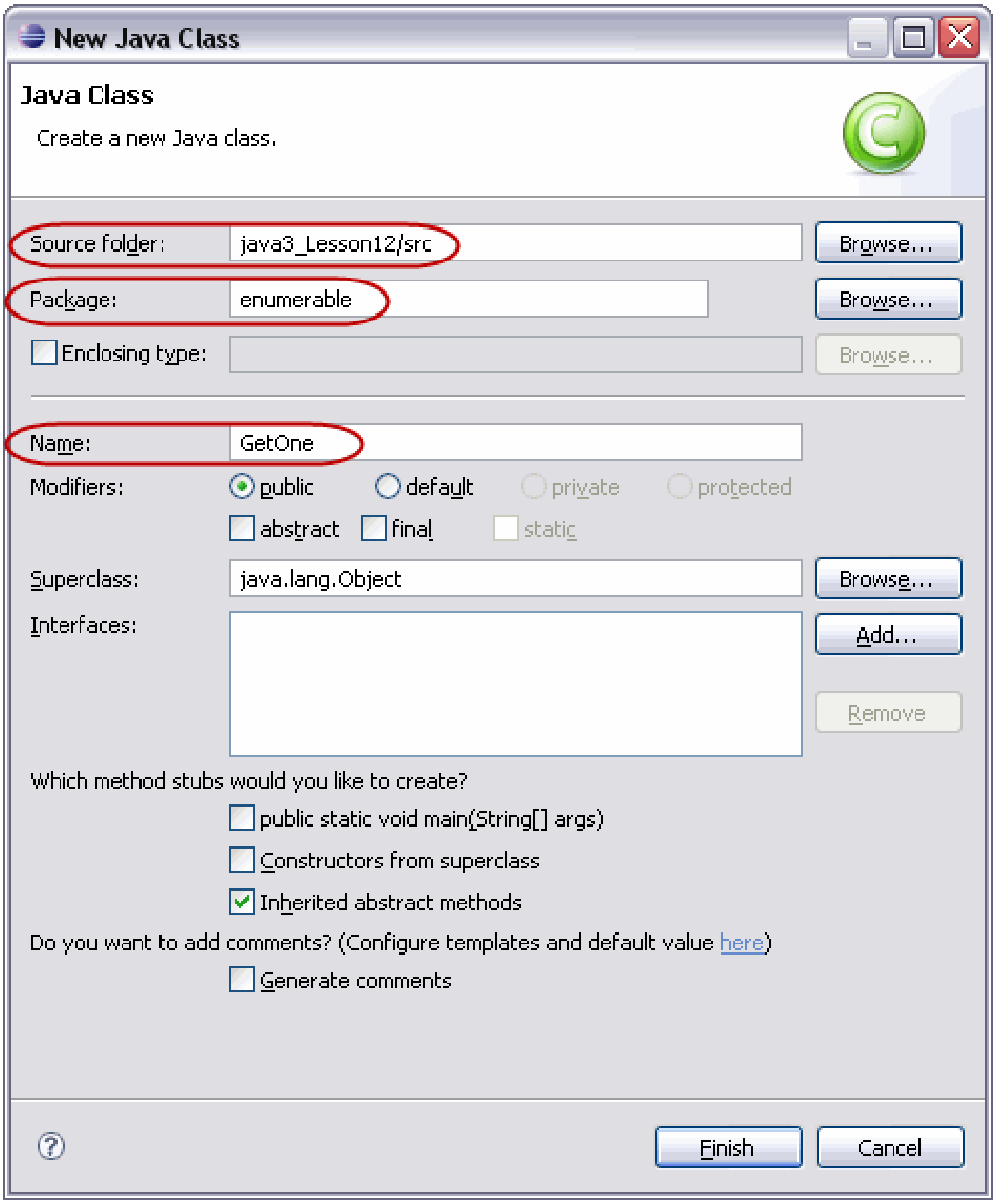
Enumerations should be objects that are constant, like the planets: Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune, and Pluto...well, maybe constants are not always constant either. Poor Pluto!

But jokes aside, if objects **are** constants, then we do not need to instantiate them (there are not multiple instances of Mars). If we have a collection of such objects, the **enum** type is a good option.

Be aware that constructors for **enum** are not the same as constructors for **classes**. The constructor for an enum type must be **package-private** or **private access**. (For most classes, that would be pretty strange.) An **enum** constructor automatically creates the constants that are defined at the beginning of the **enum** body. You cannot invoke an enum constructor yourself.

Accessing Members of the Enumeration

Let's try an example to access one of our mammals from another class. Create a new class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson12/src

Package: enumerable

□ Enclosing type:

Name: GetOne

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

Add the code for **GetOne** as shown in **blue**:

CODE TO TYPE: GetOne

```
package enumerable;

public class GetOne {

    public static void main(String[] args) {
        MammalEnum test;
        test = MammalEnum.CHEETAH;
        System.out.println(test + " height is " + test.getHeight());
    }
}
```

Save and **Run** it. Pretty cool, huh?

Now let's make a change. Add the **blue** code and remove the **red** code:

CODE TO EDIT: GETONE

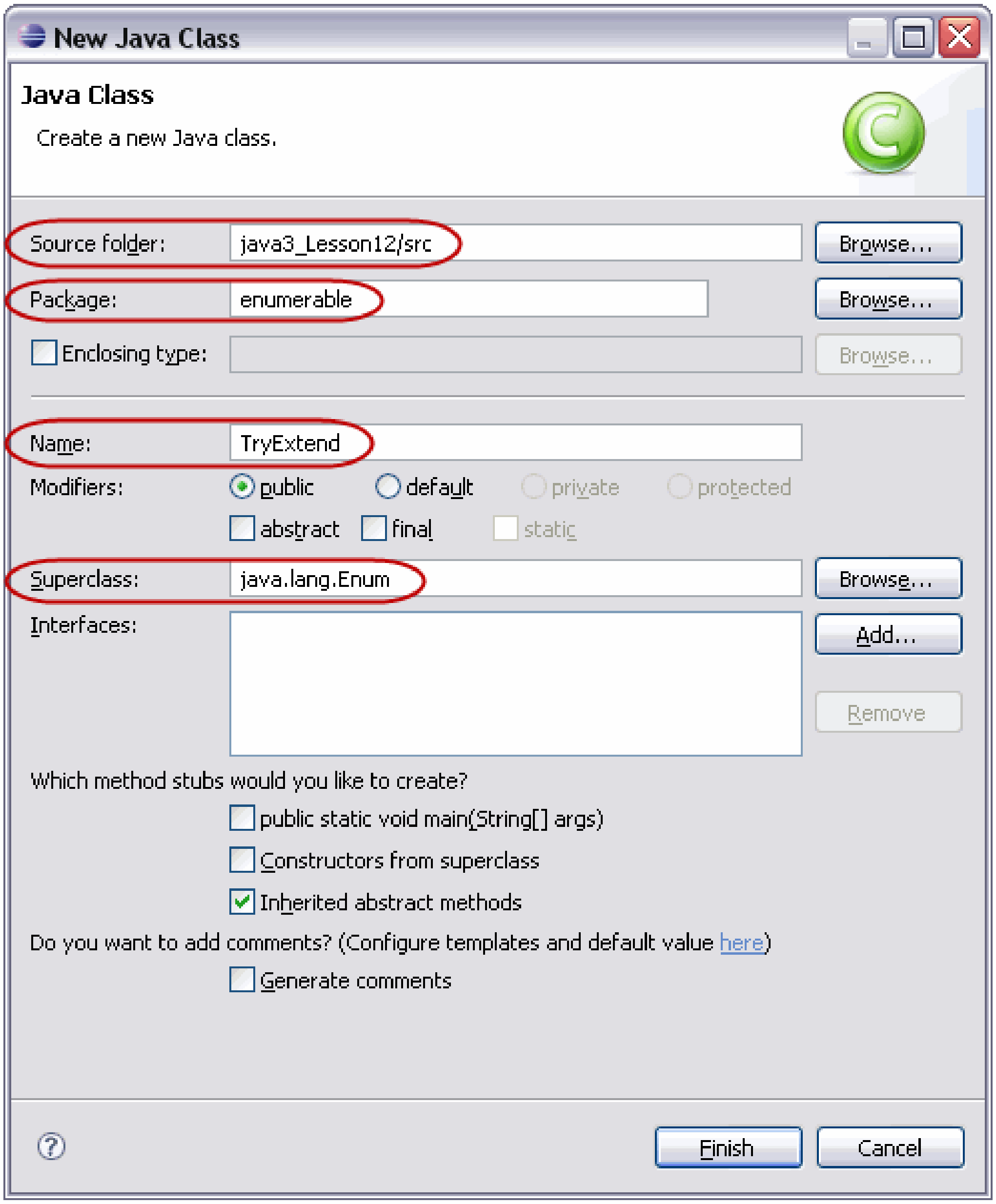
```
package enumerable;

public class GetOne {

    public static void main(String[] args) {
        MammalEnum test;
        test = MammalEnum.CHEETAHCOUGAR;
        System.out.println(test + " height is " + test.getHeight());
    }
}
```

You can't do it because **COUGAR** is not an object in the enumeration. You can change COUGAR back to CHEETAH to return the code to a functional state.

More about Enum

API In the API, go to **java.lang.Enum** (it's in java.lang, so no import is needed). Now, **that** is interesting. I think we get the idea behind *generics*, but what does this **Class Enum<E extends Enum<E>>** mean? By default, any *enum* that is defined will inherit from **java.lang.Enum**. So **Class Enum<E extends Enum<E>>** indicates that any specification of an **enum** will **extend** the class **Enum**. Let's try to extend **Enum** explicitly:

In java3_Lesson12, create a new Class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson12/src

Package: enumerable

□ Enclosing type:

Name: TryExtend

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Enum

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

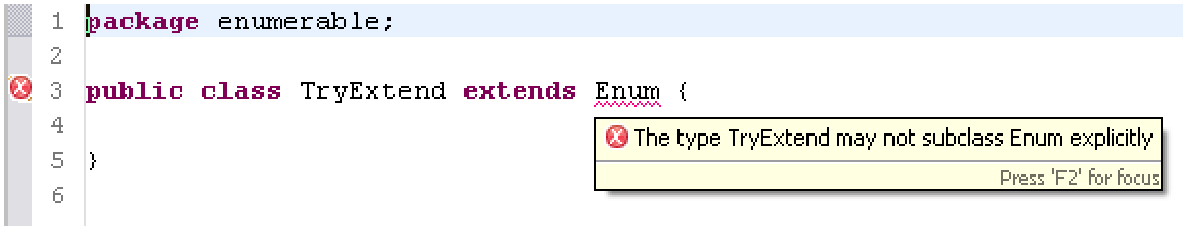
□ Constructors from superclass

☑ Inherited abstract methods

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

□ Generate comments

We have an error:



```
package enumerable;

public class TryExtend extends Enum {

}

```

The type TryExtend may not subclass Enum explicitly

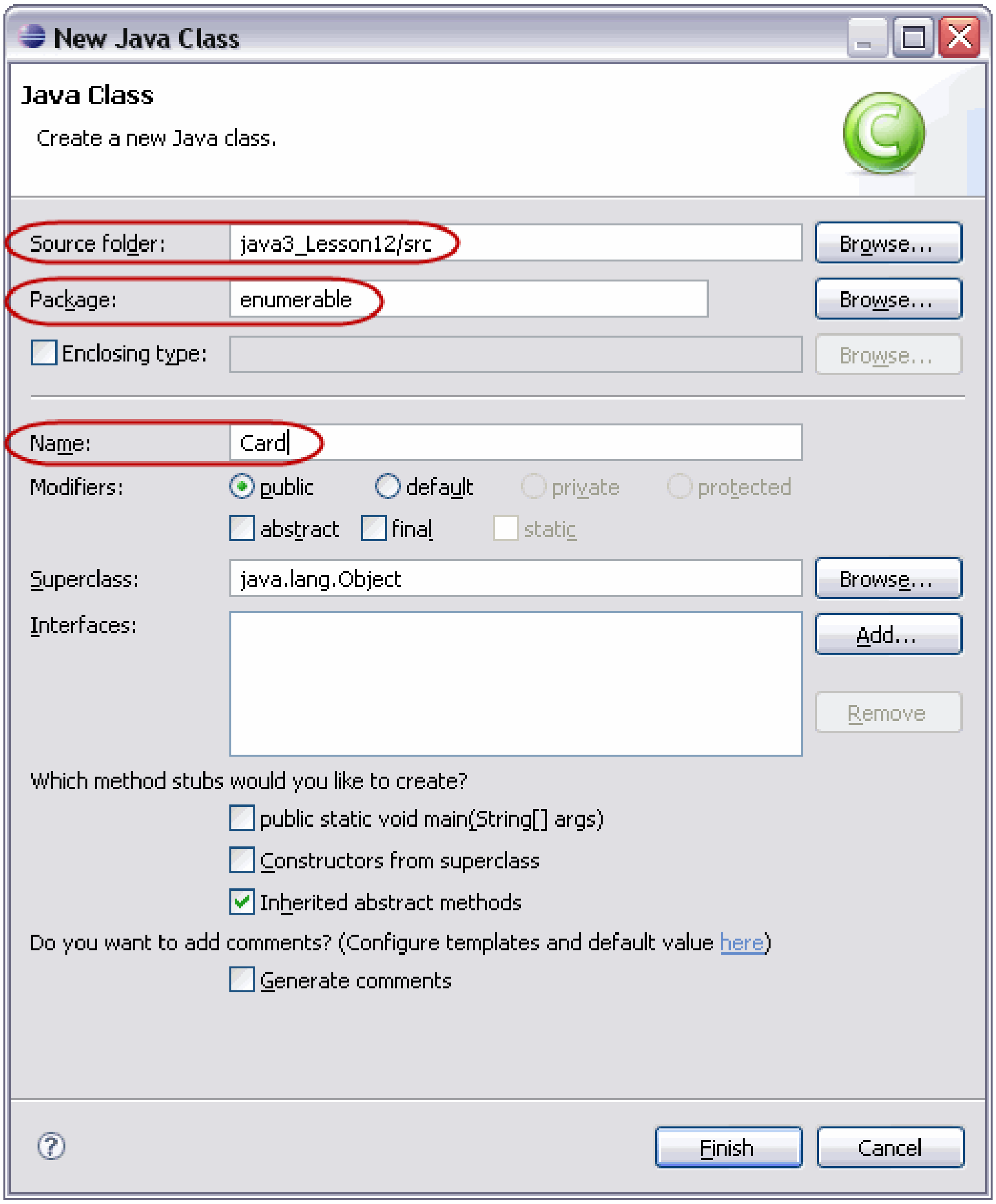
That's interesting too—if you read the class specification in the API, it says **public abstract class Enum<E extends Enum<E>>**. It has an **abstract** modifier. If a class is **abstract**, it **must** be subclassed. However, **java.lang.Enum** is highly specialized, and cannot be explicitly subclassed.

Enum Inside of Classes

Remember that an **enum** is, for all intents and purposes, a set of constants. It is not so specialized though, that its classes become unusable. They **can** be defined inside of your regular classes.

To illustrate, let's play some cards. This code was inspired by the **enum** example in the Java Enum Guide. We'll work with parts of it and add more in the next lesson.

In java3_Lesson12, create a new Class as shown:



Source folder: java3_Lesson12/src

Package: enumerable

□ Enclosing type:

Name: Card

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

Create **Card** as shown below (you'll notice similarities to our **MammalEnum** class) in **blue**:

CODE TO TYPE: Card

```
package enumerable;

import java.util.*;

public class Card {

  public enum Face { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN,
 KING, ACE }

  public enum Suit { HEARTS, DIAMONDS, CLUBS, SPADES }

  private final Face face;
  private final Suit suit;

  private Card(Face face, Suit suit) {
    this.face = face;
    this.suit = suit;
  }

  public Face getFace() {
    return face;
  }

  public Suit getSuit() {
    return suit;
  }

  public String toString() {
    return face + " of " + suit;
  }

  public static void demo(){
    ArrayList<Card> aDeck = new ArrayList<Card>();
      for (Suit suit : Suit.values())
        for (Face face : Face.values())
          aDeck.add(new Card(face, suit));
      for (Card each : aDeck)
        System.out.println(each);
    }

    public static void main(String [] args){
     Card.demo();
    }
}
```

OBSERVE: Card

```
package enumerable;

import java.util.*;

public class Card {

  public enum Face { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN,
 KING, ACE }

  public enum Suit { HEARTS, DIAMONDS, CLUBS, SPADES }

  private final Face face;
  private final Suit suit;

  private Card(Face face, Suit suit) {
    this.face = face;
    this.suit = suit;
  }

  public Face getFace() {
    return face;
  }

  public Suit getSuit() {
    return suit;
  }

  public String toString() {
    return face + " of " + suit;
  }

  public static void demo(){
    ArrayList<Card> aDeck = new ArrayList<Card>();
      for (Suit suit : Suit.values())
        for (Face face : Face.values())
          aDeck.add(new Card(face, suit));
      for (Card each : aDeck)
        System.out.println(each);
    }

    public static void main(String [] args){
      Card.demo();
    }
}
```

In the Card class, we define two enums, **FACE** and **SUIT**. We instantiate **face** and **suit** as instance variables of type **FACE** and **SUIT**, our enums. Each instance of our **Card** class is going to represent one **FACE** object and one **SUIT** object.

In the **demo()** method, we create an **ArrayList** named **aDeck**, which will hold our 52 **Card** objects. We loop through the **SUIT** enum's **values()**, get one of the **SUIT** objects (**HEARTS**, **CLUBS**, etc.), and store it in the local variable **suit**. For each **Suit**, we loop through the **FACE** enum's **values()**, get one of the **FACE** objects (**KING**, **ACE**, etc.) and store it in the local variable **face**. Then we **add()** a new card to **aDeck** with the values of **suit** and **face**. Finally, we loop through **aDeck** and print out **each Card** object, implicitly invoking its **toString()** method.

Save and **Run** it. Notice that we listed out all of the cards one at a time. Since a deck of cards stays the same all of the time, it would be smart to make up the deck and store it as a **static** (class) variable, so it's accessible from the class.

A few things to keep in mind when considering **static** methods:

- A static method can be invoked though the class name, without any objects instantiated.
- Because it is not bound to any instance, it can access only other **static** members.
- It can be called by instances, but accessed **independently from the class.**

Edit Card as shown below by adding the **blue** code and removing the **red** code:

CODE TO EDIT: Card

```
package enumerable;

import java.util.*;

public class Card {

  public enum Face { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN,
 KING, ACE }

  public enum Suit { HEARTS, DIAMONDS, CLUBS, SPADES }

  private final Face face;
  private final Suit suit;
  private static final List<Card> theDeck = new ArrayList<Card>();

  private Card(Face face, Suit suit) {
    this.face = face;
    this.suit = suit;
  }

  private static List<Card> initializeDeck (){
    for (Suit suit : Suit.values())
      for (Face face : Face.values())
        theDeck.add(new Card(face, suit));
    return theDeck;
  }

  public Face getFace() {
    return face;
  }

  public Suit getSuit() {
    return suit;
  }

  public String toString() {
    return face + " of " + suit;
  }

  public static void demo(){
    ArrayList<Card> aDeck = new ArrayList<Card>();
    for (Suit suit : Suit.values())
      for (Face face : Face.values())
        aDeck.add(new Card(face, suit));
    for (Card each : aDeck)
      System.out.println(each);
  }

  public static void main(String [] args){
   Card.demo();
    List<Card> aDeck = Card.initializeDeck();
    System.out.println(aDeck);
  }
}
```

Save and **Run** it.

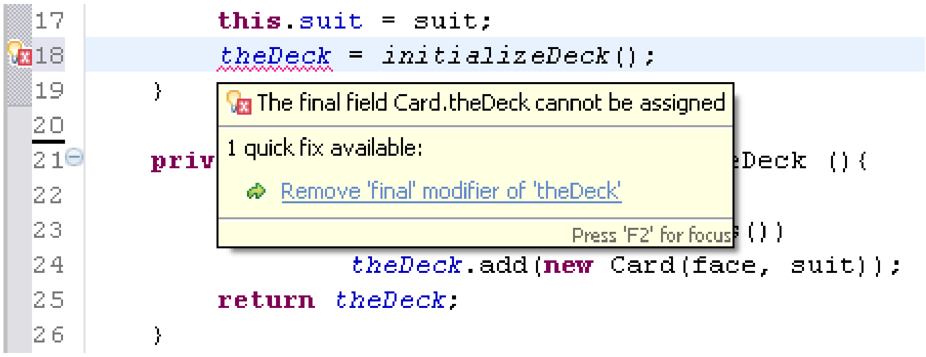
This is good—now we can get rid of the **demo()** method, which doesn't have much to do with a **Card** other than showing it. It would be even better if we didn't have to call the method **initializeDeck()** either—especially since it's **private** and we couldn't call it from outside the class anyway. But wait a minute— we **want** it to be private, because we don't want people messing with our deck and its contents! Maybe it would be best then, if we could always have the deck of cards available from the class, through its class variable. But without the **initializeDeck()** method, the array **theDeck** would be empty. Let's add the method to the Constructor to get **theDeck** filled.

Edit Card as shown in blue:

CODE TO TYPE: Card

```
package enumerable;

import java.util.*;

public class Card {

    public enum Face { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }

    public enum Suit { HEARTS, DIAMONDS, CLUBS, SPADES }

    private final Face face;
    private final Suit suit;
    private static final List<Card> theDeck = new ArrayList<Card>();

    private Card(Face face, Suit suit) {
        this.face = face;
        this.suit = suit;
        theDeck = initializeDeck();
    }

    private static List<Card> initializeDeck (){
        for (Suit suit : Suit.values())
            for (Face face : Face.values())
                theDeck.add(new Card(face, suit));
        return theDeck;
    }

    public Face getFace() {
       return face;
    }

    public Suit getSuit() {
       return suit;
    }

    public String toString() {
       return face + " of " + suit;
    }

    public static void main(String [] args){
      List<Card> aDeck = Card.initializeDeck();
      System.out.println(aDeck);
    }
}
```

We have a new error message:



```
        this.suit = suit;
        theDeck = initializeDeck();
    }

    priv                                      Deck (){
                                              ())
                theDeck.add(new Card(face, suit));
        return theDeck;
    }
```

The final field Card.theDeck cannot be assigned

1 quick fix available:

Remove 'final' modifier of 'theDeck'

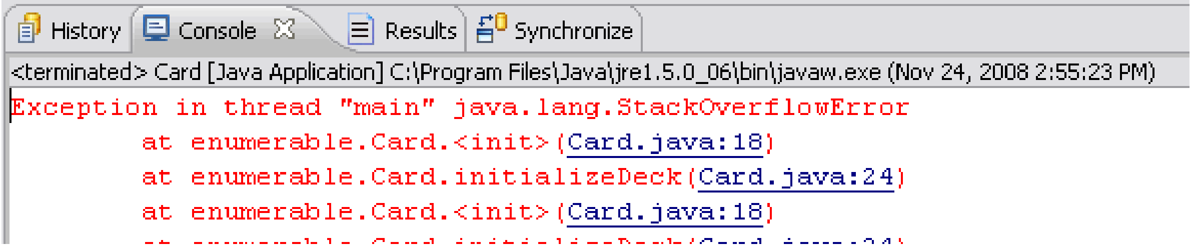
Press 'F2' for focus

Change the **Card** class by removing the **red** code as shown:

CODE TO EDIT: Card

```
package enumerable;

import java.util.*;

public class Card {

    public enum Face { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEE
N, KING, ACE }

    public enum Suit { HEARTS, DIAMONDS, CLUBS, SPADES }

    private final Face face;
    private final Suit suit;
    private static final List<Card> theDeck = new ArrayList<Card>();

    private Card(Face face, Suit suit) {
        this.face = face;
        this.suit = suit;
        theDeck = initializeDeck();
    }

    private static List<Card> initializeDeck (){
        for (Suit suit : Suit.values())
            for (Face face : Face.values())
                theDeck.add(new Card(face, suit));
        return theDeck;
    }

    public Face getFace() {
        return face;
    }

    public Suit getSuit() {
        return suit;
    }

    public String toString() {
        return face + " of " + suit;
    }

    public static void main(String [] args){
        List<Card> aDeck = Card.initializeDeck();
        System.out.println(aDeck);
    }
}
```

Save and **Run** it. We still have a problem:



```
<terminated> Card [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Nov 24, 2008 2:55:23 PM)
Exception in thread "main" java.lang.StackOverflowError
        at enumerable.Card.<init>(Card.java:18)
        at enumerable.Card.initializeDeck(Card.java:24)
        at enumerable.Card.<init>(Card.java:18)
        at enumerable.Card.initializeDeck(Card.java:24)
```

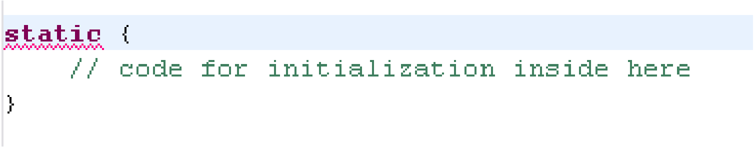
Do you see why? We are trying to make a **Deck** of **Card** in the **Card** constructor. We need to have the **Card** before we can make a **Deck**.

One solution for this would be to make a **static initialization block**. Read on.

Static Initialization Blocks

A static initialization block is a normal block of code enclosed in Curly brackets **{}** and preceded by the **static** keyword. It's not a method, it's an **initializer**. Its basic purpose is to perform initialization of static variables that can't be accomplished in a variable declaration.

It is not "called" when an object of the class is instantiated; it is executed the first time the class itself is referenced, similar to static variable declarations. Here's an example:



```
static {
    // code for initialization inside here
}
```

Key characteristics of static initialization blocks:

- A class can have any number of static initialization blocks.
- They can appear anywhere in the class body.
- The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code. This is important because variables in one might rely on the other having been instantiated.

Note *Initializing the class* is not the same as *instantiating an object.* Initializing the class happens only once per class per classloader. So static initializers are run once per class, when the class is loaded, which occurs the first time your code references it.

Edit Card as shown. Add the code in **blue** and remove the code in **red**:

CODE TO EDIT: Card

```
package enumerable;

import java.util.*;

public class Card {

    public enum Face { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEE
N, KING, ACE }

    public enum Suit { HEARTS, DIAMONDS, CLUBS, SPADES }

    private final Face face;
    private final Suit suit;
    private static final List<Card> theDeck = new ArrayList<Card>();

    private Card(Face face, Suit suit) {
        this.face = face;
        this.suit = suit;
        initializeDeck();
    }

    private static List<Card> initializeDeck (){
        for (Suit suit : Suit.values())
            for (Face face : Face.values())
                theDeck.add(new Card(face, suit));
        return theDeck;
    }

    public Face getFace() {
       return face;
    }

    public Suit getSuit() {
       return suit;
    }

    public String toString() {
       return face + " of " + suit;
    }

    public static void main(String [] args){
        List<Card> aDeck = Card.initializeDeck();
        System.out.println(aDecktheDeck);
    }
}
```

OBSERVE: Card

```
package enumerable;

import java.util.*;

public class Card {

    public enum Face { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEE
N, KING, ACE }

    public enum Suit { HEARTS, DIAMONDS, CLUBS, SPADES }

    private final Face face;
    private final Suit suit;
    private static final List<Card> theDeck = new ArrayList<Card>();

    private Card(Face face, Suit suit) {
        this.face = face;
        this.suit = suit;
    }

    static {
        for (Suit suit : Suit.values())
            for (Face face : Face.values())
                theDeck.add(new Card(face, suit));
    }

    public Face getFace() {
       return face;
    }

    public Suit getSuit() {
       return suit;
    }

    public String toString() {
       return face + " of " + suit;
    }

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

We have three separate and distinct variables for **face**, **face**, and **face**, as well as for **suit**, **suit**, and **suit**. **face** and **suit** are instance variables of the class **Card**. **face** and **suit** are parameters of the **Card** constructor. And, **face** and **suit** are local variables to the **for loop** in which they were created.

Save and **Run** it. Now let's provide a class method that will allow people access to the **Deck** from the **Card** class.

Edit the **Card** class as shown below in **blue**:

CODE TO TYPE:

```
package enumerable;

import java.util.*;

public class Card {

    public enum Face { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEE
N, KING, ACE }

    public enum Suit { HEARTS, DIAMONDS, CLUBS, SPADES }

    private final Face face;
    private final Suit suit;
    private static final List<Card> theDeck = new ArrayList<Card>();

    private Card(Face face, Suit suit) {
        this.face = face;
        this.suit = suit;
    }

    static {
        for (Suit suit : Suit.values())
            for (Face face : Face.values())
                theDeck.add(new Card(face, suit));
    }

    public Face getFace() {
        return face;
    }

    public Suit getSuit() {
        return suit;
    }

    public String toString() {
        return face + " of " + suit;
    }

    public static ArrayList<Card> newDeck() {
        return new ArrayList<Card>(theDeck);
    }

    public static void main(String [] args){
        List <Card> mine = Card.newDeck();
        System.out.println(mine);
    }

}
```

Save and **Run** it. Awesome!

Now, usually **switch** statements need **byte**, **short**, **char**, or **int** primitive data types—but they also work with enumerated types. Let's check that out. **Edit Card** as shown in **blue**:

CODE TO TYPE:

```
package enumerable;

import java.util.*;
import java.awt.Color;

public class Card {

    public enum Face { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }

    public enum Suit { HEARTS, DIAMONDS, CLUBS, SPADES }

    private final Face face;
    private final Suit suit;
    private static final List<Card> theDeck = new ArrayList<Card>();  // declares theDeck

    private Card(Face face, Suit suit) {
        this.face = face;
        this.suit = suit;
    }

    static {
    for (Suit suit : Suit.values())
         for (Face face : Face.values())
         theDeck.add(new Card(face, suit));
    }

    public Face getFace() {
       return face;
    }

    public Suit getSuit() {
       return suit;
    }

    public String toString() {
       return face + " of " + suit;
    }

   public static ArrayList<Card> newDeck() {
        return new ArrayList<Card>(theDeck);
   }

    public Color testSwitch(){
        Color result = null;
        switch(suit){
            case SPADES:
            case CLUBS: result = Color.black;  break;
            case HEARTS:
            case DIAMONDS: result = Color.red;  break;
        }
        return result;
    }

    public static void main(String [] args){
        List<Card> deck  = Card.newDeck();
        Card myCard = deck.get(20);
        if (myCard.testSwitch() == Color.black)
            System.out.println(myCard + " is black");
        else System.out.println(myCard + " is red");
    }
}
```

Save and **Run** it.

You may wonder why we had the **testSwitch()** method return a **Color**, rather than a **String**, which would have enabled us to say "NINE of DIAMONDS is red" and thereby avoid using the **if** statement. You'll understand the reason behind that choice when you get busy with the project for this lesson.

A Bit More About Enum

Enumerations can be declared as their own class (**enum**), but they can also be declared within classes and interfaces as well (as we saw with the Cards). In this way, they behave like inner classes.

The enum object type is limited to the explicit set of values. That means that you can't call the constructor to create new elements for the enum. The values have an established order, defined by the order in which they are declared in the code. The values correspond to a string name, which is the same as their declared name in the source code.

We'll apply more of this information in the next lesson. We'll refine our skills and the capabilities of **Card**s, associating our **Card** class with specific card icons. See you there!

Image Mapping and Handling

Tying It All Together

In the Java first course, we provided a file to download that created your first project. We'll do that again for this lesson to create your final project, **java3_Lesson13**. You'll find some images in the new file that you'll need to complete the lesson.

Click here to get java3_Lesson13 and the image files. It contains an **src** directory with a **games** package and a **games.images** folder to use for your work with cards. It should be listed with your other projects in the Package Explorer view.

Open the **src** folder to see the **games.images** folder and make sure the images are there. Take a look at the **Cards** class in the **src** folder and **games** package. **Open** it in the Editor and **Run** it. It's the same package we had in the last lesson. We're going to create additional classes and then extend their functionality.

Our main objectives for this lesson are to:

- Give the **Cards** added functionality and view.
- Use **Cards** examples to add images, and then identify and move shapes and images within our graphics project.

To reach those goals, we'll work on some examples and eventually complete our graphics drawing project. So far, we have:

- determined a class hierarchy for inheritance.
- created the **abstract** class **Shape**.
- created a panel for user choices.
- used interfaces as listeners:
  - using adapter classes.
  - using anonymous inner classes.
- incorporated a Collection Framework class to hold different **Shape**s.
- used **enum** to specify **Color** choices.

To complete our graphics drawing project, our plan is to:

- load **Image**s.
- determine which **Shape** has been selected.
- use a mouse listener to move the **Shape**s.
- put all of our pieces together.

We'll use the **Cards** class from the previous lesson and add images to demonstrate those last few tasks.

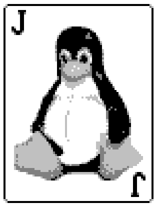
Getting Images

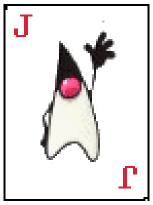
In this lesson we will use the initial **Card** class with the embedded **enum**s from the example in Oracle's Java Enum Guide. We'll add actual images in order to see the cards. It's most convenient that someone has already created the card images and put them on the web with permissions for us to use them!

We found some code we can use for our task in the section on moving images here (thanks, Fred Swartz). To allow you to follow the *use* terms, we included the open source initiative notice for the MIT license. The images are GPL (GNU General Public License).

We'll gradually add more functionality to our example, by adding utility classes. These utility classes will convert our **Cards** (which are currently **String**s) to **Image**s. By working through examples, you'll learn how to display images, identify which has been selected, then move it around and complete your graphics project.

Tip Your program will allow people to draw geometric figures, so I suggest you keep your images small (maximum widths and heights around 150 pixels) to save space. Below are two jokers that were not included in the deck that you downloaded (the dimensions of each of these images are about 75x100).





Here is the **Cards** class that was downloaded with the images:

OBSERVE: Cards

```
package games;

import java.util.*;

public class Cards {

    public enum Face { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }

    public enum Suit { HEARTS, DIAMONDS, CLUBS, SPADES }

    private final Face face;
    private final Suit suit;
    private static final List<Cards> theDeck = new ArrayList<Cards>();

    private Cards(Face face, Suit suit) {
        this.face = face;
        this.suit = suit;
    }

    static {
    for (Suit suit : Suit.values())
         for (Face face : Face.values())
         theDeck.add(new Cards(face, suit));
    }

    public Face getFace() {
       return face;
    }

    public Suit getSuit() {
       return suit;
    }

    public String toString() {
       return face + " of " + suit;
    }

    public static ArrayList<Cards> newDeck() {
        return new ArrayList<Cards>(theDeck);
    }

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

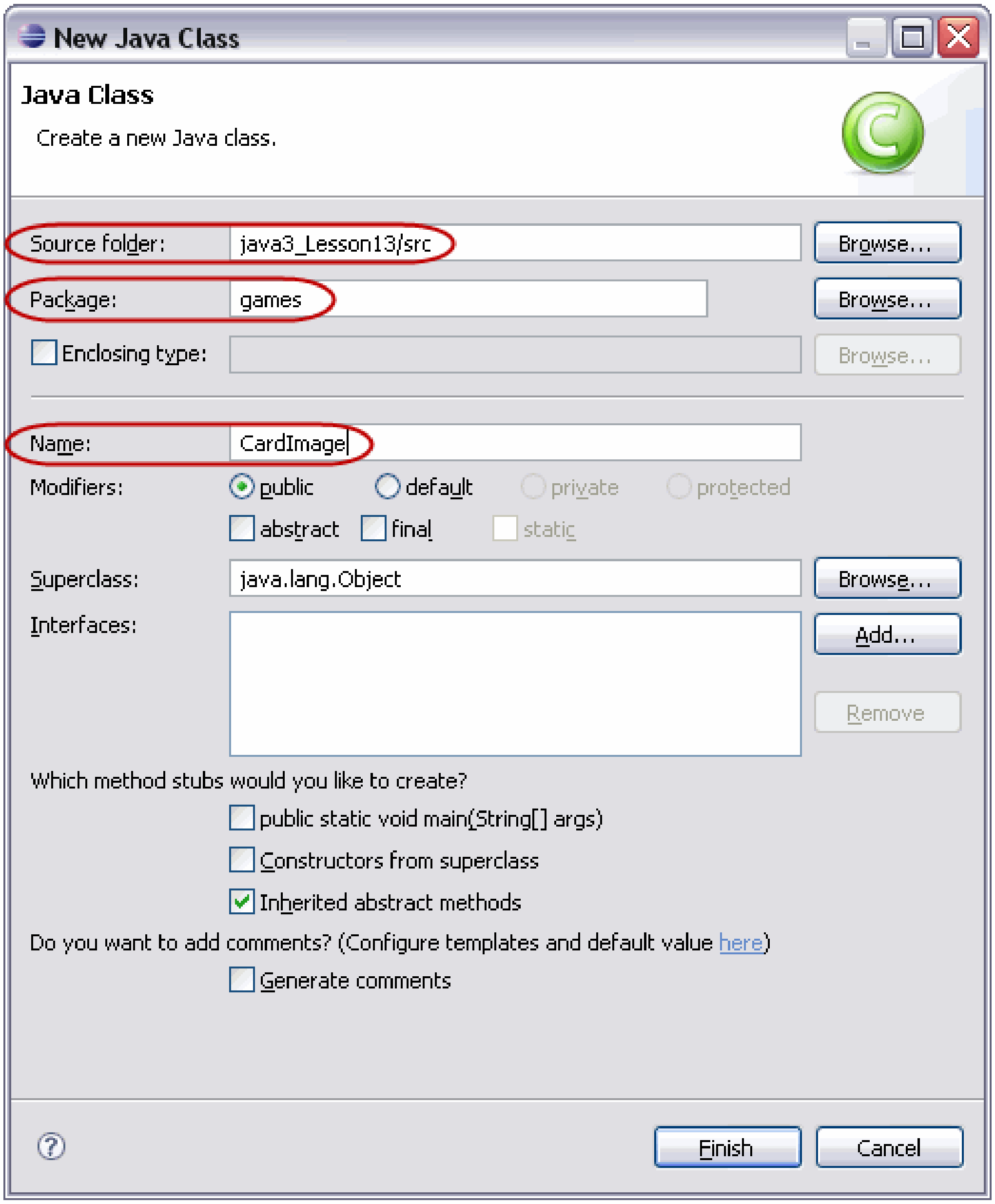
Mapping with a Hashtable

We want to have "real" cards, so we need images. You've actually already downloaded them, but we need to create a **mapping** to match the **Cards** strings above with their corresponding images. For this example, we're going to use a **Hashtable**, which is part of the Collections Framework.

API Go to the **java.util** package. Scroll down to the **Hashtable** class; read and digest the information you find there.

Our **keys** will be the individual **Cards** and their **values** will be the associated card images.

In the java3_Lesson13 project, create a new class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson13/src

Package: games

□ Enclosing type:

Name: CardImage

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

Create **CardImage** by typing the **blue** code as shown:

CODE TO TYPE: CardImage

```
package games;

import javax.swing.*;
import java.util.*;
import java.net.URL;

public class CardImage {
    private Hashtable<Cards, ImageIcon> cardIcons = new Hashtable<Cards, ImageIcon>(52)
;
    private ClassLoader cldr;

    public CardImage(){
        cldr = this.getClass().getClassLoader();
        cardIcons = makeTable(Cards.newDeck());
    }

    private Hashtable <Cards, ImageIcon> makeTable(List<Cards> theDeck){
        for (Cards each : theDeck)
        {
            String mySuit = suitMap(each.getSuit());
            String myFace= faceMap(each.getFace());
            String imagePath = "games/images/" + myFace + mySuit + ".gif";
            URL imageURL = cldr.getResource(imagePath);
            ImageIcon img = new ImageIcon(imageURL);
            cardIcons.put(each, img);
        }
        return cardIcons;
    }

    private String suitMap(Cards.Suit cardSuit){
        return cardSuit.toString().toLowerCase().substring(0,1);
    }

    private String faceMap(Cards.Face cardFace ){
        String result = null;
        switch(cardFace)
        {
            case TWO: result = "2"; break;
            case THREE: result = "3"; break;
            case FOUR: result = "4"; break;
            case FIVE: result = "5"; break;
            case SIX: result = "6"; break;
            case SEVEN: result = "7"; break;
            case EIGHT: result = "8"; break;
            case NINE: result = "9"; break;
            case TEN: result = "t"; break;
            case JACK: result = "j"; break;
            case QUEEN: result = "q"; break;
            case KING: result = "k"; break;
            case ACE: result = "a"; break;
        }
     return result;
    }

    public Hashtable<Cards, ImageIcon> getTable(){
        return cardIcons;
    }

    public static void main(String[] args){
        CardImage testMe = new CardImage();
        List<Cards> myDeck = Cards.newDeck();
        for(Cards each : myDeck)
        {
            System.out.print(each + ": ");
            System.out.println(testMe.cardIcons.get(each));
        }
        }
}
```

Whew! That was a lot to type. Let's break it down. (By the way, you aren't fooling anyone, we know you're using copy and paste! Try to resist.)

OBSERVE: CardImage

```
package games;

import javax.swing.*;
import java.util.*;
import java.net.URL;

public class CardImage {
    private Hashtable<Cards, ImageIcon> cardIcons = new Hashtable<Cards, ImageIcon>(52)
;
    private ClassLoader cldr;

    public CardImage(){
        cldr = this.getClass().getClassLoader();
        cardIcons = makeTable(Cards.newDeck());
    }

    private Hashtable <Cards, ImageIcon> makeTable(List<Cards> theDeck){
        for (Cards each : theDeck)
        {
            String mySuit = suitMap(each.getSuit());
            String myFace= faceMap(each.getFace());
            String imagePath = "games/images/" + myFace + mySuit + ".gif";
            URL imageURL = cldr.getResource(imagePath);
            ImageIcon img = new ImageIcon(imageURL);
            cardIcons.put(each, img);
        }
        return cardIcons;
    }

    private String suitMap(Cards.Suit cardSuit){
        return cardSuit.toString().toLowerCase().substring(0,1);
    }

    private String faceMap(Cards.Face cardFace ){
        String result = null;
        switch(cardFace)
        {
            case TWO: result = "2"; break;
            case THREE: result = "3"; break;
            case FOUR: result = "4"; break;
            case FIVE: result = "5"; break;
            case SIX: result = "6"; break;
            case SEVEN: result = "7"; break;
            case EIGHT: result = "8"; break;
            case NINE: result = "9"; break;
            case TEN: result = "t"; break;
            case JACK: result = "j"; break;
            case QUEEN: result = "q"; break;
            case KING: result = "k"; break;
            case ACE: result = "a"; break;
        }
      return result;
    }

    public Hashtable <Cards, ImageIcon> getTable(){
        return cardIcons;
    }

    public static void main(String[] args){
        CardImage testMe = new CardImage();
        List<Cards> myDeck = Cards.newDeck();
        for(Cards each : myDeck)
        {
            System.out.print(each + ": ");
            System.out.println(testMe.cardIcons.get(each));
        }
        }
}
```

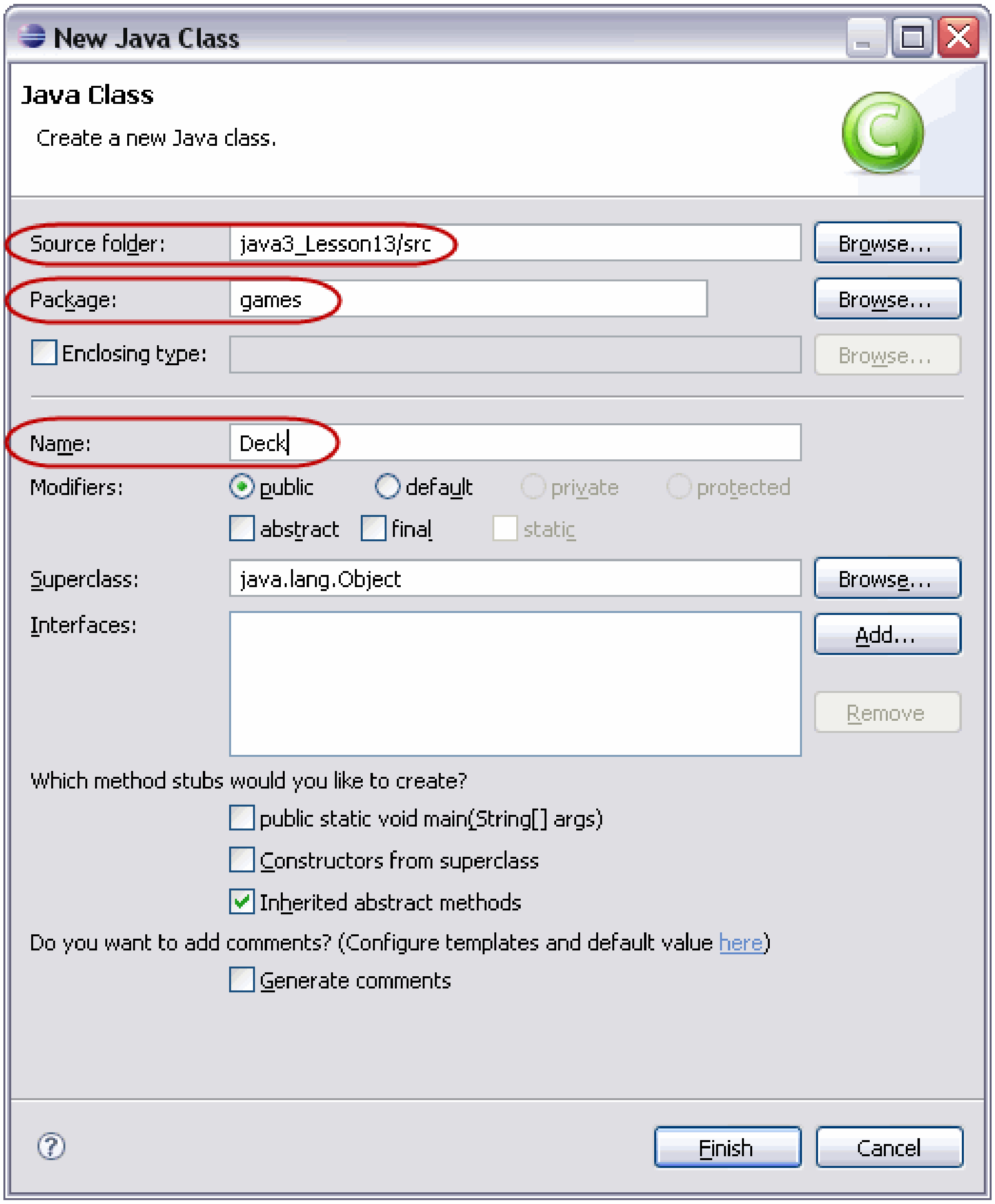
This class is represented by two instance variables. The first is **cardIcons**, a **Hashtable** that maps **Cards** to **ImageIcons**. The second is **cldr**, a **Classloader** that makes it easier to load the images from the disk.

Most of the work for this **CardImage** class is done in the **makeTable()** method, which takes a **List** of **Cards** as a parameter. We loop through the **Cards** objects in the **List**, using the local variable **each**. For **each Cards**, we get the first character of its suit (**mySuit**) by calling **suitMap()**, and we get a one character representation of its face (**myFace**) by calling **faceMap()**. We use **mySuit** and **myFace** to build a path to an image file which we load into an **ImageIcon** named **img**. Then we add **each Cards** object and its associated **img ImageIcon**, into the **cardsIcons Hashtable**.

Save it. We've included a **main** method so you can **Run** it. Check out the mapping in the console output. Each card has its proper image file.

We **could** have used *generics* to create a deck of **Cards** or **ImageIcon** cards, but instead we made sure that the **Cards** deck always has a corresponding matching **ImageIcon** "deck." Now we want to create a class for **Deck** that allows us to do the stuff we like to do with **Decks**, like shuffle and deal hands.

In java3_Lesson13, create a new class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson13/src

Package: games

□ Enclosing type:

Name: Deck

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

Create **Deck** by typing the **blue** code as shown:

CODE TO TYPE: Deck

```
package games;

import java.util.*;
import javax.swing.*;

public class Deck {

    private List<Cards> thisDeck;
    private List<ImageIcon> visualDeck;
    private Cards [][] dealtHands;
    private ImageIcon [][] visualHands;
    private CardImage makeImages;
    private Hashtable <Cards, ImageIcon> myMap;

    public Deck() {
        thisDeck = Cards.newDeck();
        visualDeck = new ArrayList<ImageIcon>();
        makeImages = new CardImage();
        myMap = makeImages.getTable();
        for (Cards each: thisDeck)
        {
            visualDeck.add(myMap.get(each));
        }
    }

    public  List<Cards> getDeck(){
        return thisDeck;
    }

    public  List<ImageIcon> getVisualDeck(){
        return visualDeck;
    }

    public Cards [] getHand(int player){
        return dealtHands[player];
    }

    public ImageIcon [] getVisualHand(int player){
        return visualHands[player];
    }

    public void shuffle(){
        Collections.shuffle(thisDeck);
        visualDeck.clear();
        myMap = makeImages.getTable();
        for (Cards each: thisDeck)
        {
            visualDeck.add(myMap.get(each));
        }
    }

    public void dealAllPlayers(int howManyPlayers, int cardsToDeal){
        dealtHands = new Cards[howManyPlayers][cardsToDeal];
        visualHands = new ImageIcon[howManyPlayers][cardsToDeal];
        this.shuffle();

        System.out.println("We have " + howManyPlayers + " fine Players tonight.");
        for (int i=0; i < howManyPlayers; i++)
        {
            System.out.println("Player " +  (i+1)+ " is dealt an interesting hand of");
            List<Cards> thisHand = dealHand(cardsToDeal);
            for (int j=0 ; j < cardsToDeal; j++)
            {
                dealtHands[i][j] = thisHand.get(j);
                visualHands[i][j] = myMap.get(thisHand.get(j));
            }
            for (Cards each :  thisHand)
            {
                System.out.println(each);

            }
        }
    }

    public  List<Cards> dealHand(int numCards) {
        int deckSize = thisDeck.size();
        List<Cards> aHand = thisDeck.subList(deckSize-numCards, deckSize);
        List<ImageIcon> visualHand = visualDeck.subList(deckSize-numCards, deckSize);
        List<Cards> hand = new ArrayList<Cards>(aHand);

        aHand.clear();
        visualHand.clear();
        return hand;
    }

    public static void main(String[] args){
        Deck myDeck = new Deck();
        int numPlayers = 2;
        int numCards = 5;
        myDeck.dealAllPlayers(numPlayers, numCards);
    }
}
```

OBSERVE: Deck

```
package games;

import java.util.*;
import javax.swing.*;

public class Deck {

    private List<Cards> thisDeck;
    private List<ImageIcon> visualDeck;
    private Cards [][] dealtHands;
    private ImageIcon [][] visualHands;
    private CardImage makeImages;
    private Hashtable <Cards, ImageIcon> myMap;

    public Deck() {
        thisDeck = Cards.newDeck();
        visualDeck = new ArrayList<ImageIcon>();
        makeImages = new CardImage();
        myMap = makeImages.getTable();
        for (Cards each: thisDeck)
        {
            visualDeck.add(myMap.get(each));
        }
    }

    public  List<Cards> getDeck(){
        return thisDeck;
    }

    public  List<ImageIcon> getVisualDeck(){
        return visualDeck;
    }

    public Cards [] getHand(int player){
        return dealtHands[player];
    }

    public ImageIcon [] getVisualHand(int player){
        return visualHands[player];
    }

    public void shuffle(){
        Collections.shuffle(thisDeck);
        visualDeck.clear();
        myMap = makeImages.getTable();
        for (Cards each: thisDeck)
        {
            visualDeck.add(myMap.get(each));
        }
    }

    public void dealAllPlayers(int howManyPlayers, int cardsToDeal){
        dealtHands = new Cards[howManyPlayers][cardsToDeal];
        visualHands = new ImageIcon[howManyPlayers][cardsToDeal];
        this.shuffle();

        System.out.println("We have " + howManyPlayers + " fine Players tonight.");
        for (int i=0; i < howManyPlayers; i++)
        {
            System.out.println("Player " +  (i+1)+ " is dealt an interesting hand of");
            List<Cards> thisHand = dealHand(cardsToDeal);
            for (int j=0 ; j < cardsToDeal; j++)
            {
                dealtHands[i][j] = thisHand.get(j);
                visualHands[i][j] = myMap.get(thisHand.get(j));
            }
            for (Cards each :  thisHand)
            {
                System.out.println(each);

            }
        }
    }

    public  List<Cards> dealHand(int numCards) {
        int deckSize = thisDeck.size();
        List<Cards> aHand = thisDeck.subList(deckSize-numCards, deckSize);
        List<ImageIcon> visualHand = visualDeck.subList(deckSize-numCards, deckSize);
        List<Cards> hand = new ArrayList<Cards>(aHand);

        aHand.clear();
        visualHand.clear();
        return hand;
    }

    public static void main(String[] args){
        Deck myDeck = new Deck();
        int numPlayers = 2;
        int numCards = 5;
        myDeck.dealAllPlayers(numPlayers, numCards);
    }
}
```

This is a long class. Let's get right to work, breaking it down. First, we create a few instance variables that will represent a **Deck** of **Cards**. The instance variable **thisDeck** represents the **newDeck()** of **Cards** objects. The **visualDeck** variable will represent the **ImageIcons** associated with that deck. The **dealtHands**[ ][ ] array will hold the currently dealt hands of **Cards**. Each row will be a player. The **visualHands**[ ][ ] array will hold the **ImageIcons** of the currently dealt hands, where the rows will be comprised of each player's card images. The **makeImages** variable will be a reference to a **CardImage** class, which we will use to get the images of the cards. And finally, the **myMap** variable will be a **Hashtable**, which will give us the ability to map a **Cards** object to an **ImageIcon**.

Okay, now let's take a look at the constructor. We create a **newDeck()** from the **Cards** class and store it in **thisDeck**. We create a new **ArrayList**<**ImageIcon**> object to store the **ImageIcons** we will get from **thisDeck**. The **makeImages** variable is set to a new **CardImage** object and **myMap** is created using the **makeImages.getTable()** method. Then we loop through **each** **Cards** object in **thisDeck** and store its associated image into the **visualDeck**.

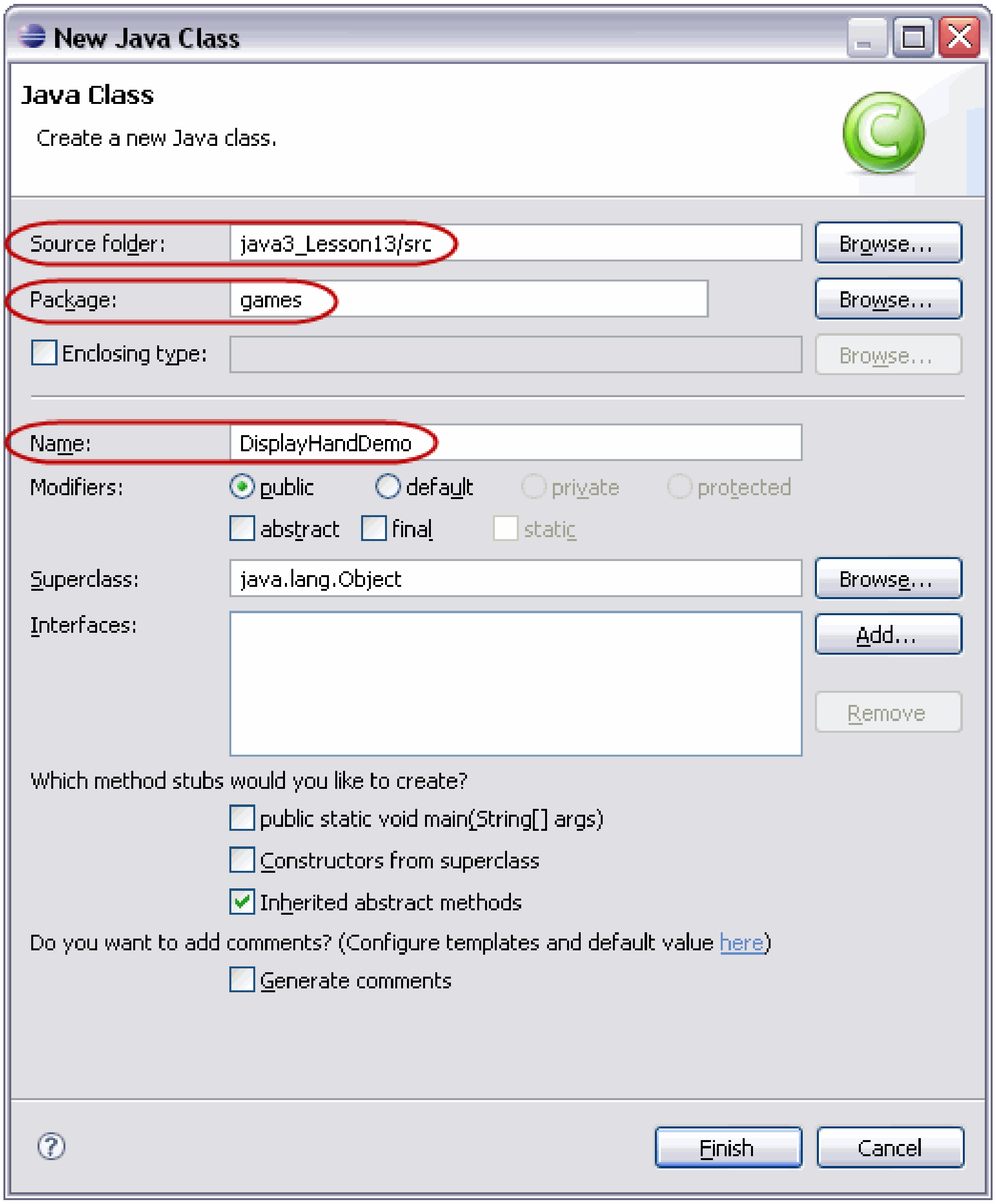
Next, we used the **Collections** class's **shuffle()** method to randomize the order of the **Cards** in **thisDeck**. We **clear()** () the **visualDeck** and then recreate it using the newly shuffled **thisDeck**.

The **dealAllPlayers()** method uses the **dealHand()** method, so let's go over the **dealHand()** method first. We are creating two local **List**<**Cards**> variables, **ahand** and **visualHand**. These will contain **subList()**s of the instance variables, **thisDeck** and **visualDeck**. These are **views** of only portions of **thisDeck** and **visualDeck**. If you **clear()** them, you are in fact clearing that **subList()** in **thisDeck** and **visualDeck** as well. We can remove the hand dealt from **thisDeck** and **visualDeck** by setting our local **hand** variable equal to new ArrayList with **aHand**'s contents and then clearing **aHand** and **visualHand**, which removes those **Cards** from both **thisDeck** and **visualDeck**. Finally, we return the local **hand** variable, which gives us a **hand** of **Cards** from **thisDeck**.

Now let's move on to the **dealAllPlayers()** method. It needs to know **howManyPlayers** there are and how many **cardsToDeal** to each player. We create the dealtHands and visualHands arrays using **howManyPlayers** for the rows, and **cardsToDeal** for the columns. Then we **shuffle()** the **Cards** in **thisDeck**, and loop though **each** player. We deal **each** player a hand and store it in the local variable, **thisHand**. Next, we loop though the number of **cardsToDeal** and store each card in each column of that player's row, in **dealtHands** and the image of that card in each column of that player's row in **visualHands**. Then for debugging, we print out **each** card in **thisHand** to the console. In the end, we have set this object to have **dealtHands** to **howManyPlayers**, with each player having **cardsToDeal** number of **Cards** in their hand. Then, from another class, we can use this object's **getHand()** method to get the hand of an individual player.

Save and **Run** it and you'll see two players, each being dealt a hand of five cards. This is great, but it hurts my brain to look at words instead of cards! Let's test the visual images with an **Applet**.

In java3_Lesson13, create a new class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson13/src

Package: games

□ Enclosing type:

Name: DisplayHandDemo

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

Create **DisplayHandDemo** by typing the code as shown below in **blue**:

CODE TO TYPE: DisplayHandDemo

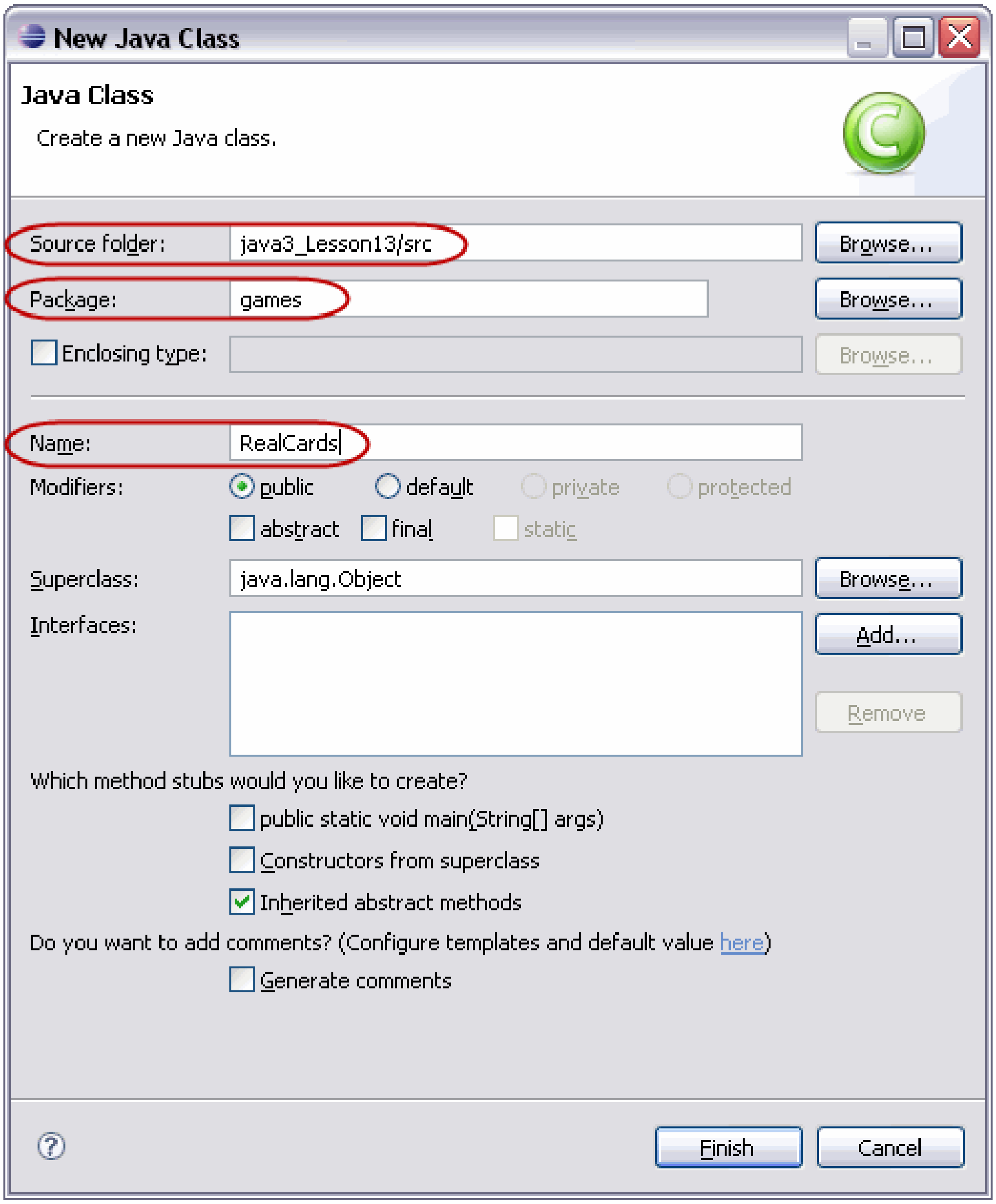
```
package games;

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

public class DisplayHandDemo extends Applet {

    private Deck myDeck = new Deck();
    int n;
    int x, y;
    int numPlayers;
    int numCards;

    public void init() {
        numPlayers = 3;
        numCards = 5;
        myDeck.dealAllPlayers(numPlayers, numCards);
    }

    public void paint(Graphics g) {
        int x = 0;
        int y = 0;

        int width = getWidth();
        int height = getHeight();
        g.setColor(Color.BLUE);
        g.fillRect(0, 0, width, height);

        for(int i = 0; i<numPlayers; i++){
            for(ImageIcon each: myDeck.getVisualHand(i))
            {
                Image justAWTimage = each.getImage();
                g.drawImage(justAWTimage,x, y, this);
                x += 15;
                y += 14;
            }
            x = x + 75;
            y = 0;
        }
    }
}
```

Save and **Run** it. Change the numbers of players and cards in the **init()** method, then **Save** and **Run** it again. With a little more work, you could even create a Blackjack game! Nice.

Moving Images: Mouse Listener

So now we can get images to the **Applet**. That's good, but we want to be able to **identify** a specific card and **move** it. We'll use the code from a web page we mentioned earlier. (Much of the code we're going to use here is from this web page, so thanks again, Mr. Swartz!)

In java3_Lesson13, create a new class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson13/src

Package: games

□ Enclosing type:

Name: RealCards

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

Create **RealCards** by typing the **blue** code as shown:

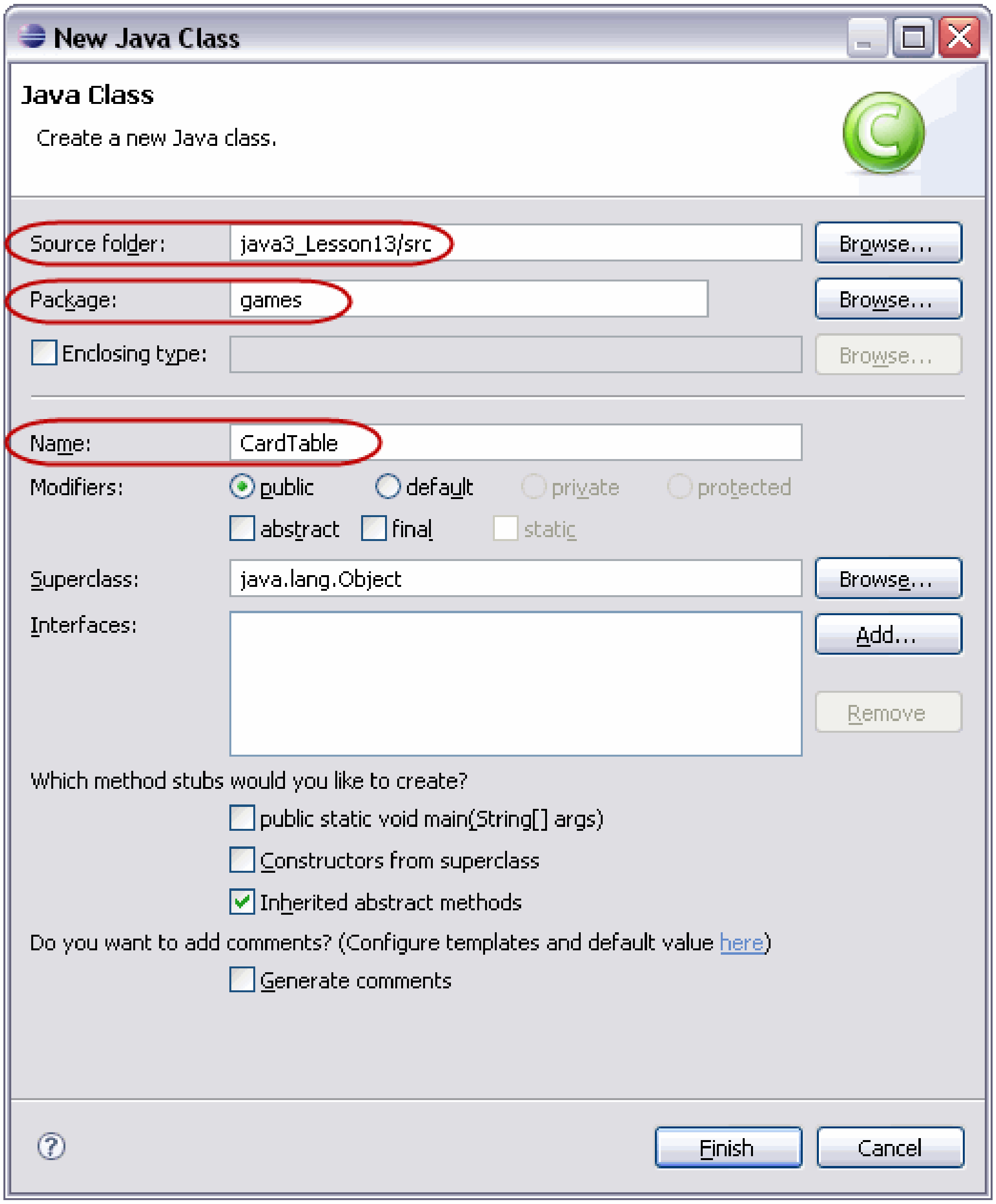
CODE TO TYPE: RealCards

```
// File    : GUI-lowlevel/cards1/cards/Card.java
// Purpose: Represents one card.
// Author : Fred Swartz - February 19, 2007 - Placed in public domain.
//
// Enhancements:
//          * Needs to have Suit and Face value.

package games;

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

class RealCards {

    private ImageIcon image;
    private int       x;
    private int       y;

    public RealCards(ImageIcon image) {
        this.image = image;
    }

    public void moveTo(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public boolean contains(int x, int y) {
        return (x > this.x && x < (this.x + getWidth()) &&
                y > this.y && y < (this.y + getHeight()));
    }

    public int getWidth() {
        return image.getIconWidth();
    }

    public int getHeight() {
        return image.getIconHeight();
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void draw(Graphics g, Component c) {
        image.paintIcon(c, g, this.x, this.y);
    }
}
```

We named this class **RealCards** to differentiate it from **Cards**. This class corresponds more directly to our "visual deck." Its purpose is to put the **RealCards** on a user interface.

Save it. We can't run it though, because there's really nothing to run yet. Let's create two more classes: one class for a "table" to put the cards on, and the other, an application to display and move them.

In java3_Lesson13, create a new class as shown:



Source folder: java3_Lesson13/src

Package: games

□ Enclosing type:

Name: CardTable

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

Create **CardTable** by typing the **blue** code as shown:

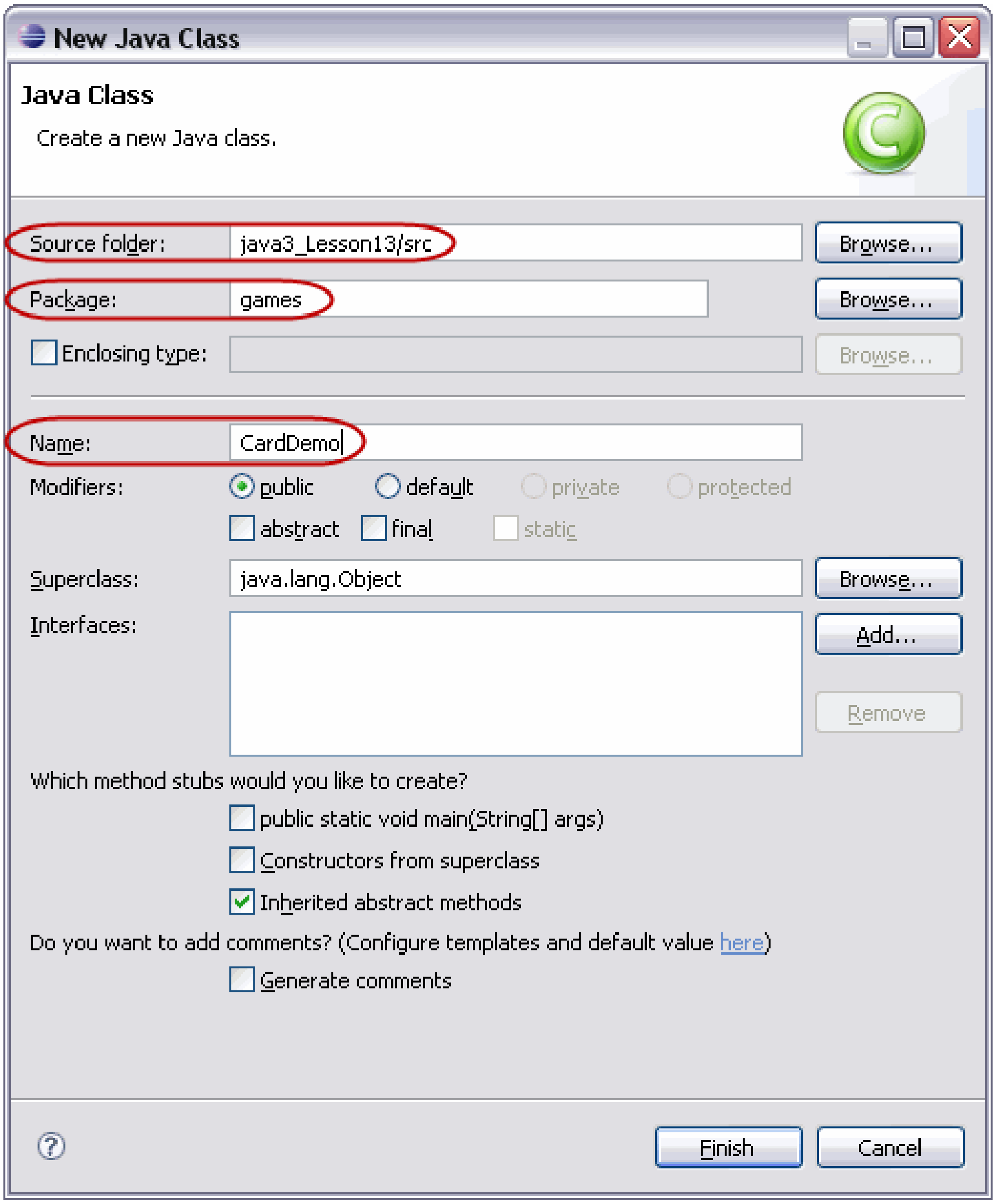
CODE TO TYPE: CardTable

```
// File   : GUI-lowlevel/cards1/cards/CardTable.java
// Purpose: This is just a JComponent for drawing the cards that are
//          showing on the table.
//
// Author : Fred Swartz - February 19, 2007 - Placed in public domain.
//
// Enhancements:
//        * Use model. Currently, it is initialized with a whole deck of cards,
//          but instead it should be initialized with a "model" which
//          it should interrogate (calling model methods) to find out what
//          should be displayed.
//        * Similarly, actions by the mouse might be used to set things in the
//          model, Perhaps by where it's dragged to, or double-clicked, or
//          with pop-up menu, or ...

package games;

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

public class CardTable extends JComponent implements MouseListener, MouseMotionListener
 {

    private static final Color BACKGROUND_COLOR = Color.GREEN;
    private static final int   TABLE_SIZE       = 400;

    private int dragFromX = 0;
    private int dragFromY = 0;

    private RealCards[] deck;
    private RealCards   currentCard = null;

    public CardTable(RealCards[] deck) {
        this.deck = deck;

        setPreferredSize(new Dimension(TABLE_SIZE, TABLE_SIZE));
        setBackground(Color.blue);

        addMouseListener(this);
        addMouseMotionListener(this);
    }

    @Override
    public void paintComponent(Graphics g) {
        //... Paint background
        int width = getWidth();
        int height = getHeight();
        g.setColor(BACKGROUND_COLOR);
        g.fillRect(0, 0, width, height);

        for (RealCards c : this.deck) {
         System.out.println(c.toString());
            c.draw(g, this);
        }
    }

    public void mousePressed(MouseEvent e) {
        int x = e.getX();   // Save the x coord of the click
        int y = e.getY();   // Save the y coord of the click

        //... Find card image this is in.  Check from top down.
        this.currentCard = null;  // Assume not in any image.
        for (int crd=this.deck.length-1; crd>=0; crd--) {
         RealCards testCard = this.deck[crd];
```

```
            if (testCard.contains(x, y)) {
                //... Found, remember this card for dragging.
                dragFromX = x - testCard.getX();  // how far from left
                dragFromY = x - testCard.getY();  // how far from top
                currentCard = testCard;  // Remember what we're dragging.
                break;        // Stop when we find the first match.
            }
        }
    }

    public void mouseDragged(MouseEvent e) {
        if (this.currentCard != null) {

            int newX = e.getX() - dragFromX;
            int newY = e.getY() - dragFromY;

            //--- Don't move the image off the screen sides
            newX = Math.max(newX, 0);
            newX = Math.min(newX, getWidth() - currentCard.getWidth());

            //--- Don't move the image off top or bottom
            newY = Math.max(newY, 0);
            newY = Math.min(newY, getHeight() - currentCard.getHeight());

            this.currentCard.moveTo(newX, newY);

            this.repaint(); // Repaint because position changed.
        }
    }

    public void mouseExited(MouseEvent e) {
        currentCard = null;
    }

    public void mouseMoved   (MouseEvent e) {}  // ignore these events
    public void mouseEntered(MouseEvent e) {}  // ignore these events
    public void mouseClicked(MouseEvent e) {}  // ignore these events
    public void mouseReleased(MouseEvent e) {}  // ignore these events
}
```

Save it. We can't run this either, because it's just a component (actually, it's a **JComponent** that uses package **javax.swing**). We need an applet or an application on which to put the component.

In java3_Lesson13, create another new class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson13/src

Package: games

□ Enclosing type:

Name: CardDemo

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

Create **CardDemo** by adding the **blue** code as shown:

CODE TO TYPE: CardDemo

```
//File   : GUI-lowlevel/cards1/cards/CardDemo
//Purpose: Basic GUI to show dragging cards.
//       Illustrates how to load images from files.
//Author : Fred Swartz - 2007-02-19 - Placed in public domain.
//
//Enhancements:
//     * This really doesn't have a user interface beyond dragging.
//       It doesn't do anything, and therefore has no model.
//       Make it play a game.
//     * Needs to have a Deck class to shuffle, deal, ... Cards.
//       Presumably based on ArrayList<Card>.
//     * Perhaps a Suit and Face class would be useful.
//     * Like Deck, there would also be a class for Hand.
//     * May need Player class too.

package games;

import java.util.List;
import javax.swing.*;



class CardDemo extends JFrame {

 private static RealCards[] _deck = new RealCards[52];

 public static void main(String[] args) {
     CardDemo window = new CardDemo();
     window.setTitle("Card Demo");
     window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     window.setContentPane(new CardTable(_deck));
     window.pack();
     window.setLocationRelativeTo(null);
     window.setVisible(true);
 }

 public CardDemo() {

     int n = 0;         // Which card.
     int xPos = 0;      // Where it should be placed initially.
     int yPos = 0;

      //... Read in the cards using visualDeck from the mapping from Cards.

     Deck myDeck = new Deck();
     List<ImageIcon> aVisualDeck = myDeck.getVisualDeck();
     for(ImageIcon each: aVisualDeck){
         RealCards card = new RealCards(each);
         card.moveTo(xPos, yPos);
         _deck[n] = card;

         //... Update local vars for next card.
         xPos += 5;
         yPos += 4;
         n++;
      }
   }
}
```

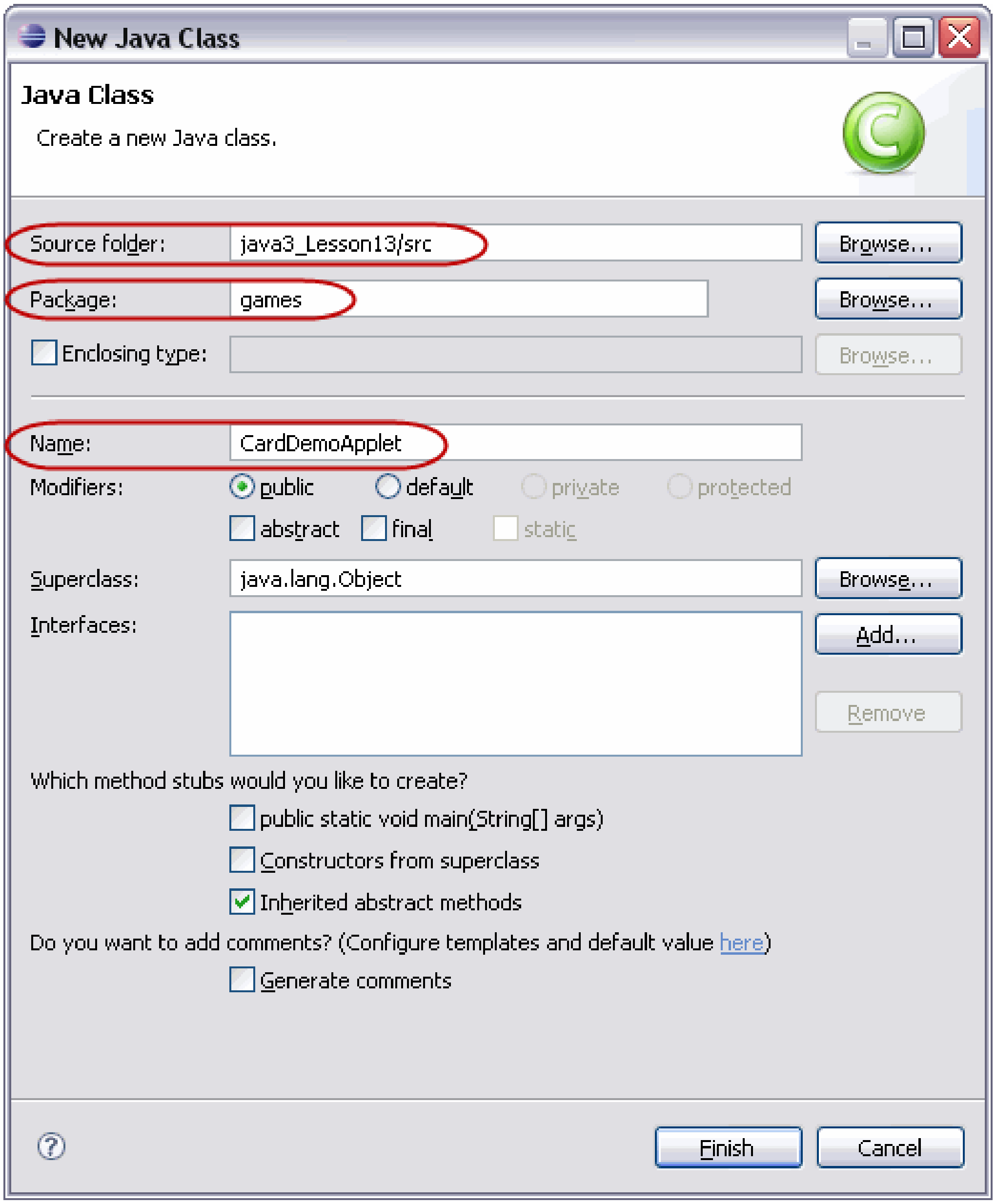
Save and **Run** it. Click on a card and drag it. Do that a few more times.

See how the mouse click can identify exactly which card has been chosen? Trace the code to see how this was done. This is what you'll be doing for your final project.

Now Make It an Applet

Your final project is an Applet.

In java3_Lesson13, create a new class as shown:



New Java Class

Java Class

Create a new Java class.

Source folder: java3_Lesson13/src

Package: games

□ Enclosing type:

Name: CardDemoApplet

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

Create **CardDemoApplet** by typing the **blue** code as shown:

CODE TO TYPE:

```
package games;

import java.util.List;
import javax.swing.*;
import java.applet.Applet;
import java.awt.Graphics;

public class CardDemoApplet extends Applet {

    private static RealCards[] _deck = new RealCards[52];
    CardTable table;

    public void init() {
        resize(400,400);
        makeCards();
        table = new CardTable(_deck);
        add(table);
    }

    public void makeCards(){
        int n = 0;
        int xPos = 0;
        int yPos = 0;

        Deck myDeck = new Deck();
        List<ImageIcon> aVisualDeck = myDeck.getVisualDeck();
        for(ImageIcon each: aVisualDeck)
        {
            RealCards card = new RealCards(each);
            card.moveTo(xPos, yPos);
            _deck[n] = card;

            //... Update local vars for next card.
            xPos += 5;
            yPos += 4;
            n++;
        }
    }

    public void paint(Graphics g) {
        table.paintComponent(g);
    }
}
```

Save and **Run** it. It **works**, but the flicker is nasty. All of the painting makes the graphics area flash each time something is drawn and the graphics area is refreshed.

Swing improved on the **java.awt** package and fixed this flicker problem. We'll look more closely at the **Swing** package in later Java courses. For now, since we're using **java.awt** for our GUI, we'll have to fix it ourselves.

Our problem occurs because of the many successive changes being made to the **paint()** method. The solution is to **paint()** everything to a temporary **buffer** and then **paint()** the whole thing at once. This technique is called **double-buffering**. In our case, our program **paint()**s to another image and then drops the image onto the Applet all at once, to reduce the flicker.

Double Buffer

Edit **CardDemoApplet** as shown in **blue**:

CODE TO TYPE:

```
package games;

import java.util.List;
import javax.swing.*;
import java.applet.Applet;
import java.awt.*;

public class CardDemoApplet extends Applet{
    Graphics bufferGraphics;
    Image doubleBuffer;
    private static RealCards[] _deck = new RealCards[52];
    CardTable table;

    public void init() {
        resize(400,400);
        makeCards();
        table = new CardTable(_deck);
        add(table);
    }

    public void makeCards(){
        int n = 0;
        int xPos = 0;
        int yPos = 0;

        Deck myDeck = new Deck();
        List<ImageIcon> aVisualDeck = myDeck.getVisualDeck();
        for(ImageIcon each: aVisualDeck)
        {
            RealCards card = new RealCards(each);
            card.moveTo(xPos, yPos);
            _deck[n] = card;

            //... Update local vars for next card.
            xPos += 5;
            yPos += 4;
            n++;
        }
    }

    public void update (Graphics g){
        if (doubleBuffer == null)
        {
            doubleBuffer = createImage(this.getSize().width, this.getSize().heig
ht);
            bufferGraphics = doubleBuffer.getGraphics();
        }

        bufferGraphics.setColor(getBackground());
        bufferGraphics.fillRect(0,0,this.getSize().width, this.getSize().height)
;

        bufferGraphics.setColor(getForeground());

        table.paintComponent(bufferGraphics);
        g.drawImage(doubleBuffer, 0,0,this);
    }

    public void paint(Graphics g) {
        update(g);
    }
}
```

Save and **Run** it. That looks pretty good! With these resources available, you're all set for your project.

Graphics Project Examples

Here are a few last examples for you to check out that will help you with your own project. **Open** this demonstration. **Read** the Readme to learn how it works. **Run** it to observe other interesting possibilities for the program.

For your project, you are only required to meet the stated specifications, but feel free to include whatever additional elements you like.

Deploying Applets and Applications Using Eclipse

Java JAR Files

Up until now, we've been using the built-in functionality in Eclipse to run our applets, but that's not how we'll run our Java applets or applications in the real world.

As you know, Eclipse is a pretty powerful Integrated Development Environment (IDE). We've explored very few of its abilities so far in these courses. One of the most useful tasks that Eclipse has performed for us is to package our applets and applications for deployment. In this lesson, we'll try something new. Instead of copying the .class files out of the **/bin** directory, we'll deploy our applets and applications in .jar files. The .jar file gets its name from the **J**ava **AR**chive file format. A JAR file is a ZIP file with an added directory for a MANIFEST file, which identifies the contents of the JAR file.

We'll create a basic applet and application and deploy them using the built-in packaging facility in Eclipse.

Deploying Applets in a JAR File

Create a new Java project named **Lesson14_JarAppletExample** in your **Java3_Lessons** working set. In that project, create a new Java class named **JarExampleApplet**, with **java.applet.Applet** as its super class.

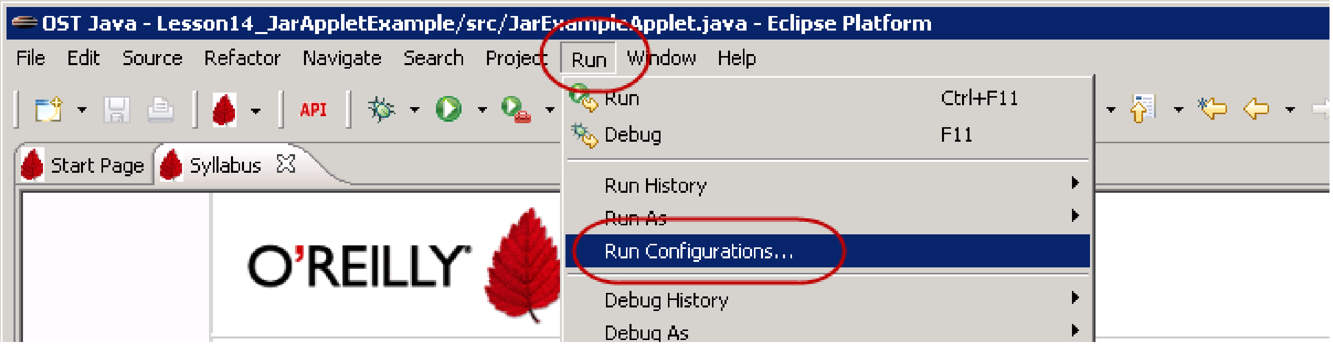
Edit the **JarExampleApplet** as shown below:

CODE TO TYPE: JarExampleApplet

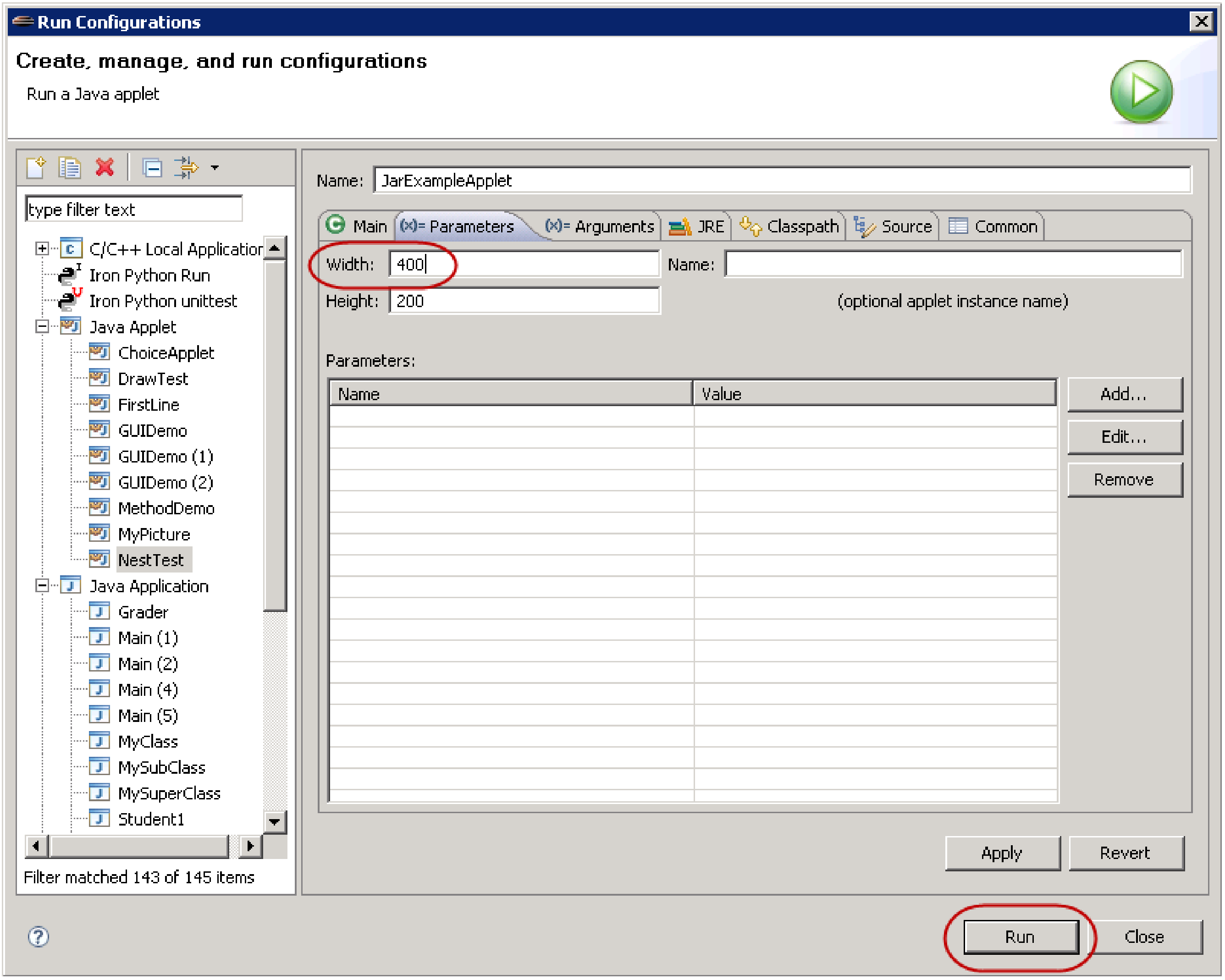
```
import java.applet.Applet;
import java.awt.Graphics;


public class JarExampleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("This Applet was read from a .jar file.", 0, 25);
    }
}
```

The applet itself isn't the focus of this lesson so we kept it short. We just print a message out to the Graphics area to make sure the applet has run.

Select **Run | Run Configurations**.



This dialog appears:



Name: JarExampleApplet

Width: 400

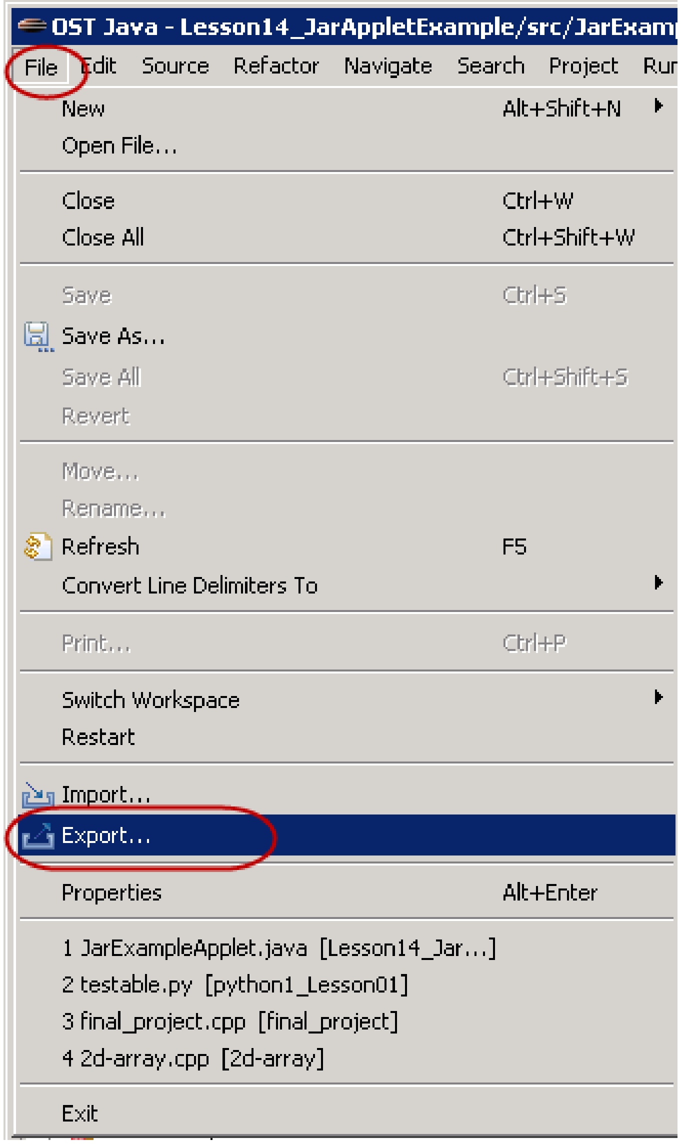
Height: 200

Name:

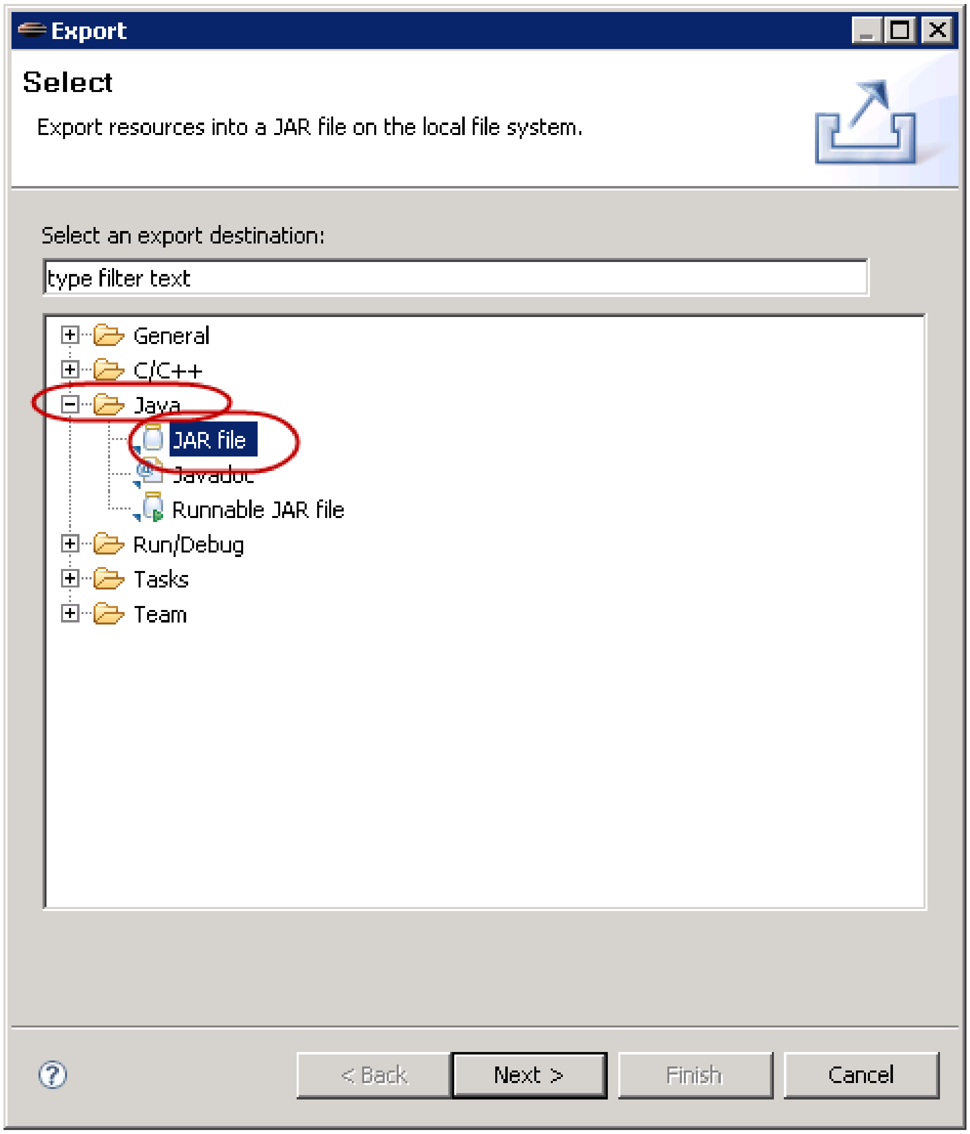
(optional applet instance name)

If **Lesson14_JarApplet** is not selected in the left column, select it. Select the **Parameters** tab. Change the Width parameter to **400** and select **Apply** or **Run**. Either selection will set the parameter for future runs of the applet, but **Run** will also close the dialog box and run the applet.

Now let's get to the deployment of an applet using the built-in Export feature of Eclipse. Select **File | Export**:



This opens the Export Dialog:



Select

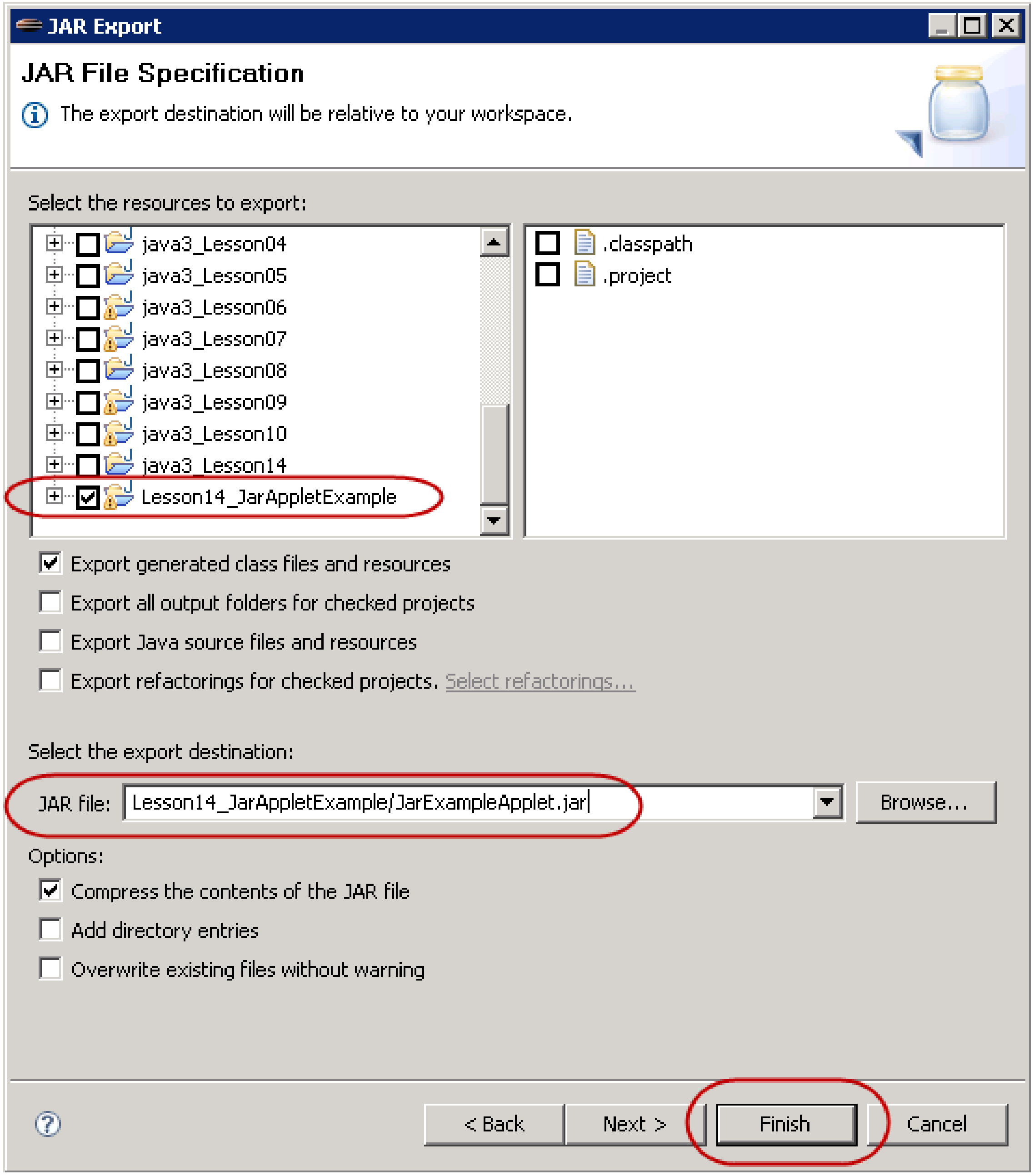
Export resources into a JAR file on the local file system.

Select an export destination:

type filter text

Expand the **Java** entry and select **JAR file**. Then, click **Next** to go to the next dialog.

Select the project you want to export and also select the path where you want the .jar file exported. In our case, we want to export the **Lesson14_JarAppletExample** project and we want the resulting .jar file in our **Lesson14_JarApplet** directory. Click **Finish** to complete the export process.



JAR File Specification

The export destination will be relative to your workspace.

Select the resources to export:

- □ java3_Lesson04
- □ java3_Lesson05
- □ java3_Lesson06
- □ java3_Lesson07
- □ java3_Lesson08
- □ java3_Lesson09
- □ java3_Lesson10
- □ java3_Lesson14
- ☑ Lesson14_JarAppletExample

- □ .classpath
- □ .project

- ☑ Export generated class files and resources
- □ Export all output folders for checked projects
- □ Export Java source files and resources
- □ Export refactorings for checked projects. Select refactorings...

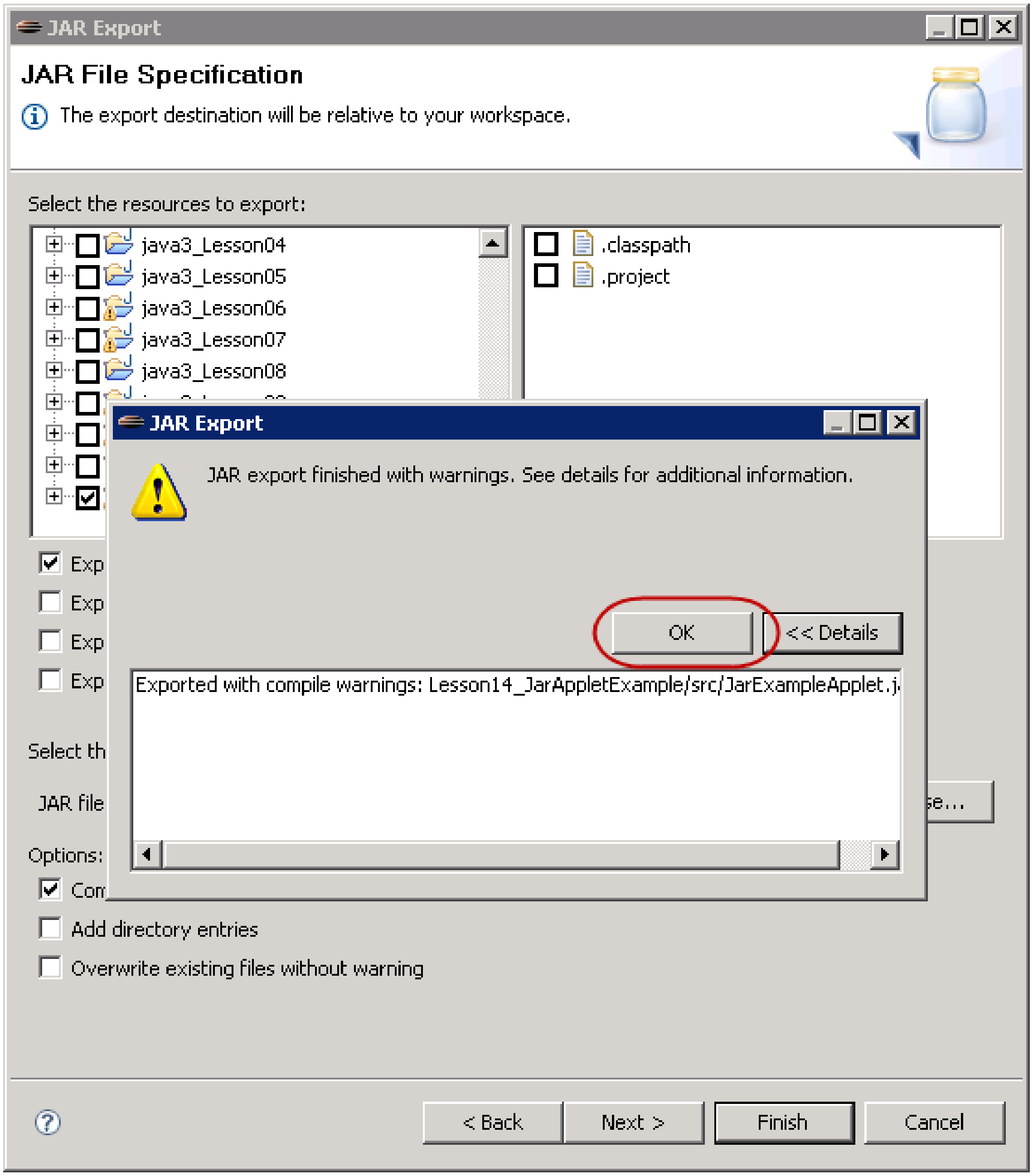
Select the export destination:

JAR file: Lesson14_JarAppletExample/JarExampleApplet.jar

Options:

- ☑ Compress the contents of the JAR file
- □ Add directory entries
- □ Overwrite existing files without warning

We did not give our applet a SerialVersionUID constant, so we will have compiler warnings. The export process lets us know that there were warnings. Click **OK** on the warning dialog:



JAR File Specification

The export destination will be relative to your workspace.

Select the resources to export:

- ⊞ □ java3_Lesson04
- ⊞ □ java3_Lesson05
- ⊞ □ java3_Lesson06
- ⊞ □ java3_Lesson07
- ⊞ □ java3_Lesson08

- □ .classpath
- □ .project

JAR export finished with warnings. See details for additional information.

Exported with compile warnings: Lesson14_JarAppletExample/src/JarExampleApplet.j

- ☑ Com
- □ Add directory entries
- □ Overwrite existing files without warning

The .jar file now appears in your project in the Package Explorer.

Because this is an applet, we need to create an .html file to load the applet into a browser. Right-click on the **Lesson14_JarAppletExample** project and select **New | HTML file**. If HTML File is not on the menu, select **Other** and then go to the **Web** item.

Create a new HTML file named **jarAppletExample.html**. Edit the new file as shown:

CODE TO TYPE: jarAppletExample.html

```
<html>
<head>
<title>Jar Applet Example</title>
</head>
<body>
<applet code="JarExampleApplet.class" archive="JarExampleApplet.jar" height="200" width
="400"></applet>
</body>
</html>
```

OBSERVE: jarAppletExample.html

```
<html>
<head>
<title>Jar Applet Example</title>
</head>
<body>
<applet code="JarExampleApplet.class" archive="JarExampleApplet.jar" height="200" width
="400"></applet>
</body>
</html>
```

In the .html file, the **code** attribute tells the browser which class to load as the applet. The **archive** attribute informs the browser of the location the .jar file that contains the code. The **height** and **width** attributes tell the browser how big to make the Graphics area for the applet. In this case, the .html file and the JarExampleApplet.jar file are in the same directory.

Unfortunately, Eclipse cannot run applets on a web page within Eclipse itself. The workaround for this is to open a system web browser, using the **Web Browser** button at the top of this tab. Then type: **V:\workspace\Lesson14_JarApplet\jarAppletExample.html** into the browser location text box and you'll see the applet at work.

The applet's HTML file loads into the browser. The browser's Java Plugin finds the Applet tag and sets up a Graphics area according to the specifications indicated by the tag. Then the browser's Java Plugin retrieves the indicated .class file from the archive file and loads it as an applet.

This allows us to put the .html file and the .jar file into our web space and access that applet via the web.

Deploying Applications in a Jar File

Creating an Application for Deployment

The process for deploying a Java application is similar to that of deploying an applet from a JAR file. Create a new project in your **Java3_Lessons** working set, named **Lesson14_JarApplication**.

Create a new class in your project, named **JarExampleApplication**. This class should have **javax.swing.JFrame** as its superclass.

Edit the class as shown in **blue**:

CODE TO TYPE: JarExampleApplication

```
import java.awt.Graphics;

import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class JarExampleApplication extends JFrame {
    public JarExampleApplication() {
        //Make the X close the application.
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        setSize(400, 400);
    }
    public void paint(Graphics g) {
        //A good idea to call super.paint() to make sure all components get repa
inted.
        super.paint(g);
        g.drawString("This Application ran from a jar file", 10, 150);
    }

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

OBSERVE: JarExampleApplication

```
import java.awt.Graphics;

import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class JarExampleApplication extends JFrame {
    public JarExampleApplication() {
        // Make the X close the application.
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        setSize(400, 400);
    }

    public void paint(Graphics g) {
        // A good idea to call super.paint() to make sure all components get
        // repainted.
        super.paint(g);
        g.drawString("This Application ran from a jar file", 10, 150);
    }

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

In this application, we use the **Swing Framework** and a **JFrame**, which is a GUI window for Java applications. We'll learn more about the **Swing Framework** in Java 4.

In the constructor in the application above, we **setDefaultCloseOperation()**, passing the **WindowConstants.EXIT_ON_CLOSE** constant. This tells the **JFrame** to close when the user clicks on the **X** close icon of the Window. If we did not set **setDefaultCloseOperation()**, the **JFrame** would be hidden, but continue running in memory.

We call **super.paint(g)** in the **paint()** method. This is to ensure any lightweight components (Swing components) are redrawn correctly.

Note The Java Container API for paint() has important concepts for you to absorb.

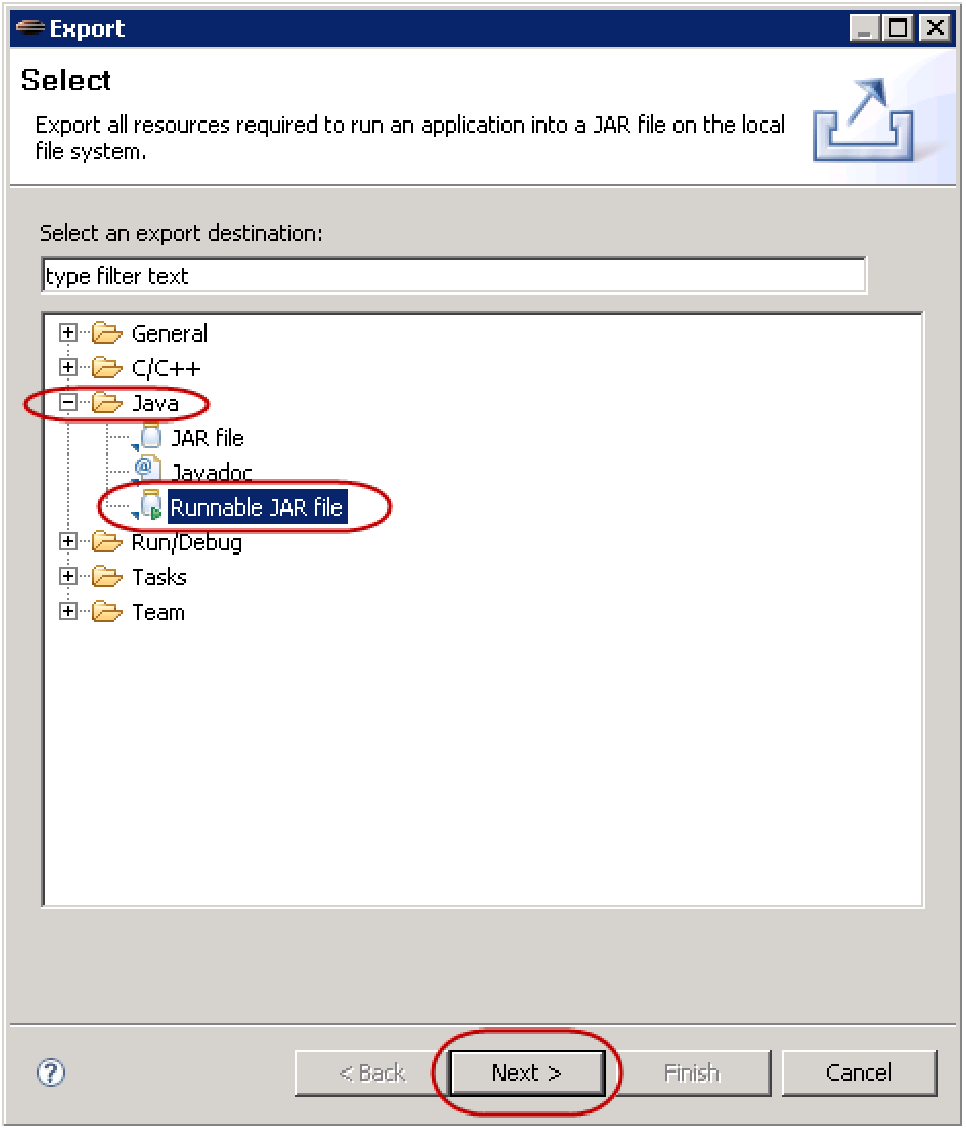
In the **main()** method, we create an instance of the **JarExampleApplication** class and make it display itself. While the GUI is visible, the application sits in an event loop waiting for user input. When the **X** close icon is clicked, the application will quit.

▶ Save and run the application to make sure its Run Configuration is up to date.

To see the Run Configuration for this application, select **Run | Run Configuration**. The configuration for this application will be used to create the self-executable JAR file in the next section.

Deploying the Application in an Executable Jar File

Despite some similarities, deploying an application is not exactly like deploying an applet. Select **File | Export**, and select the **Runnable Jar File** option:



Export

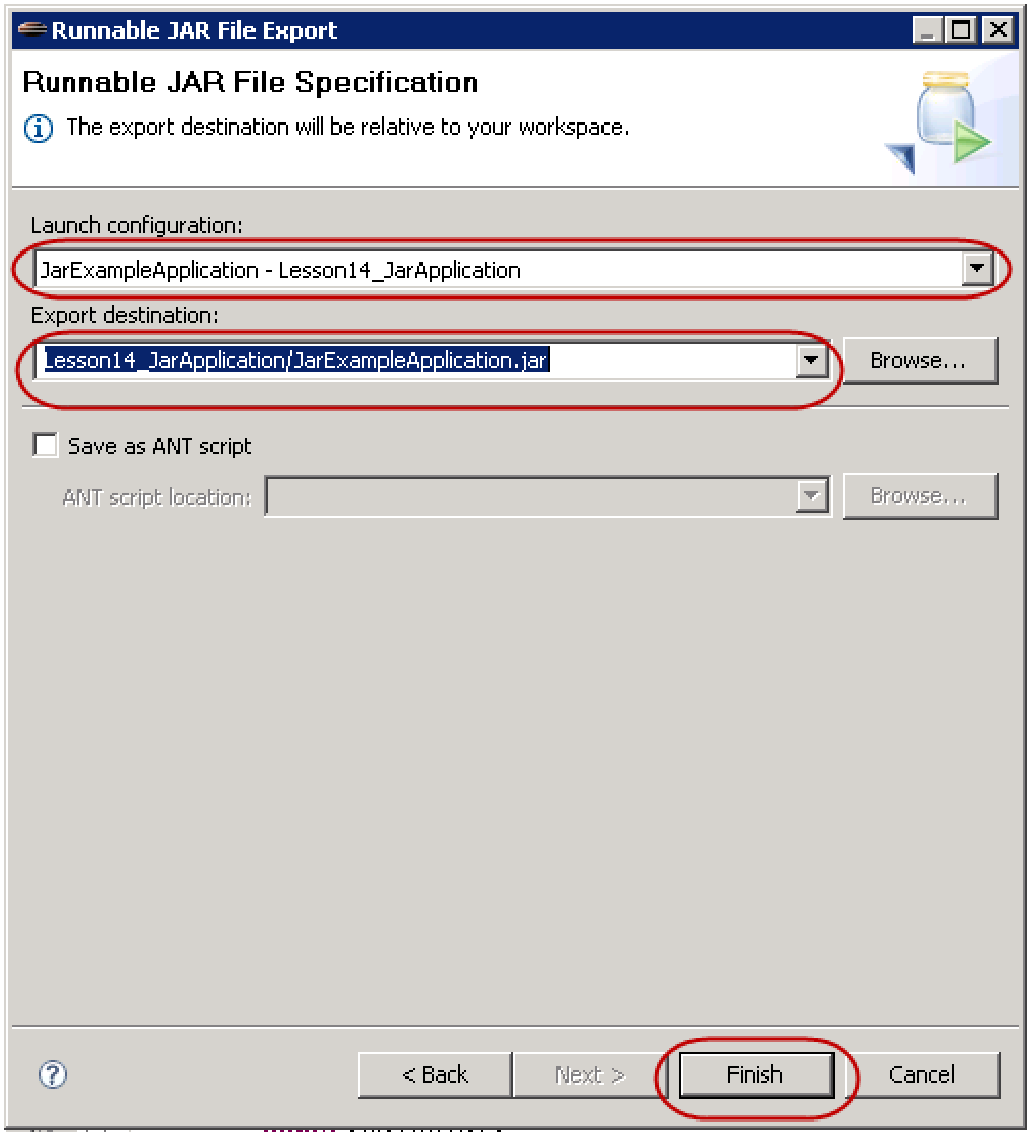
Select

Export all resources required to run an application into a JAR file on the local file system.

Select an export destination:

type filter text

Select **Next** to open a dialog box or go to the Run menu to select a Run Configuration to use as a template for running the application. Select the project directory as a path location where the JAR file will be stored. (It could actually go anywhere on the file system.) Complete the dialog as shown:



Runnable JAR File Specification

The export destination will be relative to your workspace.

Launch configuration: JarExampleApplication - Lesson14_JarApplication

Export destination: Lesson14_JarApplication/JarExampleApplication.jar

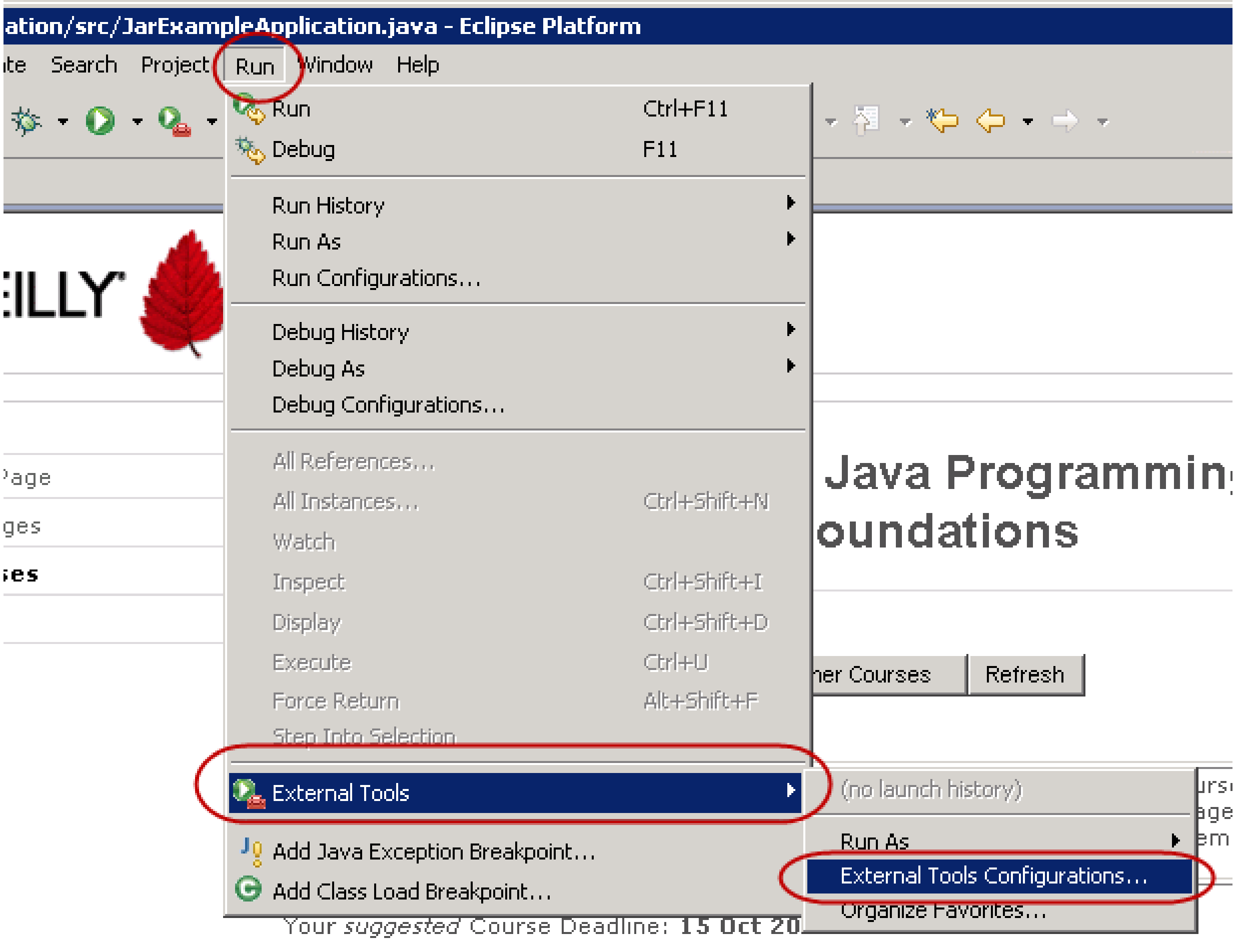
Click **Finish**.

Running an Executable Jar File

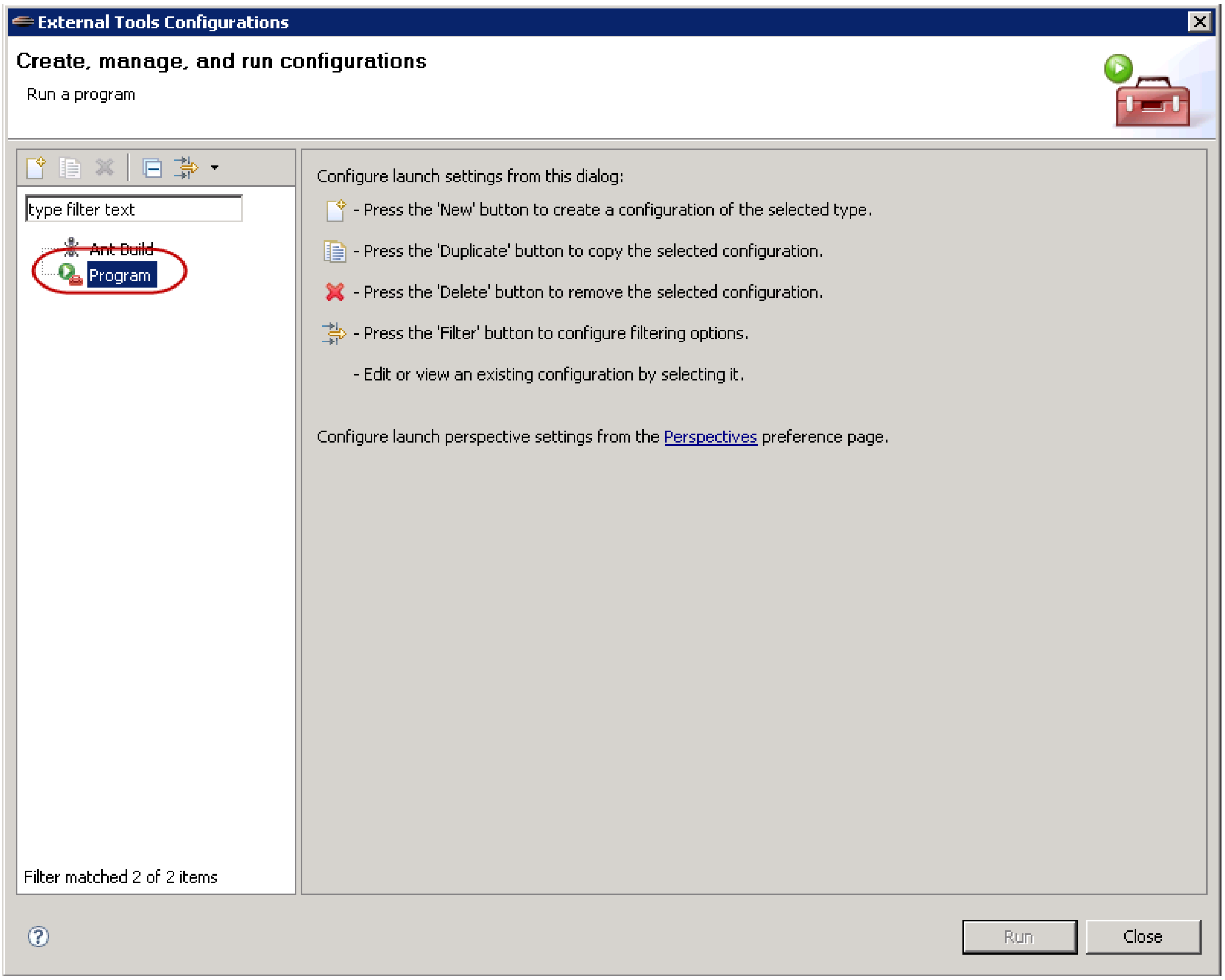
Now that we have an executable JAR file, we'll want to know how to run it. Normally, we could just drop it onto the desktop and double-click it to see it run; but, since our Terminal Servers do not have desktops, we'll have to do a little more work.

First, let's set up an **External Tool** to run the Windows Command Line.

Select **Run | External Tools | External Tools Configurations...**.



Double-click the **Program** item in the left pane of the dialog:



External Tools Configurations
Create, manage, and run configurations
Run a program
type filter text
Ant Build
Program
Run
Close

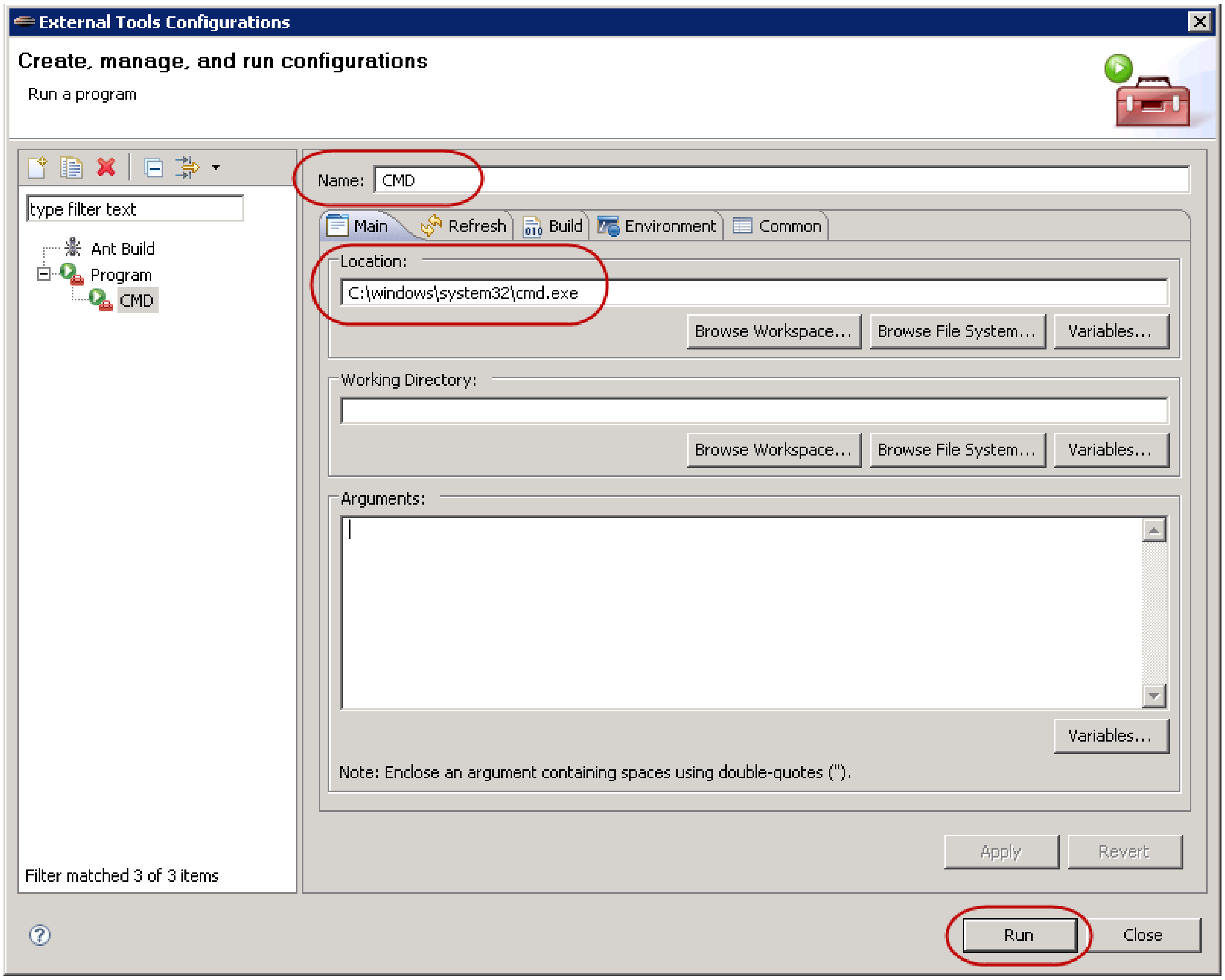
Configure launch settings from this dialog:

- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the Perspectives preference page.

Filter matched 2 of 2 items

This opens a new pane on the right side of the dialog. In the Name text box, type **CMD**. In the Location text box, type **C:\windows\system32\cmd.exe**, and click **Run**. This will run the Windows Command Line interpreter in the Console window of Eclipse. It will also set the External Tool **CMD** in the External Tools Menu.



Create, manage, and run configurations

Run a program

Name: CMD

Location: C:\windows\system32\cmd.exe

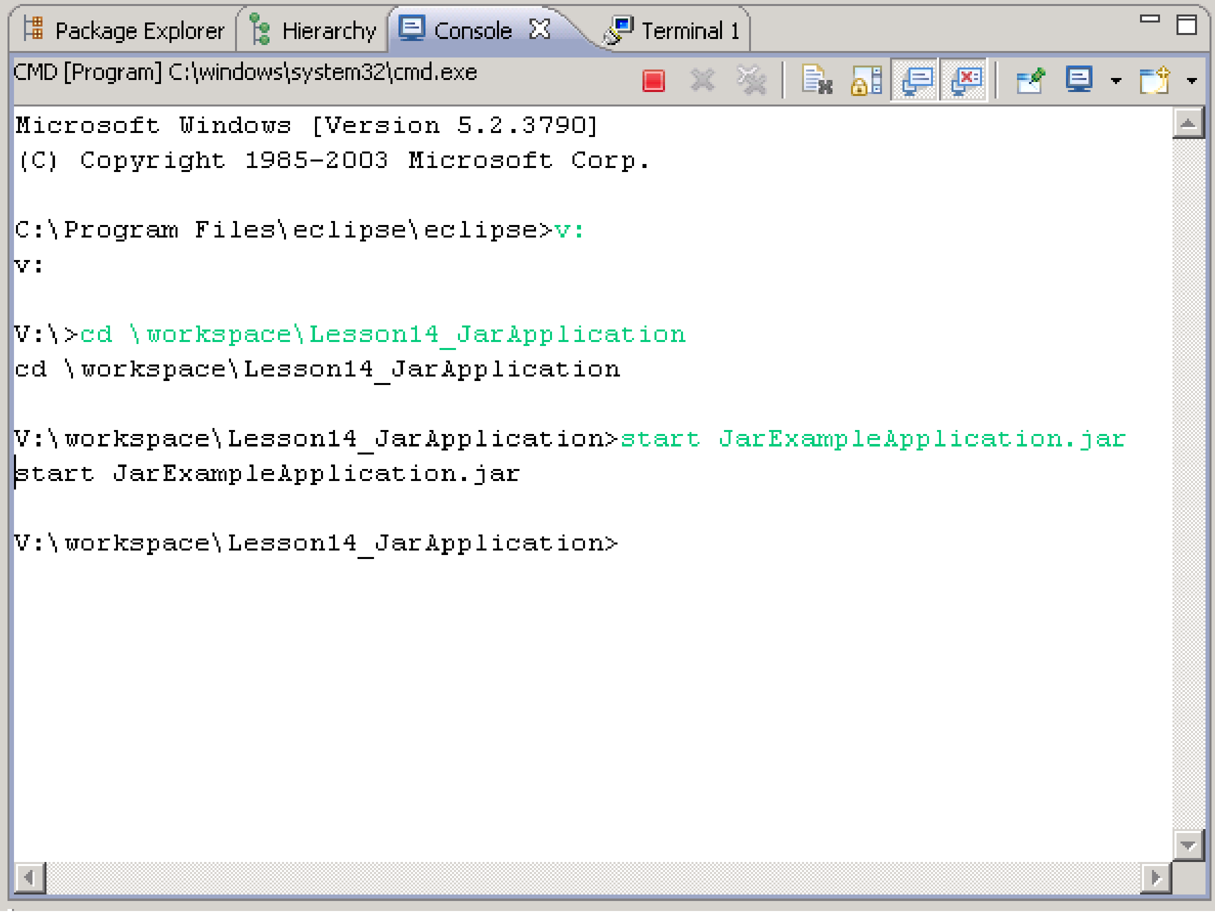
Note: Enclose an argument containing spaces using double-quotes (").

Filter matched 3 of 3 items

Click in the Console view of Eclipse, type **V:**, and press **Enter**. This will switch us over to the V: drive.

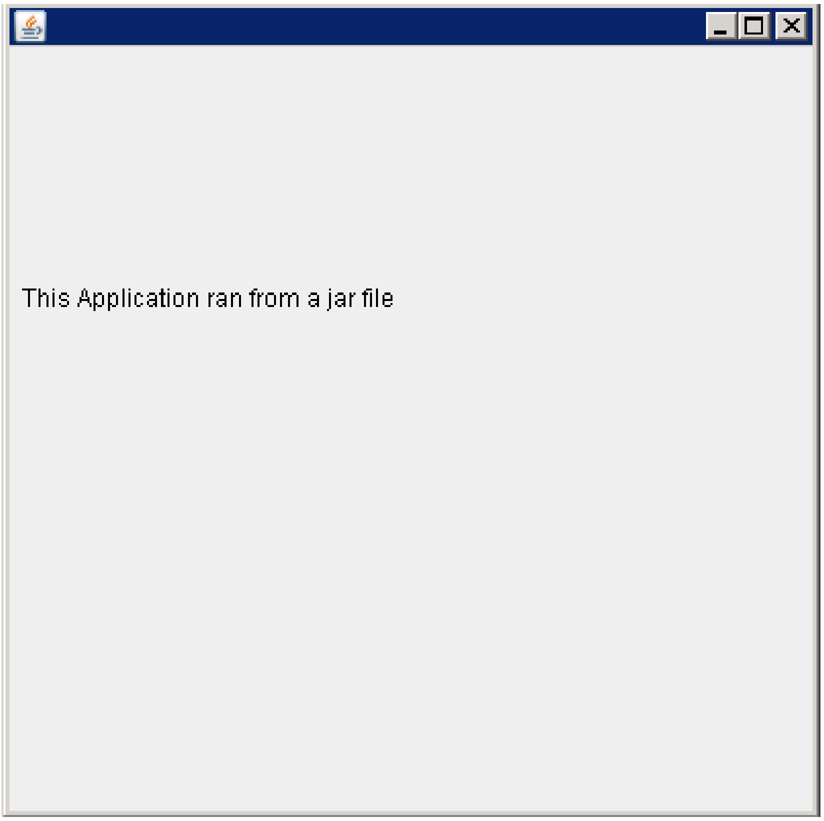
Type **cd \workspace\Lesson14_JarApplication** to switch to the directory where JarExampleApplication.jar file is located.

Type **start JarExampleApplication.jar** to start the application. This will automatically run Java and load up the application. You can see the application running in the image here (your input from above is seen in light green):



CMD [Program] C:\windows\system32\cmd.exe

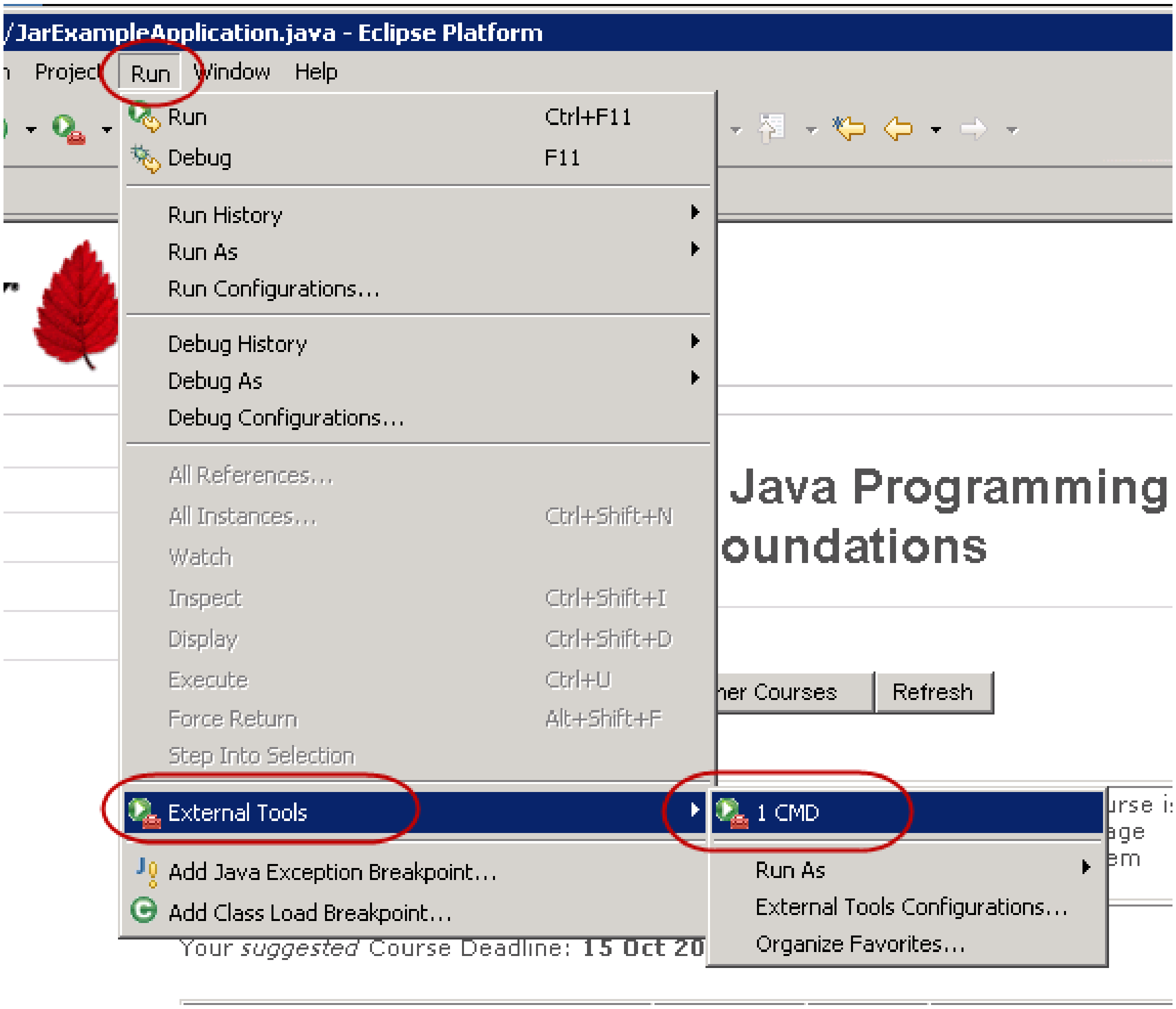
```
Microsoft Windows [Version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.

C:\Program Files\eclipse\eclipse>v:
v:

V:\>cd \workspace\Lesson14_JarApplication
cd \workspace\Lesson14_JarApplication

V:\workspace\Lesson14_JarApplication>start JarExampleApplication.jar
start JarExampleApplication.jar

V:\workspace\Lesson14_JarApplication>
```



This Application ran from a jar file

To stop the application, click the **X** close icon in the JFrame. To stop the Command Line Interpreter, click the **red square** in the Console view.

Click on the **Run | External Tools** menu item. There is now a **CMD** entry in the sub-menu. This can be used to run the Windows Command Line Interpreter at any time.



Double-click on an executable JAR file in order to run it.

Great job today. Keep practicing and playing around with the examples until you are one with Java JAR files. See you in the next and final lesson for Java 3!

Working With Files

Working With Files

There are lots of ways to manage files in Java. In this lesson, we'll go over options we have when working with text files.

The File Class

The File class lets us create files and directories to use when we work with the various file streams, readers, and writers, provided by the Java API.

Create a new Java project named **Java3_FileIO**. In that project, create a new class named **FileIO**. Edit the **FileIO** class by adding the **blue** code as shown:

CODE TO TYPE: FileIO.java

```
import java.io.File;
import java.io.IOException;

public class FileIO {
    File myFile;

    public FileIO(String path) {
        myFile = new File(path);
        try {
            myFile.createNewFile();
        } catch (IOException e) {
            e.printStackTrace();
            //exit with an exit code. Exit code 0 indicates normal exit.
            System.exit(1);
        }
    }

    public static void main(String [] args) {
        FileIO fileTest = new FileIO("filetest.txt");
    }
}
```

Save and run it. Right-click on the Package Explorer and select Refresh from the pop-up menu. Now the **filetest.txt** file is in your project. If we use a file name as the path, the current directory will be used to create the file:

OBSERVE: FileIO.java

```
import java.io.File;
import java.io.IOException;

public class FileIO {
    File myFile;

    public FileIO(String path) {
        myFile = new File(path);
        try {
            myFile.createNewFile();
        } catch (IOException e) {
            e.printStackTrace();
            //exit with an exit code. Exit code 0 indicates normal exit.
            System.exit(1);
        }
    }

    public static void main(String [] args) {
        FileIO fileTest = new FileIO("filetest.txt");
    }
}
```

Let's break down our example. In the **FileIO** class, we create a **File** reference variable, named **myFile**. In the **Constructor**, we create a **File** object from the **path String**. The file itself does not yet exist. We **try** to **createNewFile()**; if **createNewFile()** is unsuccessful, we **catch** (**IOException e**), and find out the reason the file could not be created. If **printStackTrace()**is in the **IOException**, the class will print a listing of the Exception in the console. If the file can't be created, we **exit()** the program with an exit code. Exit codes that are non-zero, indicate that there was a problem. It is up to the programmer to define exit codes; there are no hard and fast rules. Exit codes should be used anytime the program is exiting in an abnormal fashion. These codes can be read by batch processes.

Note Remember, just creating the **File** object does not actually create the file on the disk.

Now let's make our code more useful by adding the code in **blue** and removing the code in **red**:

CODE TO TYPE: FileIO.java

```
import java.io.File;
import java.io.IOException;
import java.lang.reflect.InvocationTargetException;

public class FileIO {
    File myFile;

    public FileIO(String path) {
        myFile = new File(path);
        ~~try {~~
~~            myFile.createNewFile();~~
~~        } catch (IOException e) {~~
~~            e.printStackTrace();~~
~~            //exit with an exit code. Exit code 0 indicates normal exit.~~
~~            System.exit(1);~~
~~        }~~
    }

    public boolean deleteFile() {
        return myFile.delete()
    }

    public File getFile() {
        return myFile;
    }

    public void setFile(String path) {
        myFile = new File(path);
    }

    public void createFile() throws InvocationTargetException{
        try {
            myFile.createNewFile();
        }
        catch(IOException e) {
            throw new InvocationTargetException(e);
        }
    }

    public static void main(String [] args) {
        String path = "myFile/filetest.txt";
        int exitCode = 0;
        FileIO fileTest = new FileIO(path~~"filetest.txt"~~);

        try {
            fileTest.createFile();
        } catch (InvocationTargetException e) {
            e.getCause().printStackTrace();
            exitCode = 1;
        }
        finally {
            System.exit(exitCode);
        }
    }
}
```

Save and Run the program. There is an error in the Console. This is because the directory **myFile** does not exist. The path String indicates that the full path is **myFile/filetest.txt**, but the **filetest.txt** file cannot be put into a directory that does not exist.

Let's fix that real quick before we look at all of the code. Add the **blue** code as shown:

CODE TO TYPE: FileIO.java

```java
import java.io.File;
import java.io.IOException;
import java.lang.reflect.InvocationTargetException;

public class FileIO {
    File myFile;

    public FileIO(String path) {
        myFile = new File(path);
    }

    public boolean deleteFile() {
        return myFile.delete()
    }

    public File getFile() {
        return myFile;
    }

    public void setFile(String path) {
        myFile = new File(path);
    }

    public void createFile() throws InvocationTargetException{
        try {
            File dirFile = myFile.getParentFile();
            dirFile.mkdirs();
            myFile.createNewFile();
        }
        catch(IOException e) {
            throw new InvocationTargetException(e);
        }
    }

    public static void main(String [] args) {
        String path = "myFile/filetest.txt";
        int exitCode = 0;
        FileIO fileTest = new FileIO(path);

        try {
            fileTest.createFile();
        }
        catch (InvocationTargetException e) {
            e.getCause().printStackTrace();
            exitCode = 1;
        }
        finally {
            System.exit(exitCode);
        }
    }
}
```

Save and Run the program. Ah, it works now! Right-click the **Java3_FileIO** project in the Package Explorer and select **Refresh** to see the directory and file in your project.

There's a lot going on in this little piece of code, so let's break it down:

OBSERVE: FileIO.java

```java
import java.io.File;
import java.io.IOException;
import java.lang.reflect.InvocationTargetException;

public class FileIO {
    File myFile;

    public FileIO(String path) {
        myFile = new File(path);
    }

    public void deleteFile() {
        myFile.delete();
    }

    public File getFile() {
        return myFile;
    }

    public void setFile(String path) {
        myFile = new File(path);
    }

    public void createFile() throws InvocationTargetException{
        try {
            File dirFile = myFile.getParentFile();
            dirFile.mkdirs();
            myFile.createNewFile();
        }
        catch(IOException e) {
            throw new InvocationTargetException(e);
        }
    }

    public static void main(String [] args) {
        String path = "myFile/filetest.txt";
        int exitCode = 0;
        FileIO fileTest = new FileIO(path);

        try {
            fileTest.createFile();
        }
        catch (InvocationTargetException e) {
            e.getCause().printStackTrace();
            exitCode = 1;
        }
        finally {
            System.exit(exitCode);
        }
    }
}
```

First, we removed the file creation from the constructor. That's so we can separate the actions this program can take. We make sure that myFile is not null, that way we can act on it in other methods later, without having to check to make sure it is null then.

We've added the methods, **deleteFile()**, **setFile()**, **getFile()**, and **createFile()** to the program so that we can perform those actions on the **myFile** object.

We are adding something new to the **createFile()** method: the **throws** clause at the end of the method header. It lets us know that this method can throw an Exception. To be specific, it can throw the **InvocationTargetException**. The **InvocationTargetException** is a checked exception that allows us to pass an exception along to another method. We surround everything we are doing in the **createFile()** method within a **try{}** block, and **catch()** any **IOException**s that occur. The **try{}** block gets the prefix of the path in the **myFile** object, as a **File** object named **dirFile** using the **getParentFile()** method of the **File** class (**myFile** object).

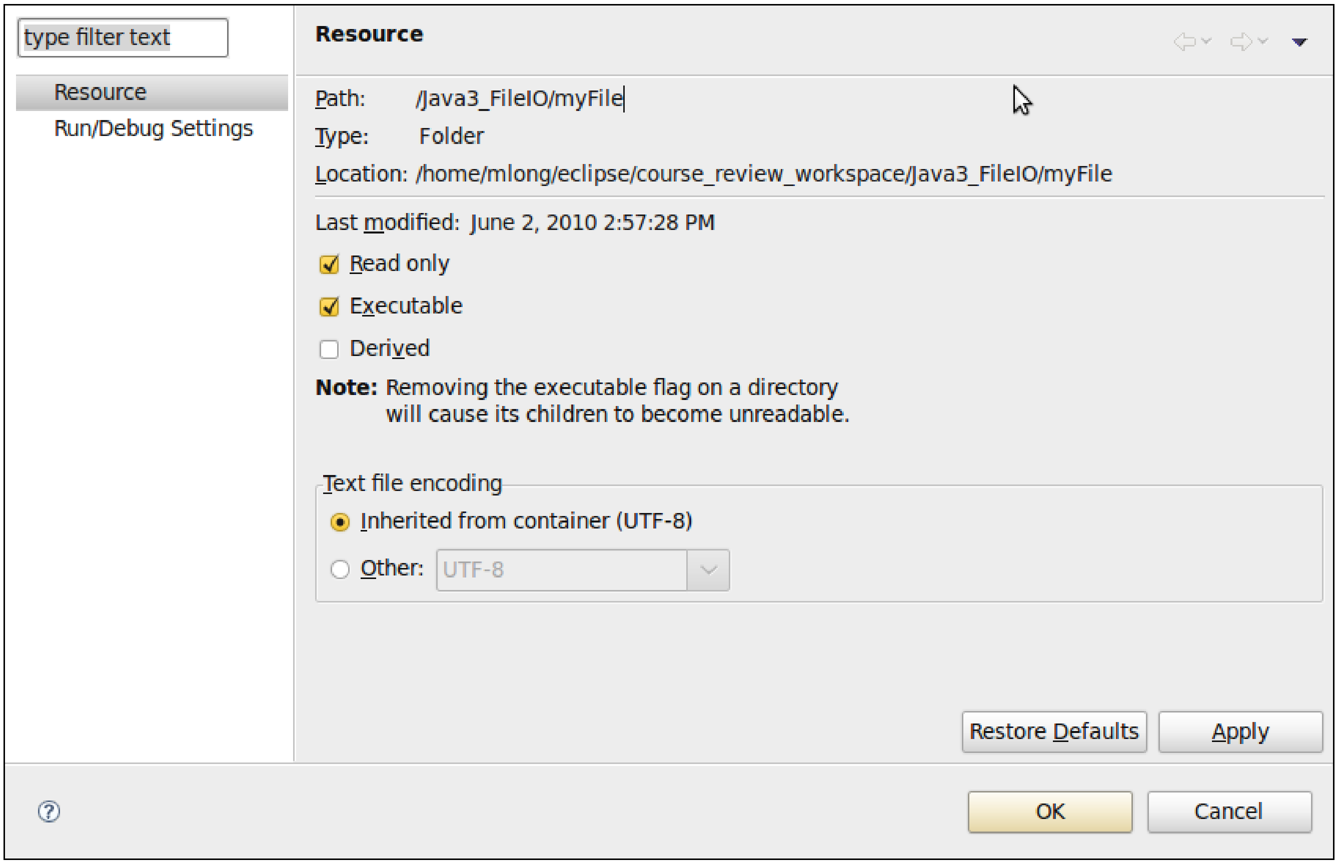
Then the **getParentFile()** method returns a **File** object that represents everything in the **myFile** object's path, **except** for the last segment, which, in our example, is *filetest.txt*. After we get that file, we can use the **dirFile.mkdirs()** method to create the directory structure we need to store the file represented by the **myFile** object. The **mkdirs()** method will create all directories in the directory structure represented, if they do not already exist. We could also use the **mkdir()** method to create a single directory. Typically, it's better to use the more comprehensive **mkdirs()** method.

Once the directory structure is built, we can then tell **myFile** to **createNewFile()**, in order to create the file on the disk. If an **IOException** is encountered, the **catch** clause will catch it, and then **throw** a **new InvocationTargetException**, passing the **IOException** object **e** as its parameter. This allows us to pass the **IOException** to whichever method called our **createFile()** method, which prevents our **createFile()** method from having to handle the exception. Since **InvocationTargetException** is checked by the compiler, any method that uses our **createFile()** method must either handle the exception or **throw** it.

Finally, we instantiate the **FileIO** class in the **main()** method. We create a **path String**, giving it the value of the path/file we want to create. We also create an **int exitCode** variable to hold the exit code value we will use to terminate the program.

then, in the **try{}** block, we tell our **fileTest** variable to create the file, using its **createFile()** method. If an **InvocationTargetException** is caught, we use the exception object **e** and the **getCause()** method to get the original **IOException** that occurred in our **createFile()** method. Next, we tell that returned object to **printStackTrace()**, which will print out the exception trace to the console. After that, we change the **exitCode** variable to a non-zero value. The **finally{}** block will execute, regardless of whether there was an exception caught. Then we tell the program to exit using the current **exitCode**.

Before we move on to the next modification, let's make sure we are really catching **IOExceptions**. Right-click on the **filetest.txt** file in the package explorer and delete it. Now, right-click on the **myFile** directory and select **Properties**. Check the Read Only box in the properties dialog and select **Apply** and close the dialog:



Path: /Java3_FileIO/myFile

Type: Folder

Location: /home/mlong/eclipse/course_review_workspace/Java3_FileIO/myFile

Last modified: June 2, 2010 2:57:28 PM

☑ Read only

☑ Executable

☐ Derived

Note: Removing the executable flag on a directory will cause its children to become unreadable.

Text file encoding

● Inherited from container (UTF-8)

○ Other: UTF-8

There is an **IOException** in the console. Change the Read Only property back to what it was before on the **myFile** directory.

There is a mysterious problem in our program. Modify the code by adding the **blue** code and removing the ~~red~~ code as shown:

CODE TO TYPE: FileIO.java

```
import java.io.File;
import java.io.IOException;
import java.lang.reflect.InvocationTargetException;

public class FileIO {
    File myFile;

    public FileIO(String path) {
        myFile = new File(path);
    }

    public boolean deleteFile() {
        return myFile.delete()
    }

    public File getFile() {
        return myFile;
    }

    public void setFile(String path) {
        myFile = new File(path);
    }

    public void createFile() throws InvocationTargetException{
        try {
            File dirFile = myFile.getParentFile();
            dirFile.mkdirs();
            myFile.createNewFile();
        }
        catch(IOException e) {
            throw new InvocationTargetException(e);
        }
    }

    public static void main(String [] args) {
        String path = "~~myFile/~~filetest.txt";
        int exitCode = 0;
        FileIO fileTest = new FileIO(path);

        try {
            fileTest.createFile();
        } catch (InvocationTargetException e) {
            e.getCause().printStackTrace();
            exitCode = 1;
        }
        /*finally {
            System.exit(exitCode);
        }*/
    }
}
```

Save and Run the program. Notice the exception in the console? It's there because our **path String** no longer contains a parent directory for our file. If our parent directory isn't there, the **createFile()** method can't create the **dirFile** object using the **File** class's **getParentFile()** method. **dirFile** is null and we can't act on a null object. We have to comment out the **finally{}** block because our code is inside of a try/catch block, but our code doesn't handle the **NullPointerException**, so it is ignored when we explicitly call **System.exit()**. We'll fix that in a moment.

Modify the code by adding the **blue** code and removing the ~~red~~ code as shown:

CODE TO TYPE: FileIO.java

```
import java.io.File;
import java.io.IOException;
import java.lang.reflect.InvocationTargetException;

public class FileIO {
    File myFile;

    public FileIO(String path) {
        myFile = new File(path);
    }

    public boolean deleteFile() {
        return myFile.delete()
    }

    public File getFile() {
        return myFile;
    }

    public void setFile(String path) {
        myFile = new File(path);
    }

    public void createFile() throws InvocationTargetException{
        try {
            File dirFile = myFile.getParentFile();
            if (dirFile != null) {
                dirFile.mkdirs();
            }
            myFile.createNewFile();
        }
        catch(IOException e) {
            throw new InvocationTargetException(e);
        }
    }

    public static void main(String [] args) {
        String path = "filetest.txt";
        int exitCode = 0;
        FileIO fileTest = new FileIO(path);

        try {
            fileTest.createFile();
        } catch (InvocationTargetException e) {
            e.getCause().printStackTrace();
            exitCode = 1;
        } catch (Exception e) {
            e.printStackTrace();
            exitCode = 2;
        }
        ~~/*~~finally {
            System.exit(exitCode);
        }~~*/~~
    }
}
```

We added another **catch** clause to catch any other exceptions (**NullPointerException**, for example) that might occur and to print out the stack trace. We also set the **exitCode** to another non-zero value. By adding these small changes, we've made our class more robust, enabling it to handle errors that we can foresee.

Before running the program, look in the package explorer and remove all files named **filetest.txt** from your project.

Save and Run the program. Right-click on the project in the package explorer and select **Refresh** to see that the **filetest.txt** file now exists in the project, but is not present in the **myFile** directory.

Note If a file already exists, the **File** class **createNewFile()** method has no effect.

Writers and Readers

There are many ways to get your programs to read and write files. Here we'll use the Java API classes, **FileReader**, **FileWriter** and **PrintWriter**. These classes, in conjunction with **BufferedReader**, make handling of text files fairly straightforward.

Writing to A File

Modify the FileIO class by adding the **blue** code and removing the ~~red~~ code as shown:

CODE TO TYPE: FileIO.java

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.lang.reflect.InvocationTargetException;

public class FileIO {
    File myFile;

    public FileIO(String path) {
        myFile = new File(path);
    }

    public boolean deleteFile() {
        return myFile.delete();
    }

    public File getFile() {
        return myFile;
    }

    public void setFile(String path) {
        myFile = new File(path);
    }

    public void createFile() throws InvocationTargetException {
        try {
            File dirFile = myFile.getParentFile();
            if (dirFile != null) {
                dirFile.mkdirs();
            }
            myFile.createNewFile();
        } catch (IOException e) {
            throw new InvocationTargetException(e);
        }
    }

    public boolean printToFile(String text, boolean append, boolean autoFlush)
            throws InvocationTargetException {
        FileWriter fWriter;
        PrintWriter pWriter;
        boolean successFlag = true;
        try {
            fWriter = new FileWriter(myFile, append);
        } catch (IOException e) {
            throw new InvocationTargetException(e);
        }
        pWriter = new PrintWriter(fWriter, autoFlush);

        pWriter.println(text);
        if (pWriter.checkError()) {
            successFlag = false;
        }
        // The file streams should close and flush on method exit
        // but to be safe, always explicitly close():
        pWriter.close();

        return successFlag;
    }

    public static void main(String[] args) {
        final int NORMAL_EXIT = 0;
        final int FILE_CREATION_ERROR = 1;
        final int FILE_ERROR = 2;
        final int FILE_WRITE_ERROR = 3;
        String path = "filetest.txt";
        int exitCode = ~~0~~NORMAL_EXIT;
        FileIO fileTest = new FileIO(path);
        boolean append = true;
        boolean autoFlush = true;

        try {
            fileTest.createFile();
            for (int i = 1; i <= 10; i++) {
                if (!fileTest.printToFile("Line: " + i, append, autoFlush)) {
                    System.out.println("An error occurred writing to file: "
                            + fileTest.getFile().getPath());
                    exitCode = FILE_WRITE_ERROR;
                    break;
                }
            }
        } catch (InvocationTargetException e) {
            e.getCause().printStackTrace();
            exitCode = ~~1~~FILE_CREATION_ERROR;
        } catch (Exception e) {
            e.printStackTrace();
            exitCode = ~~2~~FILE_ERROR;
        } finally {
            System.exit(exitCode);
        }
    }
}
```

OBSERVE: FileIO.java

```java
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.lang.reflect.InvocationTargetException;

public class FileIO {
    File myFile;

    public FileIO(String path) {
        myFile = new File(path);
    }

    public boolean deleteFile() {
        return myFile.delete();
    }

    public File getFile() {
        return myFile;
    }

    public void setFile(String path) {
        myFile = new File(path);
    }

    public void createFile() throws InvocationTargetException {
        try {
            File dirFile = myFile.getParentFile();
            if (dirFile != null) {
                dirFile.mkdirs();
            }
            myFile.createNewFile();
        } catch (IOException e) {
            throw new InvocationTargetException(e);
        }
    }

    public boolean printToFile(String text, boolean append, boolean autoFlush)
            throws InvocationTargetException {
        FileWriter fWriter;
        PrintWriter pWriter;
        boolean successFlag = true;
        try {
            fWriter = new FileWriter(myFile, append);
        } catch (IOException e) {
            throw new InvocationTargetException(e);
        }
        pWriter = new PrintWriter(fWriter, autoFlush);

        pWriter.println(text);
        if (pWriter.checkError()) {
            successFlag = false;
        }
        // The file streams should close and flush on method exit
        // but to be safe, always explicitly close():
        pWriter.close();

        return successFlag;
    }

    public static void main(String[] args) {
        final int NORMAL_EXIT = 0;
        final int FILE_CREATION_ERROR = 1;
        final int FILE_ERROR = 2;
        final int FILE_WRITE_ERROR = 3;
        String path = "filetest.txt";
        int exitCode = NORMAL_EXIT;
        FileIO fileTest = new FileIO(path);
        boolean append = true;
        boolean autoFlush = true;

        try {
            fileTest.createFile();
            for (int i = 1; i <= 10; i++) {
                if (!fileTest.printToFile("Line: " + i, append, autoFlush)) {
                    System.out.println("An error occurred writing to file: "
                            + fileTest.getFile().getPath());
                    exitCode = FILE_WRITE_ERROR;
                    break;
                }
            }
        } catch (InvocationTargetException e) {
            e.getCause().printStackTrace();
            exitCode = FILE_CREATION_ERROR;
        } catch (Exception e) {
            e.printStackTrace();
            exitCode = FILE_ERROR;
        } finally {
            System.exit(exitCode);
        }

    }
}
```

Okay, let's break this thing down, starting with the **printToFile()** method. We are accepting three parameters in this method, **text**, **append**, and **autoFlush**. The **text** parameter is the text we want to write to the file. The **append** parameter lets us tell the method whether to append the text to the file or to replace the contents of the file with the **String text**. The **autoFlush** parameter tells the system if it should flush the data streams each time something is printed to the file.

In the **printToFile()** method, we create instances of **FileWriter** (**fWriter**)and **PrintWriter** (**pWriter**) which will allow us to write data to the file. We create the **fWriter** instance, passing it our file, **myFile**, and the **append** parameter, telling the **FileWriter** instance whether we want to append or overwrite the data in the file. We do this in a **try{}** block so that we can **catch()** any **IOExceptions** that occur.

The **PrintWriter** instance, **pWriter**, is created. We pass the **fWriter** object and the **autoFlush** parameter to **pWriter**. The **autoFlush** parameter, when true, tells the **pWriter** object to completely write data to the file, rather than letting it cache.

Now, we can write data to the file in exactly the same way we use **System.out** object. We **pWriter.println()** our **text** parameter to the file and then **close()** the file to make sure that all data streams are flushed and that the system resources used by the file are closed. This should happen automatically, when the method exits, but always do it manually. It's good practice.

Save and run the file. Right-click on the project and select Refresh in order to refresh the display of the project contents. Double-click on the **filetest.txt** file to display its contents in the editor. Close the file and run the program again. Refresh the project and then open the **filetest.txt** file again. Observe that the contents have been appended. Change the value of the **append** variable in **main()**, and run the file again. You'll see that only the last loop is saved in the file.

Reading a File

Reading a text file is a bit easier than writing one. Let's add to our code to see how it is done. Add the **blue** code as shown:

CODE TO TYPE: FileIO.java

```java
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.lang.reflect.InvocationTargetException;

public class FileIO {
    File myFile;

    public FileIO(String path) {
        myFile = new File(path);
    }

    public boolean deleteFile() {
        return myFile.delete();
    }

    public File getFile() {
        return myFile;
    }

    public void setFile(String path) {
        myFile = new File(path);
    }

    public void createFile() throws InvocationTargetException {
        try {
            File dirFile = myFile.getParentFile();
            if (dirFile != null) {
                dirFile.mkdirs();
            }
            myFile.createNewFile();
        } catch (IOException e) {
            throw new InvocationTargetException(e);
        }
    }

    public boolean printToFile(String text, boolean append, boolean autoFlush)
            throws InvocationTargetException {
        FileWriter fWriter;
        PrintWriter pWriter;
        boolean successFlag = true;
        try {
            fWriter = new FileWriter(myFile, append);
        } catch (IOException e) {
            throw new InvocationTargetException(e);
        }
        pWriter = new PrintWriter(fWriter, autoFlush);

        pWriter.println(text);
        if (pWriter.checkError()) {
            successFlag = false;
        }
        // The file streams should close and flush on method exit
        // but to be safe, always explicitly close():
        pWriter.close();

        return successFlag;
    }

    public String readFile() throws InvocationTargetException {
        FileReader fReader;
        BufferedReader bReader;
        String txtLine = "";
        String returnText = "";
        try {
            fReader = new FileReader(myFile);
            bReader = new BufferedReader(fReader);
            while ((txtLine = bReader.readLine()) != null) {
                returnText += txtLine + "\n";
            }
            return returnText;
        } catch (IOException e) {
            throw new InvocationTargetException(e);
        }
    }

    public static void main(String[] args) {
        final int NORMAL_EXIT = 0;
        final int FILE_CREATION_ERROR = 1;
        final int FILE_ERROR = 2;
        final int FILE_WRITE_ERROR = 3;

        String path = "filetest.txt";
        int exitCode = NORMAL_EXIT;
        FileIO fileTest = new FileIO(path);
        boolean append = true;
        boolean autoFlush = true;

        try {
            fileTest.createFile();
            for (int i = 1; i <= 10; i++) {
                if (!fileTest.printToFile("Line: " + i, append, autoFlush)) {
                    System.out.println("An error occurred writing to file: "
                            + fileTest.getFile().getPath());
                    exitCode = FILE_WRITE_ERROR;
                    break;
                }
            }
            System.out.println(fileTest.readFile());

        } catch (InvocationTargetException e) {
            e.getCause().printStackTrace();
            exitCode = FILE_CREATION_ERROR;
        } catch (Exception e) {
            e.printStackTrace();
            exitCode = FILE_ERROR;
        } finally {
            System.exit(exitCode);
        }

    }
}
```

OBSERVE: FileIO.java

```java
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.lang.reflect.InvocationTargetException;

public class FileIO {
    File myFile;

    public FileIO(String path) {
        myFile = new File(path);
    }

    public boolean deleteFile() {
        return myFile.delete();
    }

    public File getFile() {
        return myFile;
    }

    public void setFile(String path) {
        myFile = new File(path);
    }

    public void createFile() throws InvocationTargetException {
        try {
            File dirFile = myFile.getParentFile();
            if (dirFile != null) {
                dirFile.mkdirs();
            }
            myFile.createNewFile();
        } catch (IOException e) {
            throw new InvocationTargetException(e);
        }
    }

    public boolean printToFile(String text, boolean append, boolean autoFlush)
            throws InvocationTargetException {
        FileWriter fWriter;
        PrintWriter pWriter;
        boolean successFlag = true;
        try {
            fWriter = new FileWriter(myFile, append);
        } catch (IOException e) {
            throw new InvocationTargetException(e);
        }
        pWriter = new PrintWriter(fWriter, autoFlush);

        pWriter.println(text);
        if (pWriter.checkError()) {
            successFlag = false;
        }
        // The file streams should close and flush on method exit
        // but to be safe, always explicitly close():
        pWriter.close();

        return successFlag;
    }

    public String readFile() throws InvocationTargetException {
        FileReader fReader;
        BufferedReader bReader;
        String txtLine = "";
        String returnText = "";
        try {
            fReader = new FileReader(myFile);
            bReader = new BufferedReader(fReader);
            while ((txtLine = bReader.readLine()) != null) {
                returnText += txtLine + "\n";
            }
            return returnText;
        } catch (IOException e) {
            throw new InvocationTargetException(e);
        }
    }

    public static void main(String[] args) {
        final int NORMAL_EXIT = 0;
        final int FILE_CREATION_ERROR = 1;
        final int FILE_ERROR = 2;
        final int FILE_WRITE_ERROR = 3;

        String path = "filetest.txt";
        int exitCode = NORMAL_EXIT;
        FileIO fileTest = new FileIO(path);
        boolean append = true;
        boolean autoFlush = true;

        try {
            fileTest.createFile();
            for (int i = 1; i <= 10; i++) {
                if (!fileTest.printToFile("Line: " + i, append, autoFlush)) {
                    System.out.println("An error occurred writing to file: "
                            + fileTest.getFile().getPath());
                    exitCode = FILE_WRITE_ERROR;
                    break;
                }
            }
            System.out.println(fileTest.readFile());

        } catch (InvocationTargetException e) {
            e.getCause().printStackTrace();
            exitCode = FILE_CREATION_ERROR;
        } catch (Exception e) {
            e.printStackTrace();
            exitCode = FILE_ERROR;
        } finally {
            System.exit(exitCode);
        }

    }
}
```

In the **readFile()** method, we create a **FileReader** object, named **fReader**, and a **BufferedReader** object named **bReader**. A **BufferedReader** allows us to read more efficiently from files, because data from the file is stored in a buffer (a separate piece of RAM) and read there, instead of being read from the file for each read.

A really interesting part of the **readFile()** method is the **while** loop condition. In this condition, **txtLine**=**bReader**.**readLine()** assigns the next line of text from the text file to **txtLine**. **readLine()** fetches the next line of text, until it reaches the line terminator and returns it. It does not return the line terminator. Next, it compares **txtLine** to **null**. If **txtLine** is not **null**, the line of text has been read from the file. In the body of the loop, we concatenate that line of text to **returnText**.

You Are a Genius!

Well done! We've just covered one of the many ways there are to read data from a Java text file. There are more complex and powerful ways to read text from a file (beyond the scope of this course), but this is a great start.

Congratulations! You've stuck with it and completed our Java 3 course lessons. We're glad we had a chance to work with you in the course, and to help you to achieve your Java goals. Java 4 continues with more useful and empowering Java topics, including Swing GUI Building, Databases, and Multi-Threaded Programs. We hope to see you there.