Java SE 7 Fundamentals

Activity Guide

D67234GC10
Edition 1.0
September 2011
D74172

ORACLE

Authors

Jill Moritz, Kenneth Somerville, Cindy Church

Technical Contributors and Reviewers

Mike Williams, Tom McGinn, Matt Heimer, Joe Darcy, Brian Goetz, Alex Buckley, Adam Messenger, Steve Watts

This book was published using: Oracle Tutor

Table of Contents

Practices for Lesson 1: Java SE 7 Fundamentals

Chapter 1

Practices for Lesson 1

Practices Overview

There is no practice for Lesson 1.

Practices for Lesson 2: Introducing the Java Technology

Chapter 2

Practices for Lesson 2

Practices Overview

In these practices, you will run a Java program, first using the DOS command line, and then from the NetBeans integrated development environment (IDE).

Practice 2-1: Running a Java Program Using the Command Line

Overview

In this practice you compile and run a Java program at the command line. A Java technology program is already created for you. You must first set the PATH variable for the DOS session before running the program.

Assumptions

The Java SE 7 development environment is installed on your computer.

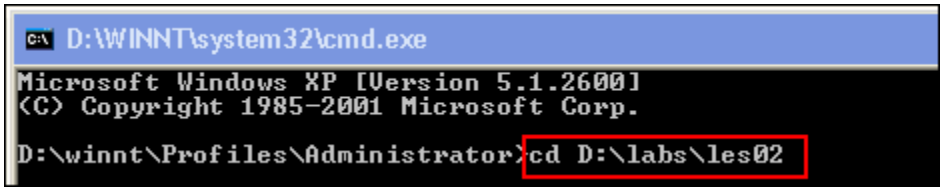
Task – Compiling and Executing a Java Program

In this task, you will compile and execute a Java Program.

1. Compile the CalcAverage.java program. The high level steps for this task are shown in the table below. If you need more assistance, you can use the detailed steps that follow the table.

Step	Description	Choices or Values
a.	Open a DOS command window and navigate to:	**D:\labs\les02**
b.	Check the contents of this directory listing to find:	**CalcAverage.java**
c.	Set the PATH variable to include:	**D:\Program Files\Java\jdk1.7.0\bin**
d.	Compile the CalcAverage java source file by typing:	**javac CalcAverage.java**

   a. From the Windows Start menu, select Start > Run. Enter **`cmd`** in the Open field and click **OK**. At the prompt, enter **`cd D:\labs\les02`**. Hit Enter.



   b. Check the directory contents by typing **`dir`** at the command prompt. Hit enter to see the results listed.

   c. Append the location of the Java executables (the compiler and the runtime executable) to the System PATH variable by entering the following at the command prompt. Hit enter.

```
D:\labs\les02>PATH = %PATH%;D:\Program Files\Java\jdk1.7.0\bin
```

You can confirm that the PATH was changed correctly by typing PATH at the next prompt. You should see the jdk1.7.0\bin appearing at the end of the PATH string.

```
D:\labs\les02>PATH
PATH=D:\Program Files\Java\jdk1.7.0\bin;d:\winnt\system32;d:\winnt;d:\winnt\syst
em32\wbem;c:\dos;c:\ntinst.ad;c:\utils;c:\detect;c:\net;D:\Program Files\Java\jd
k1.7.0\bin
```

d. Compile the .java file by typing `javac` CalcAverage.java. His enter. After a slight delay the prompt will return.

```
D:\labs\les02>javac CalcAverage.java
```

2. Run the CalcAverage.java program. The high level steps for this task are shown in the table below. If you need more assistance, you can use the detailed steps that follow the table.

Step	Window/Page Description	Choices or Values
a.	Confirm that the file was successfully compiled. List the directory content and look for:	**CalcAverage.class**
b.	Run the CalcAverage program. It will prompt you to enter three integers separated by spaces. Do so and hit enter to see the average of the three integers.	**java CalcAverage**

a. Look for the compiled class, `CalcAverage.class`, by listing the contents of the directory again. Type `dir` and hit enter.

```
D:\labs\les02>dir
 Volume in drive D is WINNT
 Volume Serial Number is FC5C-B059

 Directory of D:\labs\les02

06/08/2011  04:57 PM    <DIR>          .
06/08/2011  04:57 PM    <DIR>          ..
06/08/2011  04:57 PM               921 CalcAverage.class
04/22/2011  03:05 PM               641 CalcAverage.java
               2 File(s)          1,562 bytes
               2 Dir(s)  459,428,415,488 bytes free
```

b. Run the CalcAverage program by invoking the `java` runtime executable. You do not need to use the .class extension of the class. Type `java CalcAverage` and hit Enter. The program will prompt you to enter three integers.

```
D:\labs\les02>java CalcAverage

Enter 3 Integers separated only by spaces: (example 20 30 40)
```

Type three integers separated by spaces and then hit Enter.

```
D:\labs\les02>java CalcAverage

Enter 3 Integers separated only by spaces: (example 20 30 40)
2 46 88

Average = 45
```

This is how you would compile and run a Java program using only a DOS console or terminal window.

Practice 2-2: Running a Java Program Using NetBeans IDE

Overview

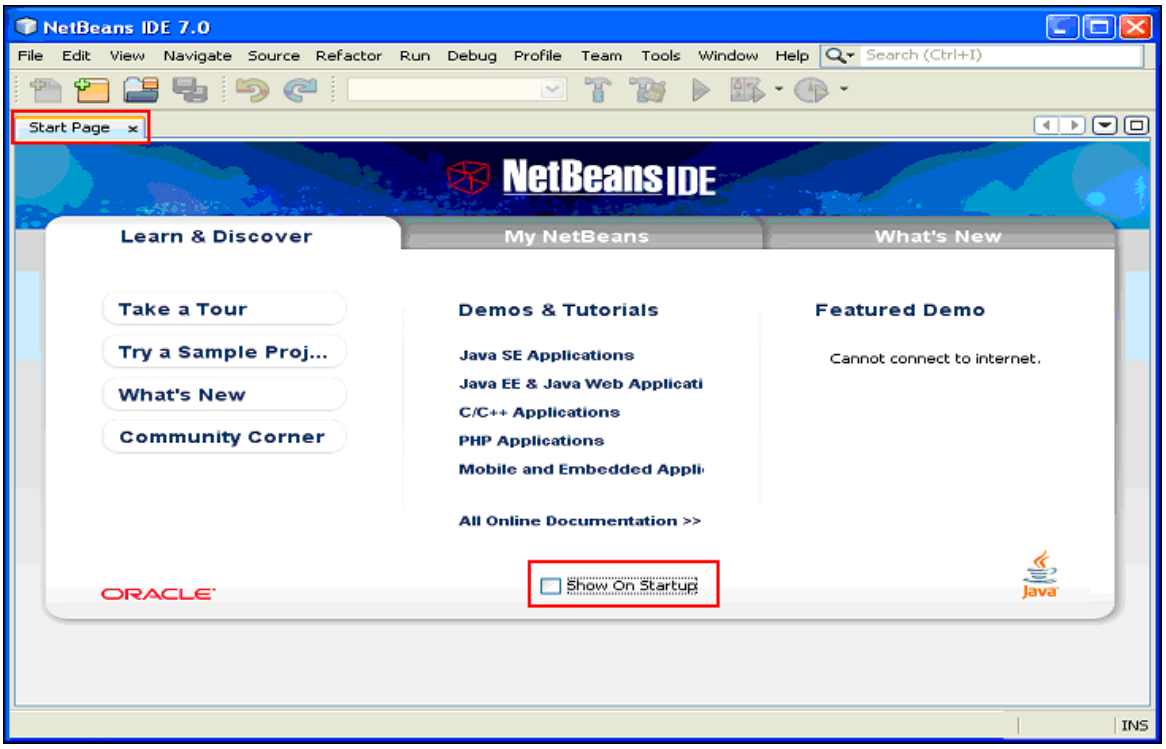
In this practice you compile and execute a Java program using NetBeans IDE. In addition, you explore some features of an IDE that let you develop programs more quickly and easily than if you use a command line.

Assumptions

The NetBeans 7.0 IDE is installed on your computer.

Tasks

1. Double-click the NetBeans icon from your computer desktop.
2. When NetBeans opens, deselect the **Show On Startup** checkbox and close the Start Page.



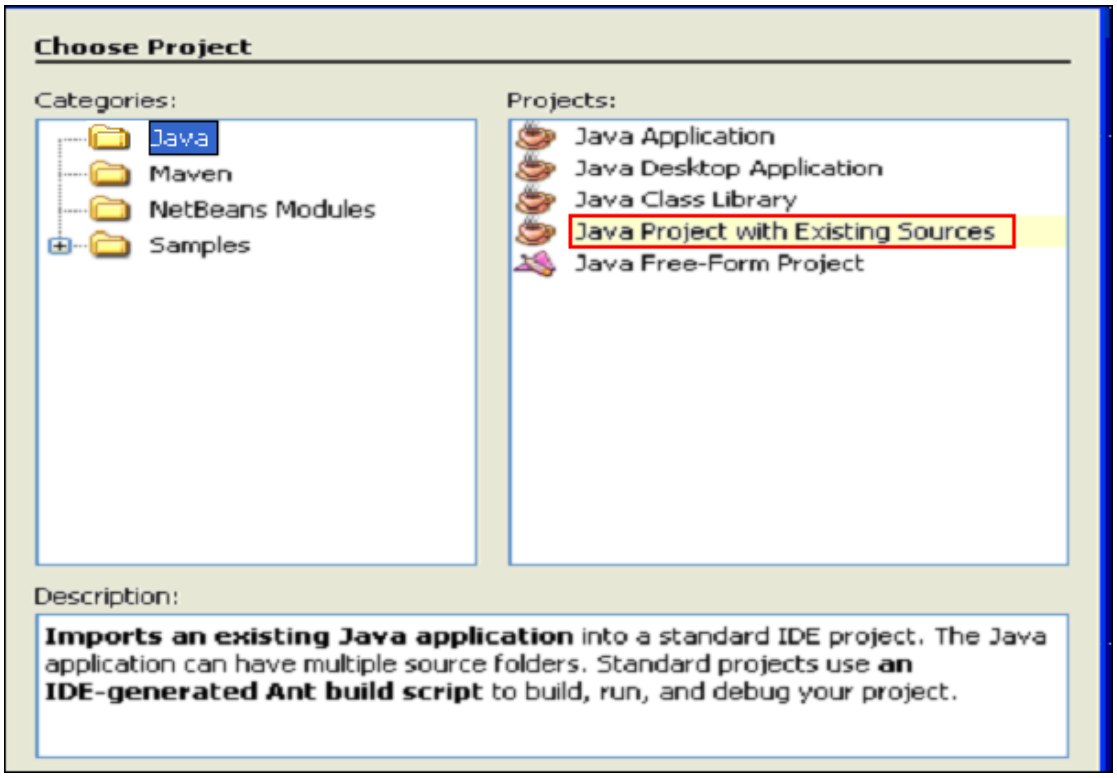
3. Create a NetBeans project that includes the CalcAverage.java file in its project source folder. The high level steps for this task are shown in the table below. If you need more assistance, you can use the detailed steps that follow the table.

Step	Window/Page Description	Choices or Values
a.	Main menu	**File > New Project …**
b.	New Project wizard: **Choose Project** step	Categories: Java Projects: Java project with existing source Click **Next**
c.	New Project wizard: **Name and Location** step	Project Name: Practice02 Deselect the Set as Main Project checkbox Click **Next**

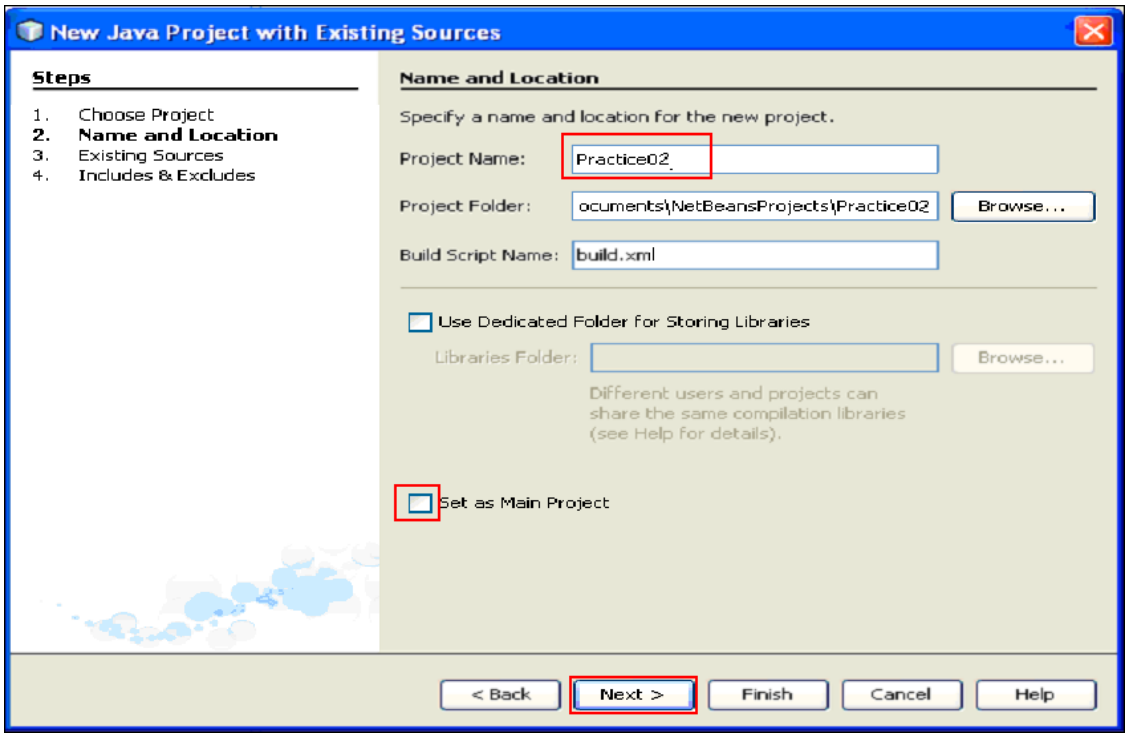
Step	Window/Page Description	Choices or Values
d.	New Project wizard: **Existing Sources** step	**Source Packages Folder**: Browse to select `D:\labs\les02` Click **Finish**
e.	Prompt window	**Delete** existing class files within the package folder. The new project appears in the Project's window of NetBeans.

a. Select **File > New Project** from the main NetBeans menu. The New Project wizard opens.

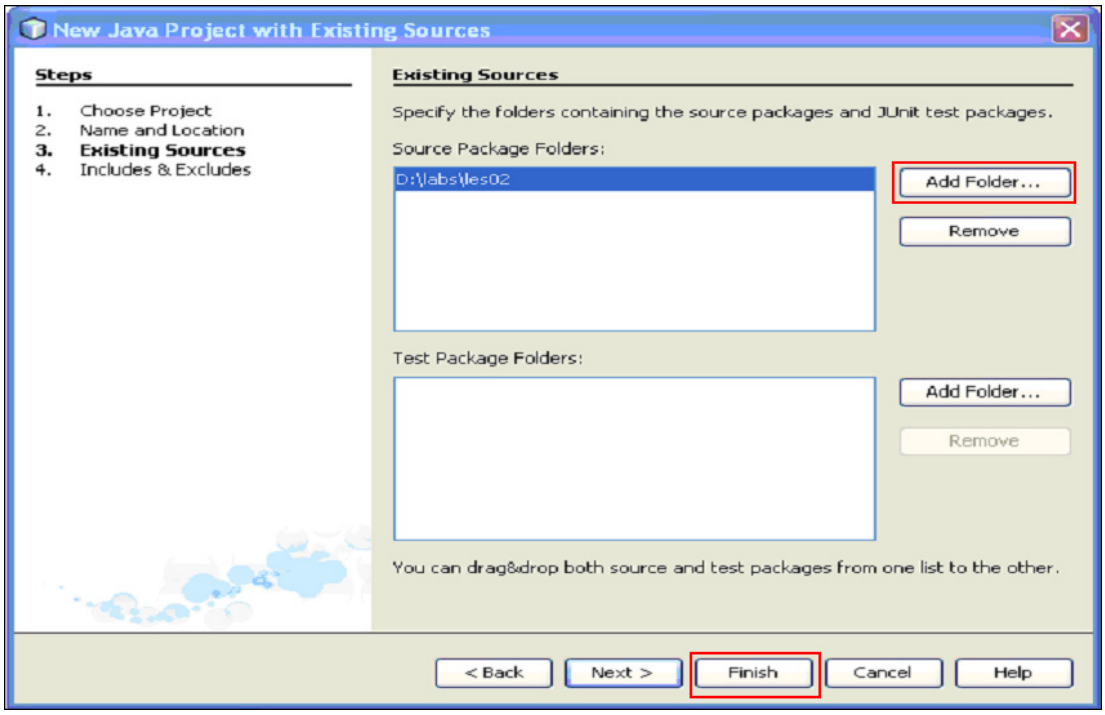
b. In the **Choose Project** step of the wizard (shown in the left column), select "Java" from the Categories column. Select "Java project with existing source" from the Projects column. Click **Next**.



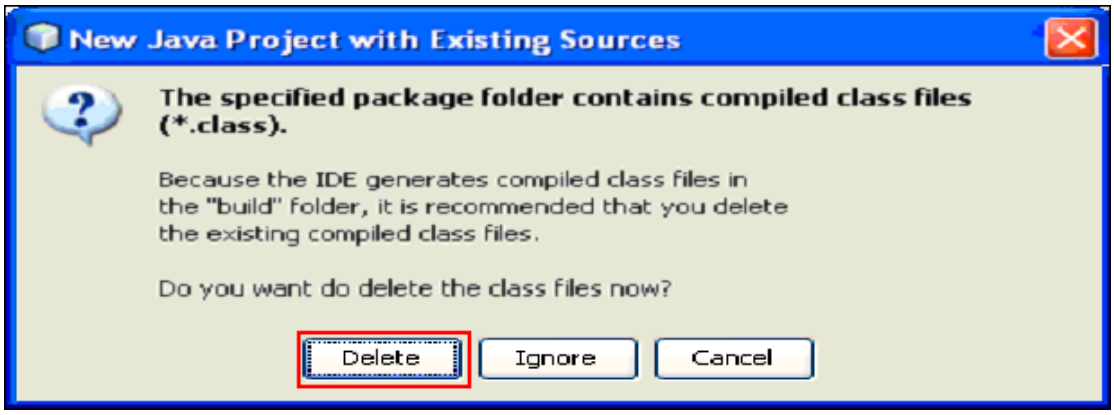
c. In the **Name and Location** step of the wizard, enter "Practice02" for the Project Name and deselect the **Set as Main Project** checkbox. Click **Next**.



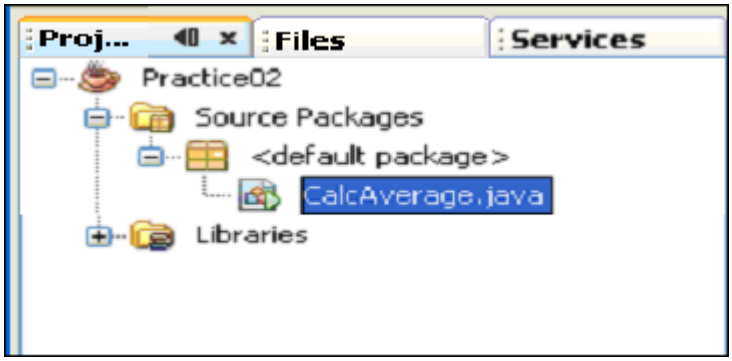
d. In the **Existing Sources** step of the wizard, add `D:\labs\les02` to the Source Packages Folder panel by clicking **Add Folder…** and browsing to the desired directory. Click **Finish**.



e. You are now prompted with the message "The specified package folders contain compiled class files" Click **Delete** to delete the CalcAverage.class file that was generated in the previous practice when you compiled the CalcAverage.java file from the DOS console. NetBeans will generate a new class file for you in this practice.



The contents of the project are now displayed in the **Projects** window within upper left pane of NetBeans. Select the Projects tab if necessary to view the Projects window. Here you see the project name at the root node. Expand the nodes beneath that to find CalcAverage.java.

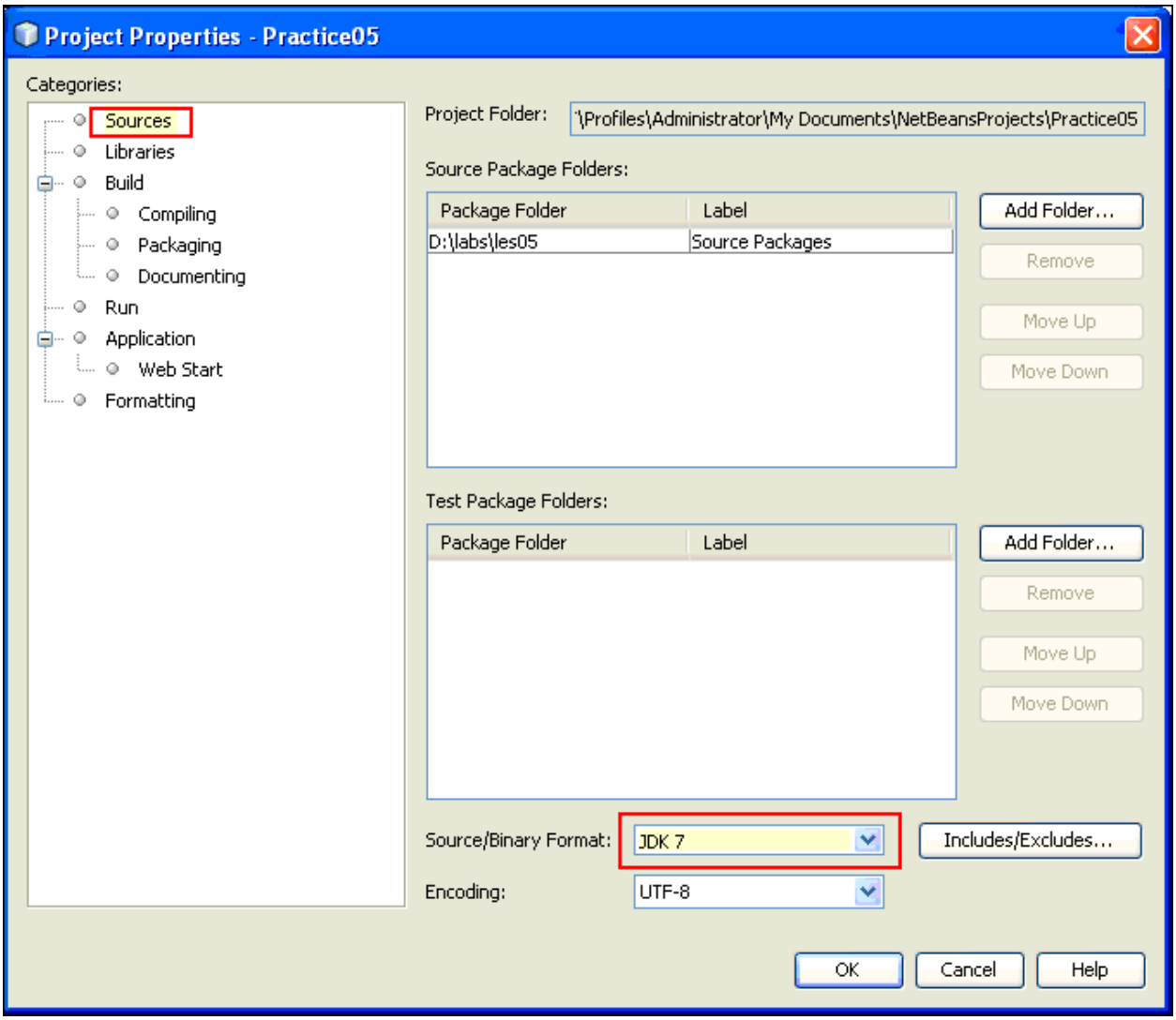


4. Modify the properties of this project to set the Source/Binary Format property to JDK 7. This will allow you to use any new language features of Java SE 7 without getting an error message from NetBeans. The table below provides the high level steps. If you need more details, follow the steps below the table.

Step	Window/Page Description	Choices or Values
a.	Main menu	**File > Project Properties (Practice02)**
b.	Project Properties window | Source category	Source/Binary Format field = **JDK 7**
c.	Project Properties window | Libraries category	Confirm that Java 7 is listed as the Java Platform
d.	Project properties window	Click **OK**

a. Select **File > Project Properties (Practice02)** from the main menu. (Alternatively, right click the **Practice02** project node in the Projects window and select **Properties**). The Project Properties window opens.

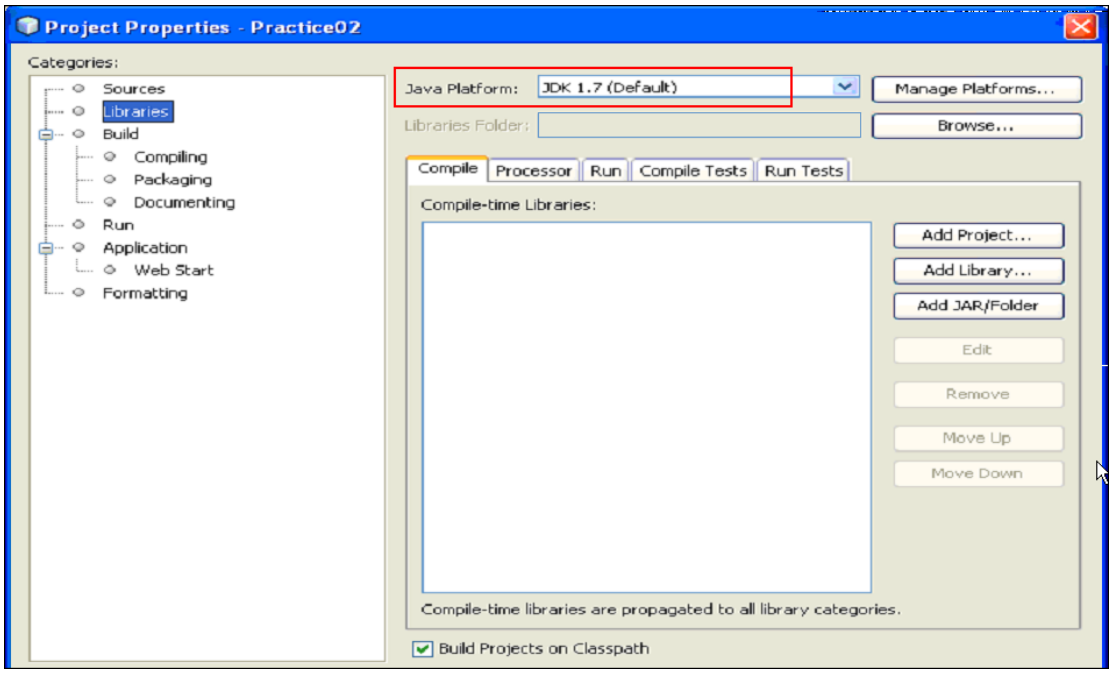
b. Select **Sources** in the Categories column. Set the **Source/Binary Format** field to "JDK 7".



Note: NetBeans allows you to specify the lowest Java platform version with which the generated code should be compatible. For instance, if you had not changed this setting to JDK 7, you would have seen error messages when using any of the core language changes included in JDK 7. NetBeans would warn you that the code would be incompatible with an earlier version.

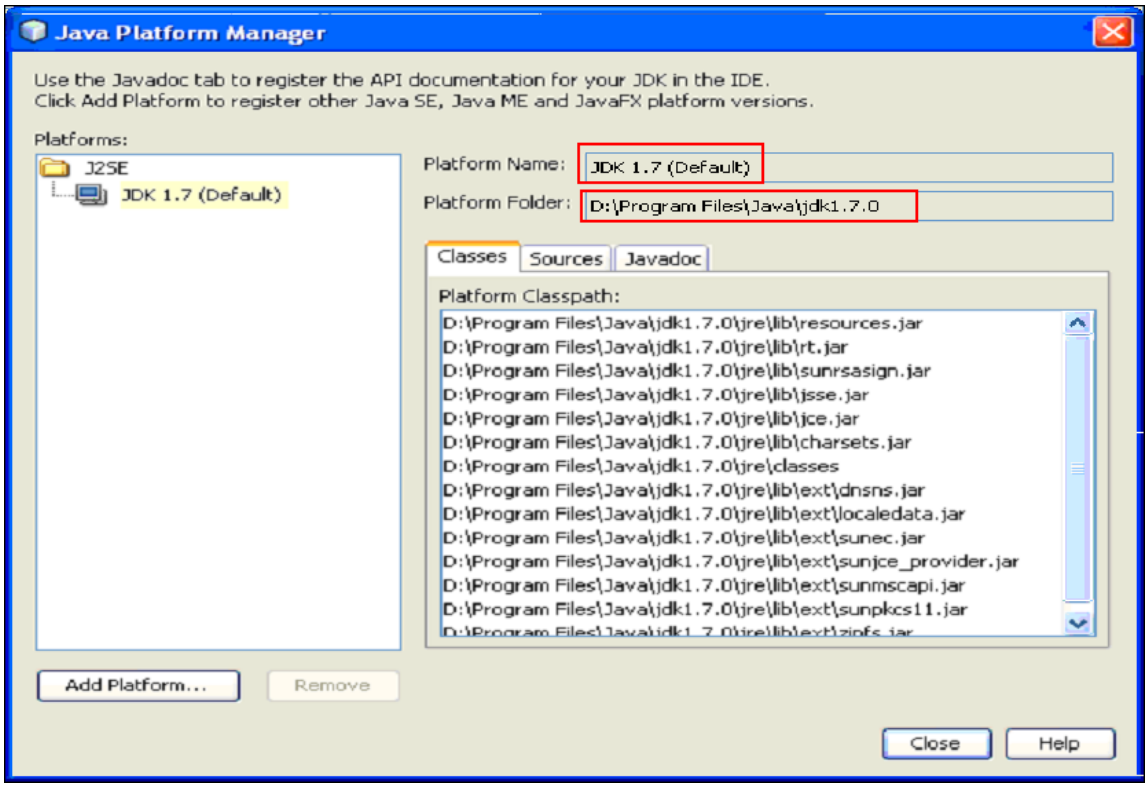
Remember that when you compiled and ran this java file from the command prompt, you had to manually set the PATH to point to the JDK 7 installation. When you use an IDE, it automatically sets a default JDK runtime environment for each NetBeans project.

c. Confirm that the Java Platform setting for the Practice02 project is **JDK 7**. Select the **Libraries** node in the Categories column. On the right, the JDK 7 is listed as the Java Platform for this project. Notice that you could select a different platform (JDK version) if you wished (assuming other platforms had been properly installed on this machine).

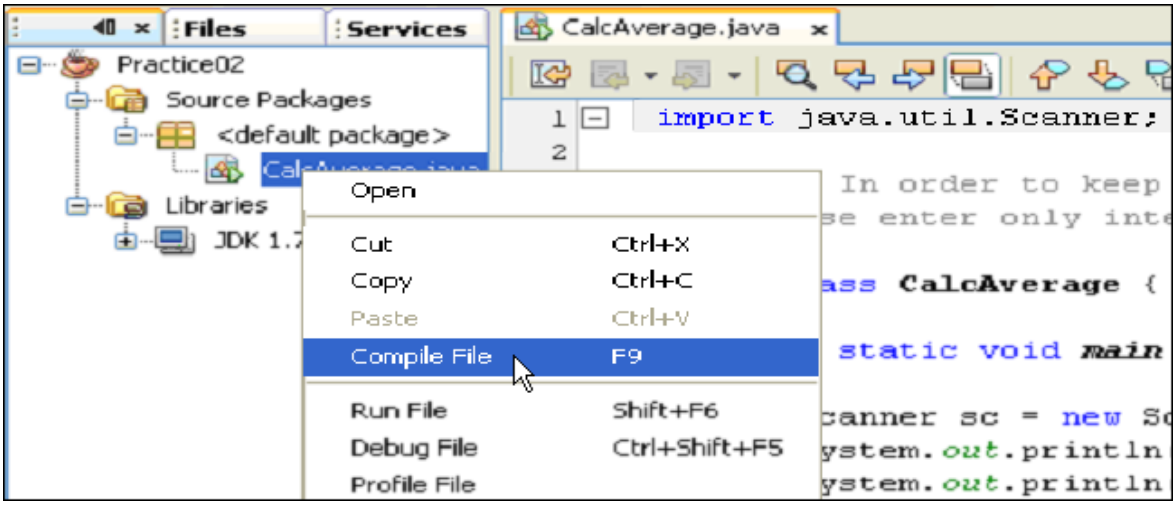


d. Click **OK** to save the change you made in step b to the project properties.

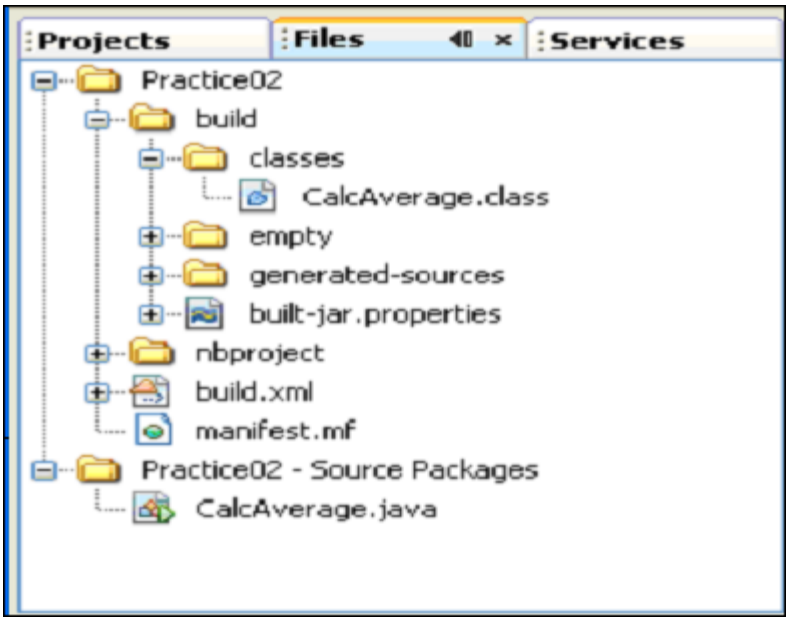
5. To determine or change the default Java Platform for NetBeans, select **Tools > Java Platforms** on the main menu. This window shows all versions of the JDK that have been properly installed on this machine. In our case, only the JDK 7 (a.k.a. JDK 1.7) has been installed so it is marked as the "Default" platform in the Platforms column. On the right, the directory location for the JDK 7 installation is shown. Close the Java Platform Manager window when you have finished examining it.



6. To view and edit the code for the CalcAverage.java file, double click it in the Project's window. It opens in the Editor pane. Notice the color coding used by the editor. (For example, keywords are in blue, string literals are in red.) This makes working with and reading you code much easier. You will learn more about using this editor in upcoming practices.
7. In the Projects window, right-click CalcAverage.java, and choose **Compile File**.

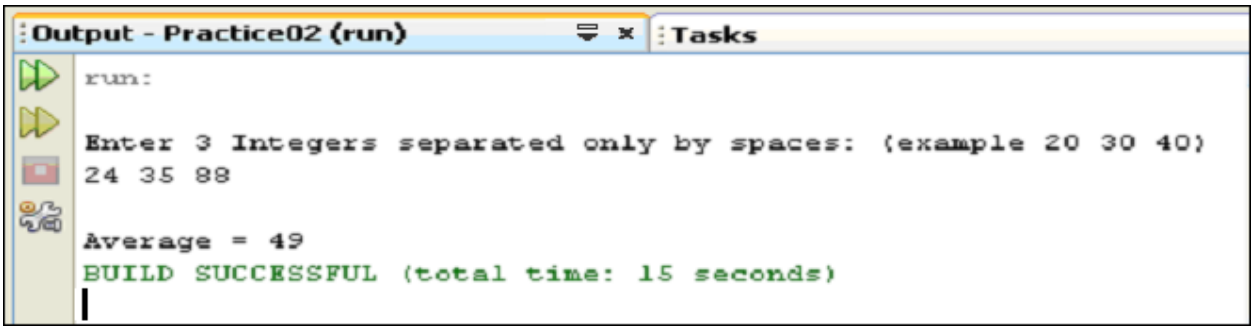


8. Assuming you had no compilation errors, you can now find the .class file by clicking the Files window and expanding **Practice02 > build > classes**.



Note: If you had made any changes to the java file, the Save button would have become enabled. By default, compilation occurs automatically with a Save.

9. Click the Projects window again. Right-click the file and choose **Run File**. The output from the program appears in the Output window. Enter the three integer values in the line beneath the output message and hit Enter to see the result.



Now you have seen how to run a simple Java program using both the DOS command prompt and the NetBeans IDE.

10. Close the Practice02 project in NetBeans. In the Projects window, right click Practice02 and select **Close** from the context menu.

Practices for Lesson 3: Thinking in Objects

Chapter 3

Practices for Lesson 3

Practices Overview

In these practices, you will first analyze a problem using object-oriented analysis, and then you will design a possible solution by using UML-like notation. Solutions for these practices can be found in `D:\labs\soln\les03`.

Practice 3-1: Analyzing a Problem Using Object-oriented Analysis

Overview

In this practice you analyze a case study and use object-oriented analysis to list the objects, attributes, and operations in the case study.

Preparation

Read the following case study, and then model the system by choosing objects and their attributes and operations.

Case Study

A soccer league needs a system to track team and player standings.

At any moment they want to be able to report a list of games played with results, a list of teams ranked by wins, and a list of players on each team ranked by goals scored.

Tasks

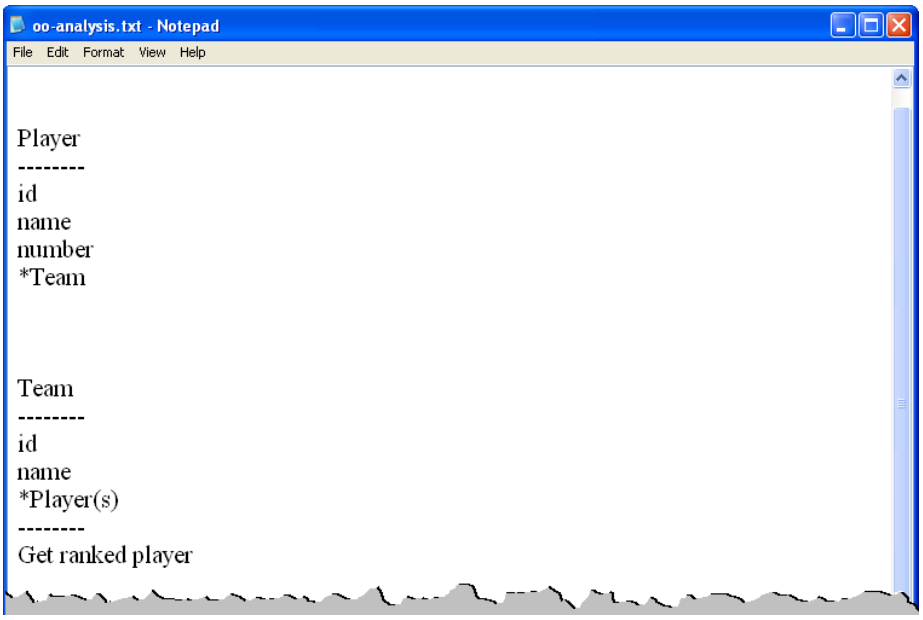
Your task is to produce an object-oriented analysis for a Java technology application that tracks soccer scores. The program should track:

- The list of *players* on each *team* ranked by *goals* scored
- The list of *games* played with results
- The list of teams in the *league* ranked by wins

Hint: You can think of the objects as nouns, attributes as adjectives, and operations as verbs. As an example, a Player is a noun, the player's name is an adjective that describes that noun, and add goal is a verb.

The application should be able to generate statistics for teams, players, and seasons.

1. Open the text editor by clicking Start > Programs > Accessories > Notepad.
2. Save the file as `D:\labs\les03\oo-analysis.txt`.
3. To get started, list the high-level classes that are included in this problem. You can list them in the text editor and use dashed lines to separate the objects, attributes, and operations as shown in the screenshot.



4. (Optional) You can use the UMLet tool if you choose. Double-click the UMLet icon 

from the Windows desktop to launch the program.

Solution

Player	Team	Game	League	Goals
id name number *Team	id name *Player(s)	id team one score team two score *Goals	*Team(s) *Game(s)	id *Team *Player time
	Get ranked player	Get results	Get game results Get ranked teams	

The asterisk (*) denotes attributes that are also objects.

Practice 3-2: Designing a Programming Solution

Overview

In this practice you continue with Practice 3-1 by using UML-like notation to represent the classes you identified.

Assumptions

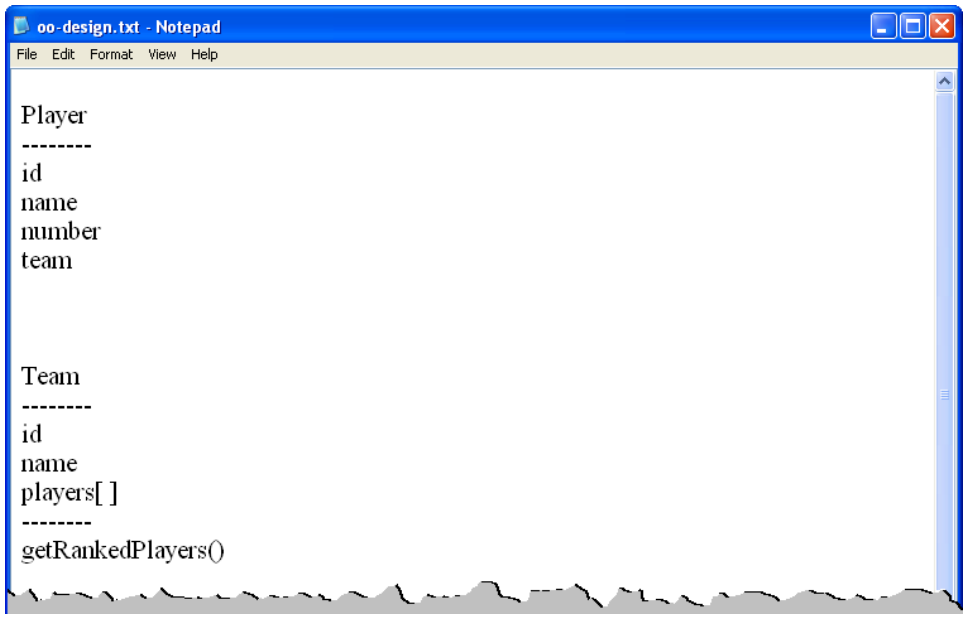
The assumptions are you have completed identifying the objects, attributes, and operations that you found in Practice 3-1.

Tasks

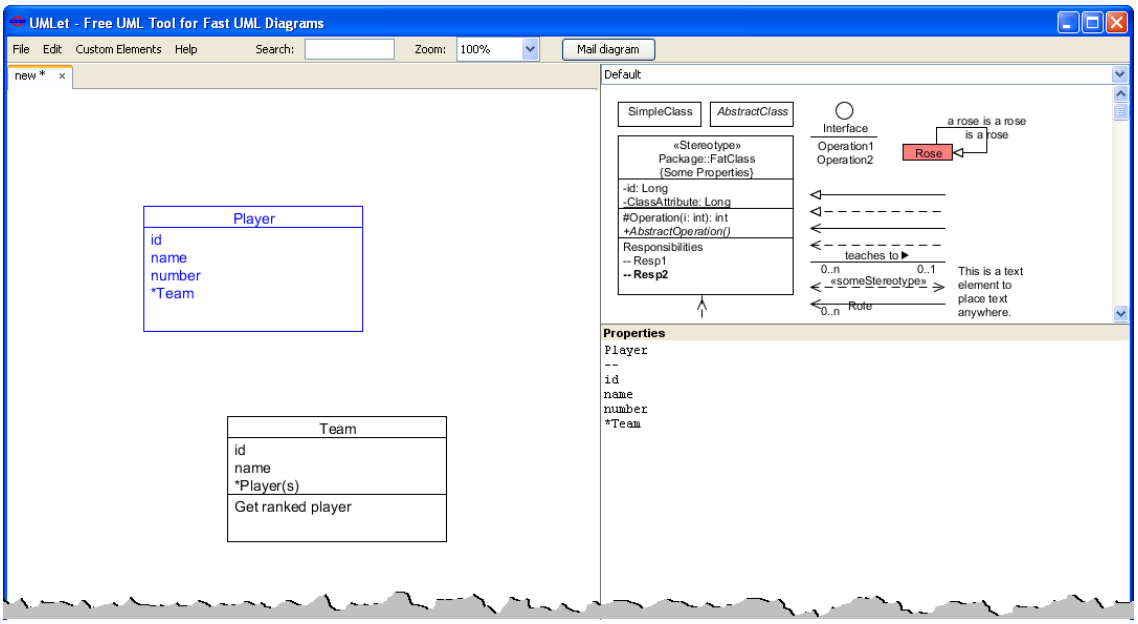
Your task is to produce a design for each of the classes in the earlier system for tracking soccer scores. Remember to:

- Use camelCase to name your classes, attribute variables, and methods
- Identify a valid range of values for each attribute (where a range is known)
- Use square brackets to indicate an attribute that represents a collection of values (players[ ])
- Use parentheses to identify methods

1. Open `D:\labs\les03\oo-analysis.txt` and save it as `D:\labs\les03\oo-design.txt`.
2. Use the classes, variables, and operations that you identified in the previous practice, and develop method names for the operations. The screenshot below is an example.



3. (Optional) You can use the UMLet tool if you choose. Double-click the UMLet icon  from the Windows desktop to launch the program.



Solution

Player	Team	Game	League	Goals
id name number team	id name players[ ]	id team one score team two score goals[ ]	teams[ ] games[ ]	id team player time
	getRankedPlayers()	getResults()	getGameResults() getRankedTeams()	

Note: Although not shown in the solution, you will need add/remove methods for each collection attribute and get/set methods for all other attributes. We have not discussed those methods at this point in the course.

Your solutions might look different from the suggested solution. The purpose of this lesson is to help you continue thinking in terms of objects, attributes, and operations. You will have another opportunity during this course to practice modeling a programming solution.

Practices for Lesson 4: Introducing the Java Language

Chapter 4

Practices for Lesson 4

Practices Overview

In these practices you examine and modify existing Java programs and also run them to test the program. Solutions for these practices can be found in `D:\labs\soln\les04`.

Practice 4-1: View and Add Code to an Existing Java Program

Overview

In this practice you are given a completed Java program. You will open it, examine the lines of code, modify it, compile it, and then test it by executing the program.

Assumptions

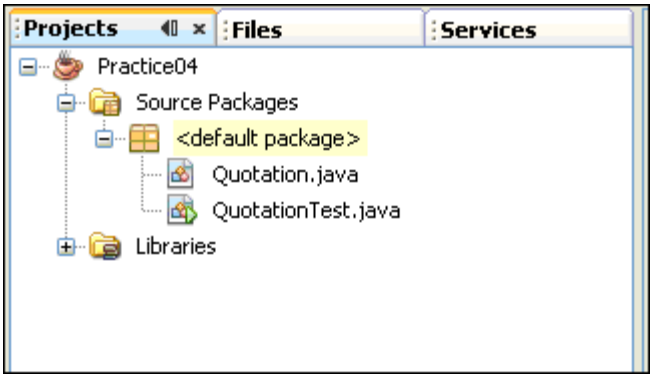
- Quotation.java and QuotationTest.java appear in the lab folder for this practice: `D:\labs\les04`

Tasks

1. Create a new project from existing Java source, just as you did in Practice 2-2. The high level steps are shown in the table below. If you need further detail, please refer to Practice 2-2, steps 3 and 4.

Step	Window/Page Description	Choices or Values
a.	Menu	File > New Project
b.	New Project wizard | Choose Project step	Category: Java Project: Java Project with Existing Sources **Next**
c.	New Project with Existing Sources wizard | Name and Location step	Project Name: Practice04 **Next**
d.	New Project with Existing Sources wizard | Existing Sources step	**Add Folder**: `D:\labs\les04` **Finish**
e.	Project Properties window | Source category	**Source/Binary Format**: JDK 7 **OK**

Note: The Projects window should now look like this when the **<default package>** node is expanded:



2. Double click the `Quotation.java` file in the Projects window to open it for editing.

3. Identify the field and the method contained within this class, using the table below:

Member	Variable or Name
Field variable:	
Method name:	

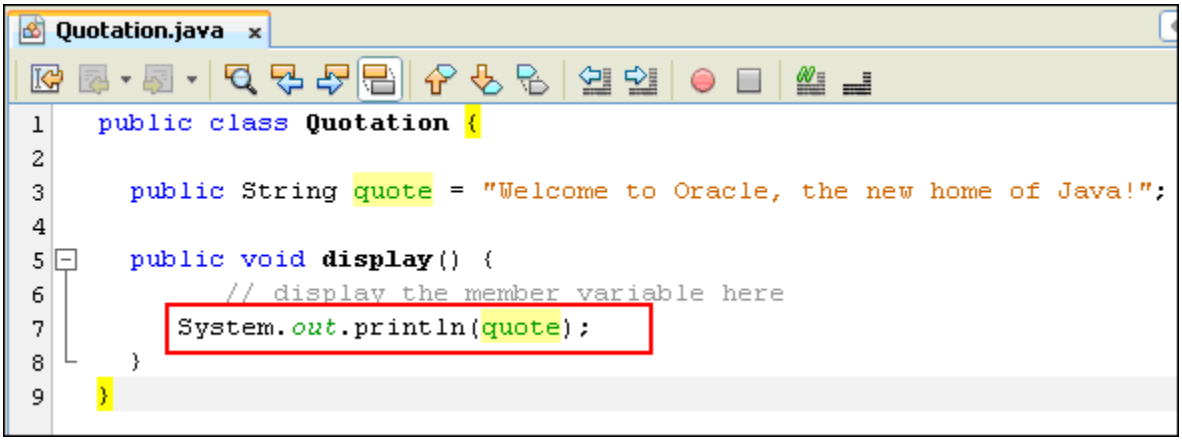
Solution: Field variable – `quote`; Method name – `display`.

4. In the display method, write the code to display the quote field. *Hint:* Use the `System.out.println` method shown in the Student Guide for this lesson. Be sure to finish the line of code with a semicolon.

   Note: You will notice, as you type the code, that NetBeans' code assist feature provides feedback and help whenever you pause in your typing. For instance, if you stop at some point at which the code, as is, would not compile successfully, it displays a red exclamation mark in the left margin. If you pause after typing the dot (".") following `System` or `out`, it gives you context sensitive help in the form of a list of methods and fields that would be valid for the particular class to the left of the dot. You can select from the list instead of typing.

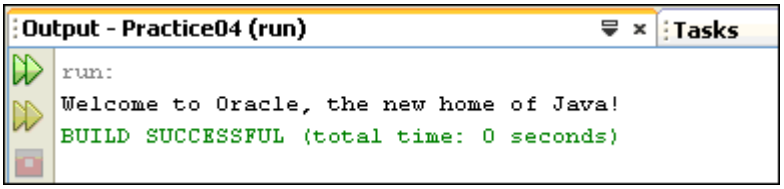
Solution:

```
System.out.println(quote);
```



5. Click the Save button to save and compile Quotation.java.
6. Open the QuotationTest.java file in the editor and examine its `main` method. It creates an instance of the Quotation class and then calls its `display` method.
7. Run the `QuotationTest` class by right clicking QuotationTest.java in the Projects window and selecting **Run File**. The output from the display method appears in the Output window.

   Note: You were able to skip the Compile step because when you select Run File, NetBeans first compiles not only the class you selected to run, but also any referenced classes within that class (`Quotation.java`).



8. Edit the `Quotation.java` file now to change the default value of the `quote` field.

9. Run `QuotationTest` again to verify the output.
10. In the Editor pane, close `Quotation.java` and `QuotationTest.java`.

Practice 4-2: Create and Compile a Java Class

Overview

In this practice you create a Java class and compile it. You also create another Java class to test the previous class.

Assumptions

None

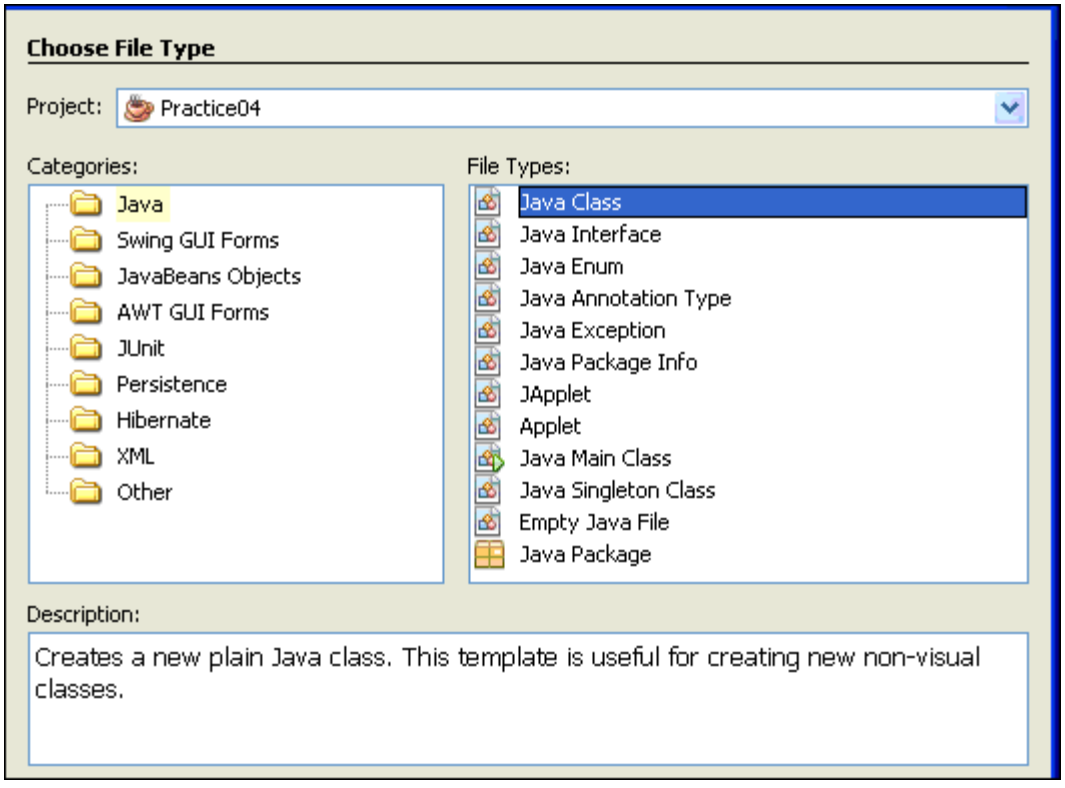
Tasks

1. Create a new Java class in the Practice04 project using the NetBeans wizard. The high level steps for this task are shown in the table below. If you need more assistance, you can use the detailed steps that follow the table.

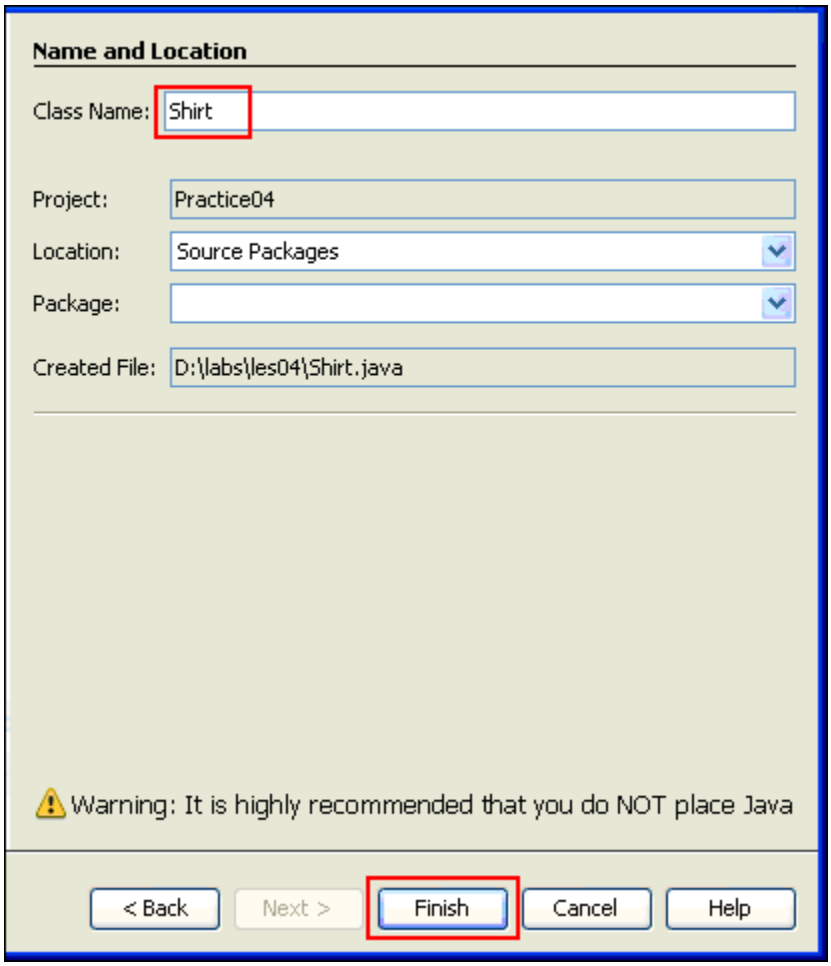
Step	Window/Page Description	Choices or Values
a.	Menu	**File > New File**
b.	New File window | Choose File Type step	Category: Java File Types: Java class **Next**
c.	New Java Class window | Name and Location step	Class Name: `Shirt` **Finish**

   a. From the main menu, select **File > New File**.

   b. The New File wizard opens and you are on step 1 "Choose File Type". Select **Java** in the Category column. Select **Java class** in the File Types column. Click **Next**.



c. In the New Java Class window you are on step 2 “Name and Location”. Enter “Shirt” as the Class Name. Click **Finish**.



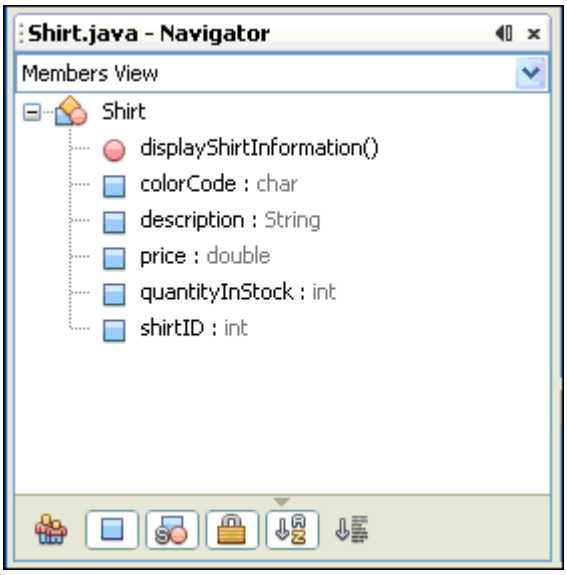
The Java source file for the new class now appears in the editor ready for you to fill in the details.

2. Enter the Java code syntax for the Shirt class shown in this lesson of the Student Guide.

Solution: You can find the solution code for the Shirt class in `D:\labs\soln\les04`

3. Click the Save button to save and compile the Shirt class. Any red error icons in the left margin should disappear after saving if there were no compilation errors. If necessary, fix any errors that appear in the Output window and save again.

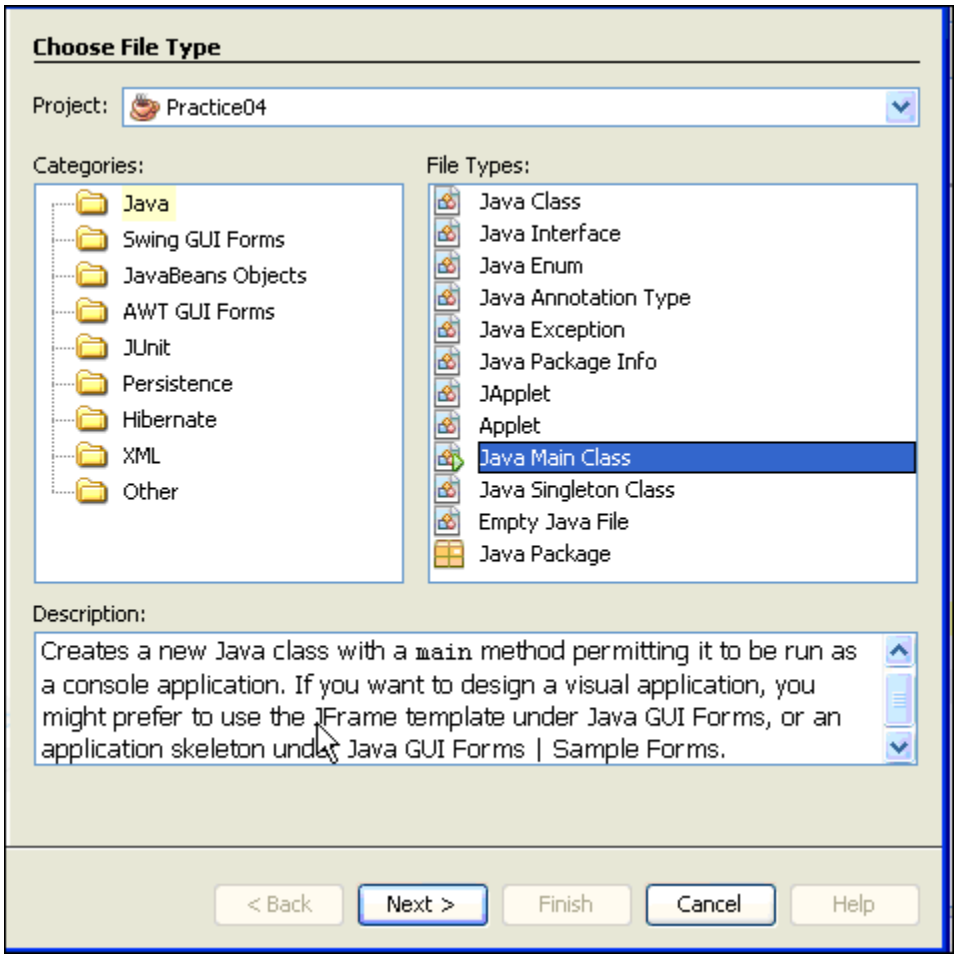
Note: The Navigator pane (lower left corner of NetBeans) for the `Shirt` class now shows the Members view of the class. Notice the color coding that distinguishes between fields and methods. Both of these are considered “Members” of the class.



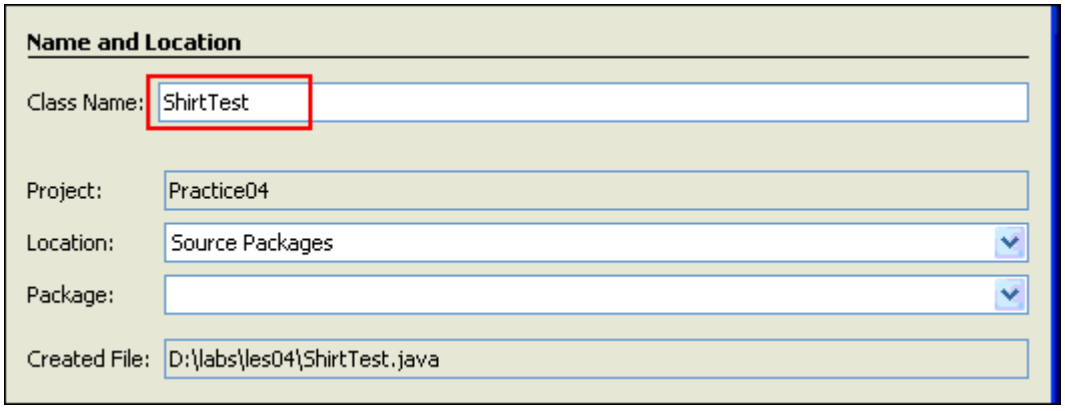
4. Follow the instructions from Step 1 to create another new class. This will be a Test class, so it will need a `main` method. To accommodate that change, the table below shows the substitutions in the Step 1 instructions you should make as you go through the New Class wizard. For more detail, see the screenshots following the table.

Step	Window/Page Description	Choices or Values
a.	New File window | Choose File Type step	File Types: **Java Main Class**
b.	New File window | Name and Location step	Name: **ShirtTest**

a. In the Choose File Type step, select **Java Main Class** instead of Java Class.



b. In the Name and Location step, enter **ShirtTest** as the name.

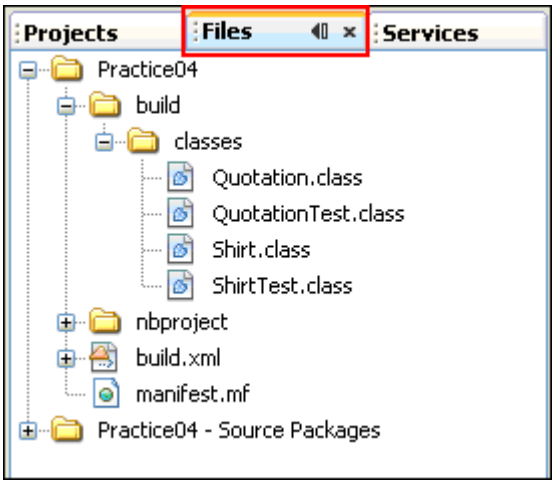


5. Replace the **To Do:** comment in the `main` method with the two lines of code that appear in the `main` method for the `ShirtTest` class shown in this lesson of the Student Guide.

Solution: You can find the solution code for the `ShirtTest` class in `D:\labs\soln\les04`

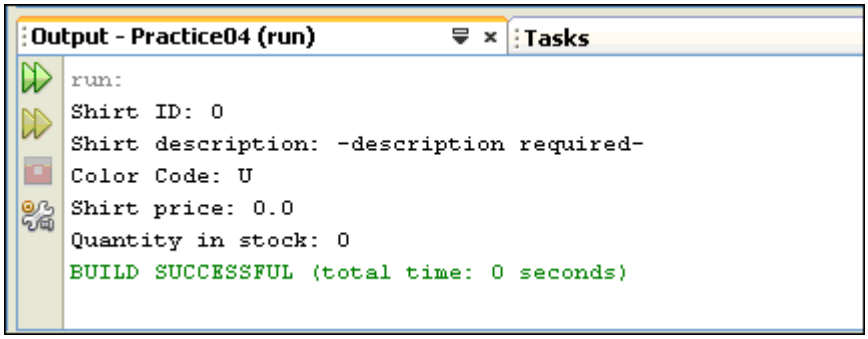
```
10  public class ShirtTest {
11      /**
12       * @param args the command line arguments
13       */
14      public static void main(String[] args) {
15          Shirt myShirt;
16          myShirt = new Shirt();
17          myShirt.displayShirtInformation();
18      }
19  }
```

6. Save and compile the code by clicking Save.
7. Confirm that the `Shirt` and `ShirtTest` classes have been successfully compiled. Click the Files tab in the upper left pane of NetBeans to open the Files window. Expand **Practice04\build\classes**. You should see `Shirt.class` and `ShirtTest.class` within the `classes` folder as shown below.



8. Return to the Projects window and run the `ShirtTest` class. Look for the output of the `displayShirtInformation` method in the Output window.

Solution: Right click `ShirtTest.java` and select **Run File** from the context menu.



9. Open (or return focus to) the `Shirt.java` file. Modify the values of `ShirtID` and `price`.
10. Run the `ShirtTest` class again. Verify that the modified values are shown in the Output window.

Practice 4-3: Exploring the Debugger

Overview

Virtually every Java IDE provides a debugger. They tend to offer the same core features and work very similarly. In this practice you debug the ShirtTest program using the NetBeans debugger. You set breakpoints, examine field values, and modify them as you step through each line of code.

Assumptions

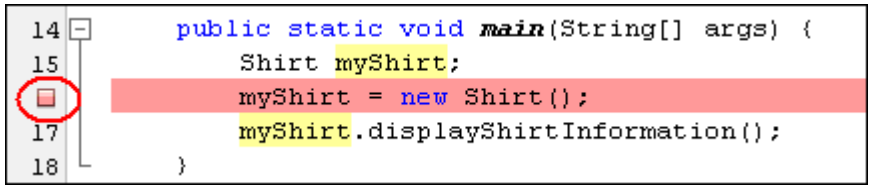
None

Tasks

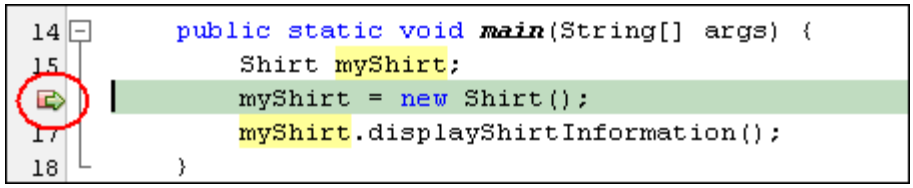
1. Set a breakpoint in the `ShirtTest` class. Click in the left margin of the editor, next to the following line of code:

   ```
   myShirt = new Shirt();
   ```

   A pink square appears in the margin, indicating a breakpoint.



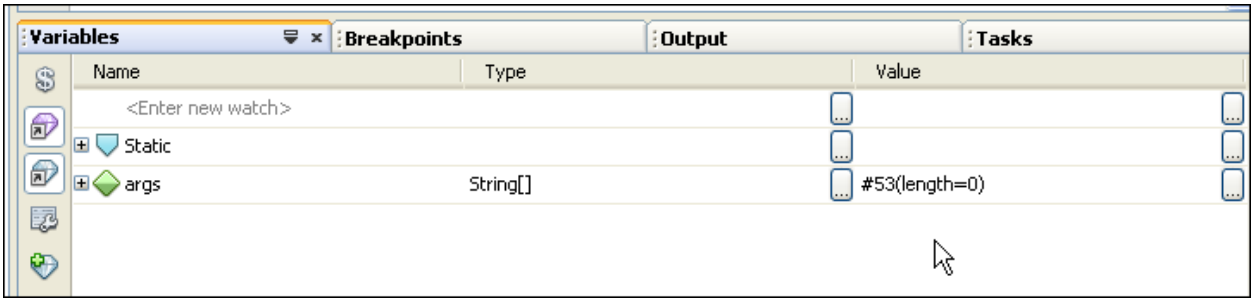
2. Run the debugger by right clicking on the `ShirtTest` file in the Projects window and selecting **Debug File**.
3. The debugger starts the program and stops at the breakpoint. In the Editor panel you should now see a different icon that points with a green arrow to the line of code.



   This line of code has not yet been executed.
4. Several other changes have occurred in the NetBeans window.
   - A new toolbar appears, containing buttons that you use when debugging.

     – Move your cursor over each of the toolbar buttons to read the toolbar tip explaining what each button does. The buttons are described below.
       – The first button, **Stop**, stops the debugging session.
       – The second button, **Pause**, pauses the execution of the debugger.
       – The third button **Continue** the execution, either to the next breakpoint or to the end of the program.
       – The fourth button, **Step Over**, moves the program forward to the next line of code in the current class (in this case, the ShirtTest class).
       – The fifth button, **Step Over Expression**, allows you to step over an entire expression to the next line of code in the current class.

  - The sixth button, **Step Into**, allows you to step into another class referenced in this current line of code.
  - The seventh button, **Step Out**, allows you to step back out of a class that you stepped into.
  - The last button, **Run to Cursor**, takes execution to the line of code where the cursor appears.
- The panel at the bottom of the window changes to show debugging output and variables and other useful information during a debug session.



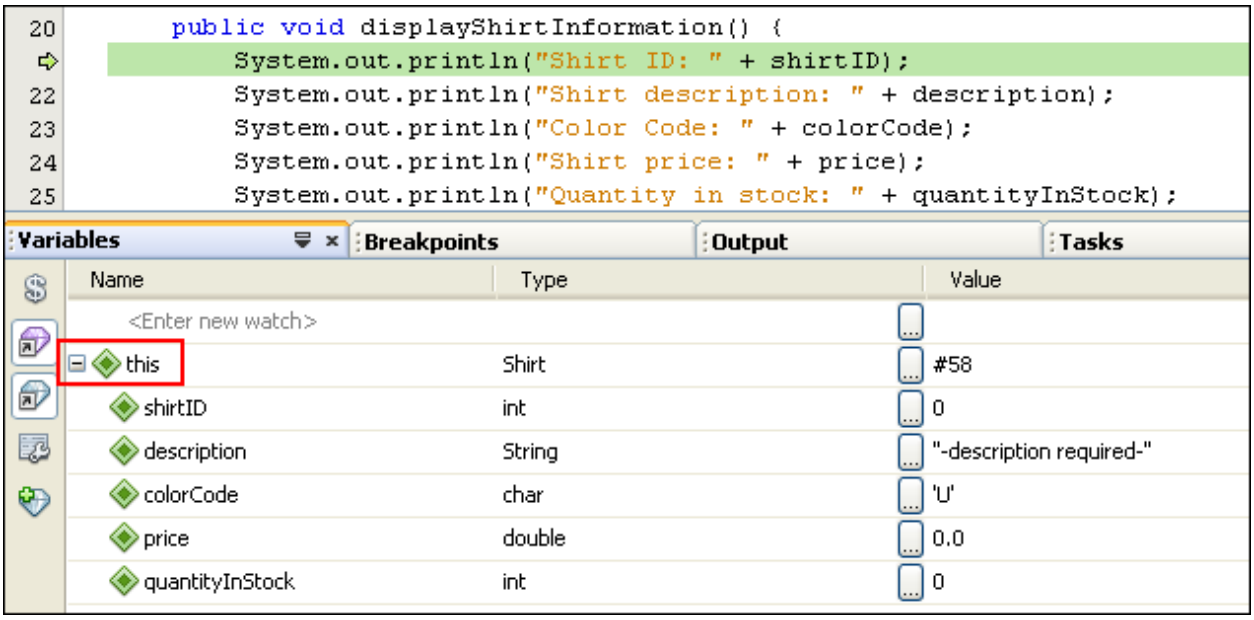
  - In the Variables panel, you see all variables that are visible to the current class. Remember that the execution was stopped *before* the `Shirt` class object has been instantiated. Consequently, you do not see the `myShirt` variable in this panel.

5. Click the Step Over button to move to the next line of code. 
6. The arrow now points to the line of code that calls the `displayShirtInformation` method on the `myShirt` object. In the Values window, you now see the `myShirt` variable. Expand it to see all of the fields of this `Shirt` object.

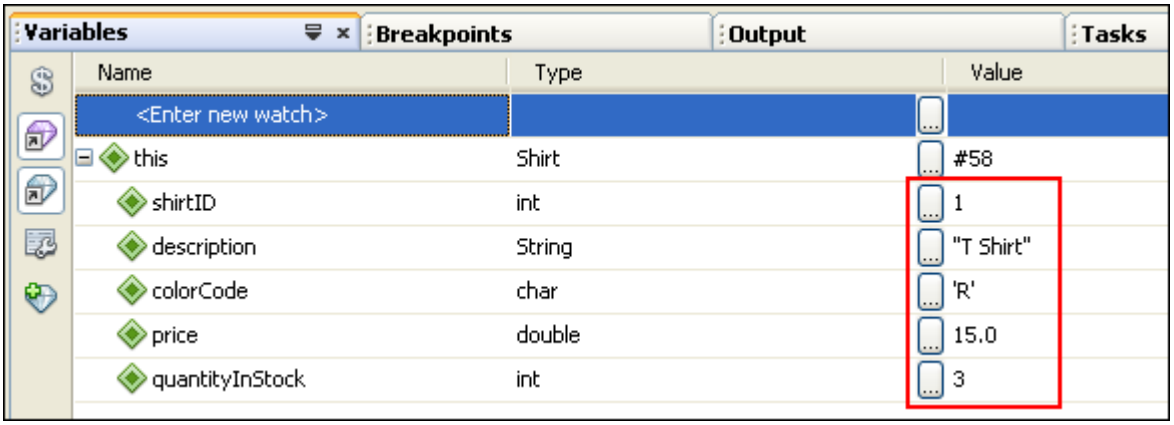
```
14      public static void main(String[] args) {
15          Shirt myShirt;
            myShirt = new Shirt();
            myShirt.displayShirtInformation();
18      }
```

At this point, the `displayShirtInformation` method has not yet been executed. You could change the values of the object's fields right now, using the Variables window if you wanted to. However, instead, you will "step into" the myShirt object and change the values during the execution of the `displayShirtInformation` method..

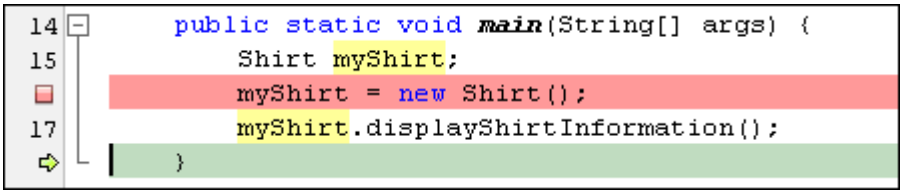
7. Click the Step Into button to step into the `displayShirtInformation` method. 
8. The arrow icon is pointing to the first executable line of code within the `displayShirtInformation` of the `Shirt` class. In the Variables window, expand **this** to see the fields of this object.



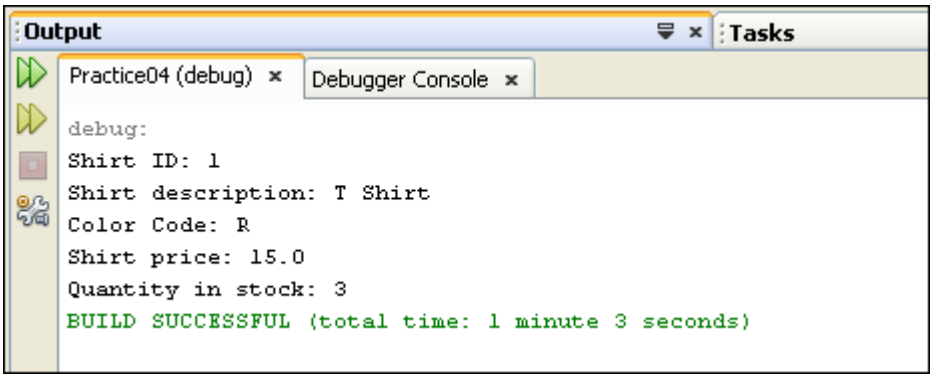
9. In the Value column click on each field's value and edit it to change the value. Ensure that you use the correct value for the data type expected and enclose any character data types with the type of quote mark indicated. After editing the final field, click the tab button so that the text you typed into the edit buffer is accepted.



10. Click the Step Out button to return to the next line of code in the `ShirtTest` class. The `displayShirtInformation` method will have completed.



11. Notice that the `myShirt` object field variables reflect the changes you made while in the method.
12. Click the Continue button now to finish execution and end the debug session.
13. Click the Output tab to view the output.



You have now experienced some of the most commonly used features of a typical IDE Debugger. You may wish to use the debugger in remaining labs to help you diagnose and fix problems you may experience in your programs.

14. Close the Practice04 project in NetBeans. In the Projects window, right click Practice04 and select **Close** from the context menu.

Practices for Lesson 5: Declaring, Initializing, and Using Variables

Chapter 5

Practices for Lesson 5

Practices Overview

In these practices, you will create several Java classes that declare, initialize and manipulate field variables. Solutions for these practices can be found in `D:\labs\soln\les05`.

Practice 5-1: Declare Field Variables in a Class

Overview

In this practice you create a class containing several fields. You declare the fields and initialize them and then test the class by running the CustomerTest program.

Assumptions

This practice assumes that the CustomerTest Java source file appears in the lab folder for this lesson: `D:\labs\les05`

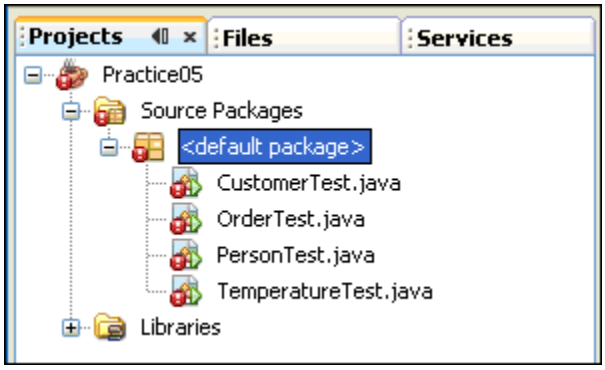
Tasks

1. Close any open project in NetBeans. In the Projects window, right click the project name and select **Close** from the context menu.
2. Create a new project from existing Java source, using the values in the table below when you complete the New Project wizard.

Step	Window/Page Description	Choices or Values
a.	Choose Project step	Category: Java Project: Java Project with Existing Sources
b.	Name and Location step	Project Name: Practice05
c.	Existing Sources step	**Add Folder**: `D:\labs\les05`
d.	Project Properties window	Set the **Source/Binary Format** property to **JDK 7**

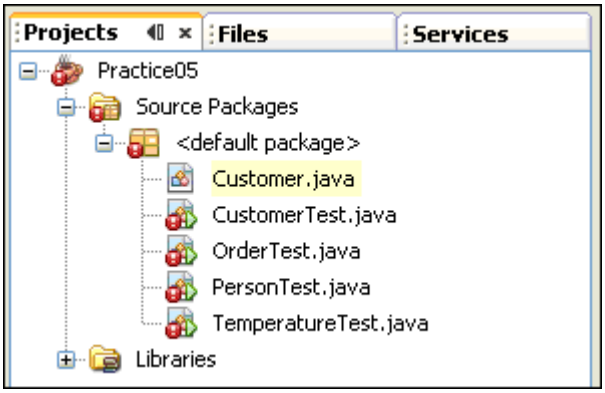
Note: If you need a more detailed reminder of how to create a new project, please refer to Practice 2-2, steps 3 and 4.

Solution: The Projects window should show four Java source files beneath the <default package> node.



3. Create a new Java class. The table below provides the high level steps. If you need more assistance, please refer to Practice 4-2, step 1.

Step	Window/Page Description	Choices or Values
a.	Menu	**File > New File**
b.	New File window | Choose File Type step	Category: Java File Types: Java class **Next**
c.	New Java Class window | Name and Location step	Class Name: Customer **Finish**



4. With Customer.java open for editing in the Editor pane, declare and initialize the fields described in the table below. If you need more assistance, the more detailed steps are provided following the table.

Field Name	Data Type	Default Value
`customerID`	`int`	<your choice>
`status`	`char`	<your choice> 'N' for new, 'O' for old
`totalPurchases`	`double`	0.0

   a. The syntax of a variable declaration and initialization is:
   ```
   modifier type variable = <value>;
   ```
   b. Assume that all fields are `public`.
   c. Include a comment at the end of each line describing the field.

   Solution: This shows one possible solution for the `customerID` declaration and initialization. The others will be similar.

```
public int customerID = 0;  // Default ID for a customer
```

5. Add a method within the `Customer` class called `displayCustomerInfo`. This method uses the `System.out.println` method to print each field to the screen with a corresponding label (such as "Purchases are: ").

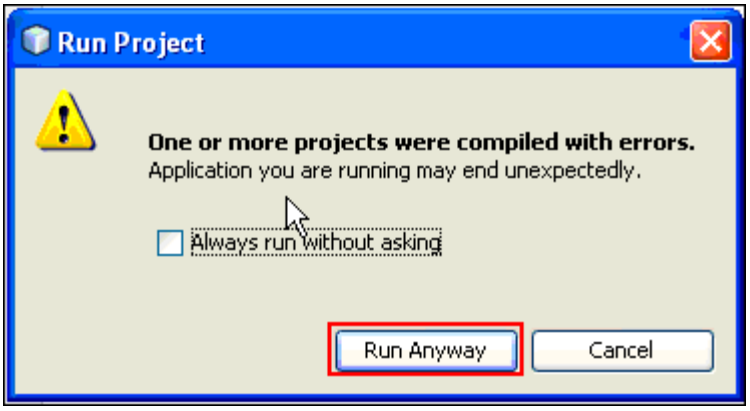
Solution:

```
public void displayCustomerInfo () {
      System.out.println("Customer ID: " + customerID);
            // continue in a similar fashion for all other fields
}
```

6. Click Save to compile the class.

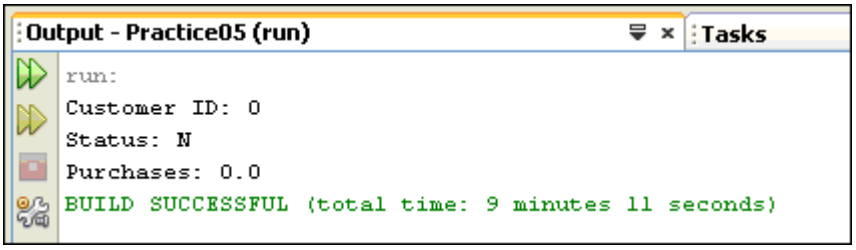
   Note: You will notice that the red error indicator next to the `CustomerTest` class in the Projects window disappears after saving the `Customer` class. The reason is that the `CustomerTest` class references the `displayCustomerInfo` method, which did not exist before you saved the file. NetBeans recognized a potential compilation error in the `CustomerTest` class, due to the missing method.

7. Run the `CustomerTest` class to test your code. If you are prompted with a warning indicating that there are compilation errors within the project, click **Run Anyway**.



   Note: All of the examples and practices in this course require a test class. In most situations, the test class is provided. However, in some situations, you will create the class.

8. Check the output to be sure that it contains the values you assigned.



Practice 5-2: Use Operators and Perform Type Casting to Prevent Data Loss

Overview

In this practice you practice using operators and type casting. This exercise has three sections. In each section you create one Java class, compile it, and test it.

Assumptions

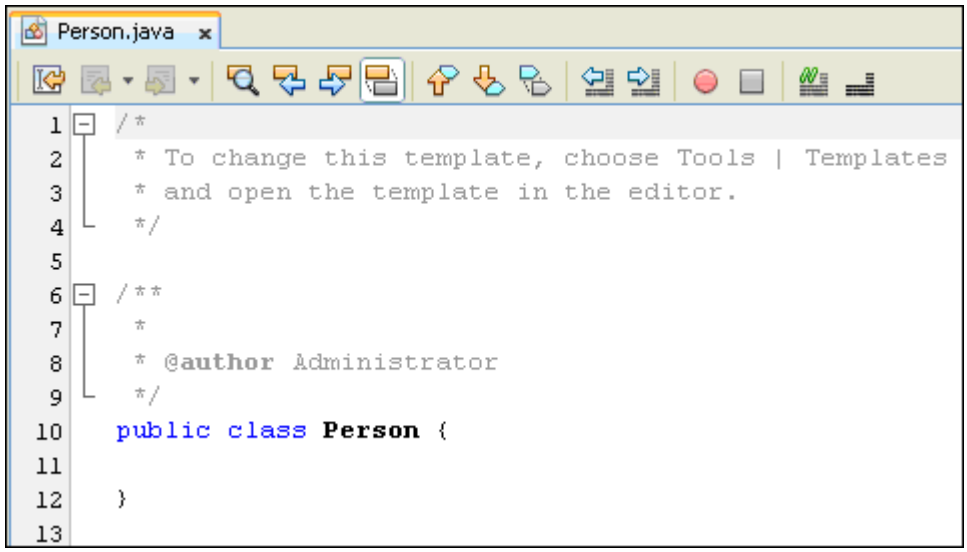
The following Java source files appear in the lab folder for this lesson: `D:\labs\les05`

- PersonTest.java
- OrderTest.java
- TemperatureTest.java

Calculating Age Using Operators

In this task, you use operators to calculate age in days, minutes, seconds, and milliseconds.

1. Select File > New File from the menu to create a new Java class called Person.



2. Using the editor, add the following fields to store age in years, days, minutes, seconds, and milliseconds. Provide meaningful names for all the fields. The table below provides more detailed information:

Year Part	Data Type	Additional Info
Years	`int`	Initialize to 1
Days	`int`	Do not initialize
Minutes	`long`	Do not initialize
Seconds	`long`	Do not initialize
Milliseconds	`long`	Do not initialize

Hint: You can declare multiple variables of the same type in one line by separating the variables by a comma. Be sure to end the line with a semicolon, just as you would any other line of code.

3. Create a new public method in this class called `calculateAge`.
   a. The method should calculate age in days, minutes, seconds, and milliseconds, assigning the value to the relevant field. The following table gives you the calculations:

Year Part	Calculated By:
Days	Year * 365
Seconds	Days * 24 * 60 * 60
Minutes	Seconds / 60
Milliseconds	Seconds * 1000

   b. Print out all the ages in various units, each in a separate line with an appropriate message. For example "You are 3156000 seconds old."

Solution:

```
public void calculateAge () {
      ageDays = ageYears * 365;
      ageSeconds = ageDays * 24 * 60 * 60;
      ageMinutes = ageSeconds / 60;
      ageMilliseconds = ageSeconds * 1000;

      System.out.println ("You are " + ageDays + " days old.");
      System.out.println("You are " + ageMinutes +
        " minutes old.");
      System.out.println("You are " + ageSeconds +
        " seconds old.");
      System.out.println("You are " + ageMilliseconds +
        " milliseconds old.");
}
```

4. Save to compile the class and then run the `PersonTest.java` file.
5. Perform several tests, by setting the value of age as 1, 24, and 80 in the `Person` class.

Solution:

For one year, the results should be: You are 365 days old. You are 31536000 seconds old. You are 525600 minutes old. You are 31536000000 milliseconds old.

Using Casting to Prevent Data Loss

In this section you use casting to ensure that data loss does not occur in your programs.

6. Create a new Java class called `Order`

7. Add three fields to the `Order` class as follows:

Field Name	Data Type	Initialized Value
`orderValue`	`long`	0L (zero L)
`itemQuantity`	`int`	10_000_000
`itemPrice`	`int`	555_500

Note: The underscores used to initialize the `int` values improve the readability of your code. They have no effect on the actual numeric value of the field. The compiler strips them out. This is one of the new language features of Java 7.

8. Create a `calculateTotal` method that will calculate the total order value (`quantity` * `price`) and print it. Be sure to type cast both `itemQuantity` and `itemPrice` to a `long` so that the `int` values will not be truncated. (Both of these values are too large for an `int` type.).

Solution:

```
public void calculateTotal(){
     orderValue = (long)itemQuantity * (long)itemPrice;
     System.out.println("Order total: "+ orderValue);
}
```

9. Save `Order.java` and then test it by running `OrderTest.java`. Verify the result by using a calculator.

Solution: Result should be 5555000000000

10. Edit the `Person.java` file to remove the type casting done in the `calculateTotal` method.
11. Compile and run `OrderTest` again to see the resulting data loss that occurs without type casting.

Creating a Temperature Program

In this section, you write a program to convert temperature from Fahrenheit to Celsius.

12. Create a new Java class called Temperature. Add a member field to the `Temperature` class that stores the temperature in Fahrenheit. Declare the field variable with an appropriate data type, such as `int`, `float`, or `double`.
13. Create a `calculateCelsius` method. Convert the Fahrenheit temperature to Celsius by subtracting 32, multiplying by 5, and dividing by 9. Be sure to observe the rules of precedence when typing this expression.

Hint: The rules of precedence are listed here for your convenience.

- Operators within a pair of parenthesis
- Increment and decrement operators
- Multiplication and division operators, evaluated left to right
- Addition and subtraction operators, evaluated left to right

Solution: This is one possible solution.

```
public class Temperature {
     public float fahrenheitTemp = 78.9F;

     public void calculateCelsius() {
          System.out.println ((fahrenheitTemp - 32) * 5 / 9);
     }
}
```

14. Compile the `Temperature` class and test it using the `TemperatureTest` class. Confirm that you get the same result running the program as you do when doing this calculation using a calculator.
15. Test the program using several values of temperature.
16. When you have finished experimenting with different values, close the Practice05 project.

Practices for Lesson 6: Working with Objects

Chapter 6

Practices for Lesson 6

Practices Overview

In these practices, you will create and manipulate Java technology objects and also create and use String and StringBuilder objects. In the last exercise you will get familiar with the Java API specification. Solutions for these practices can be found in `D:\labs\soln\les06`.

Practice 6-1: Create and Manipulate Java Objects

Overview

In this practice you create instances of a class and manipulate these instances in several ways. This Practice has two sections. In the first section you create and initialize object instances. In the second section, you manipulate object references.

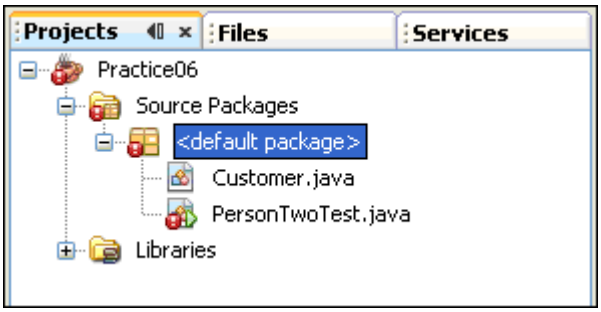
Assumptions

The `Customer.java` file appears in the lab folder for this lesson: `D:\labs\les06`

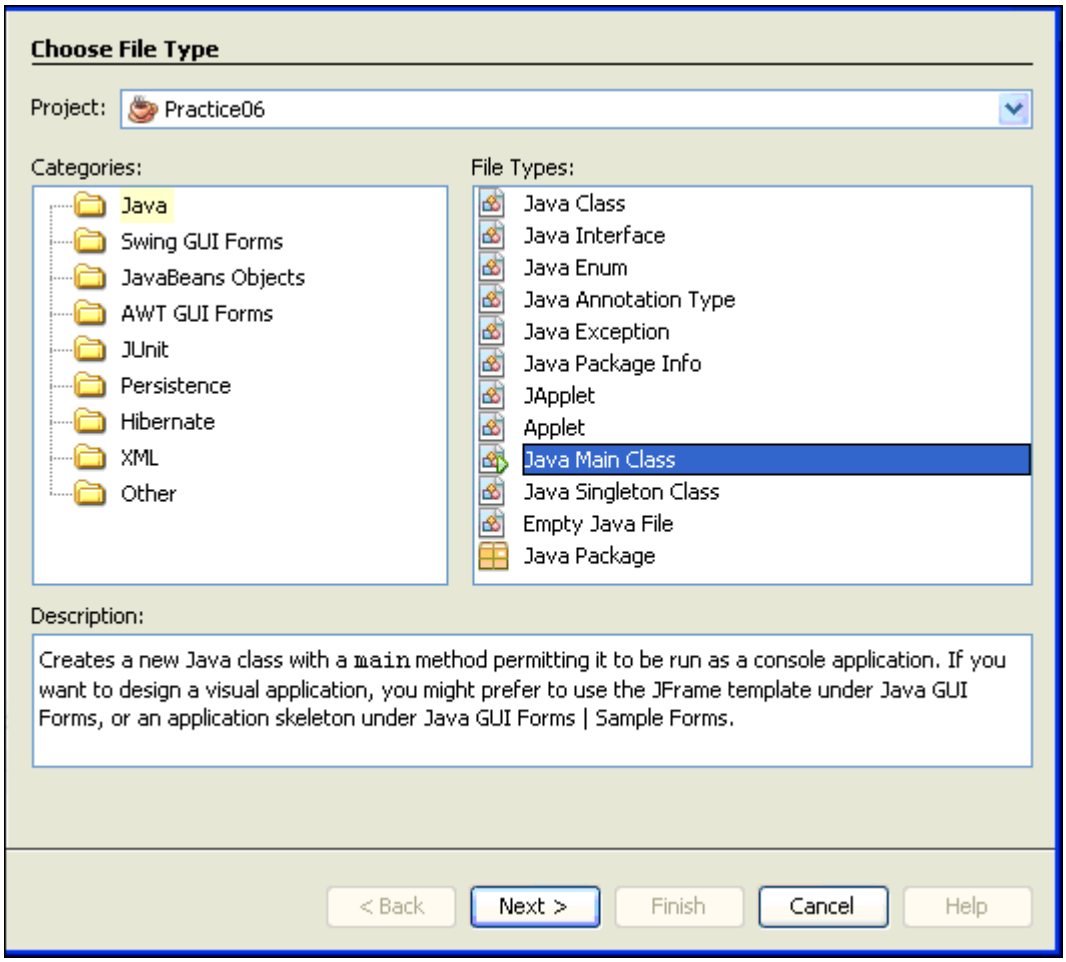
Initializing Object Instances

A `Customer` class is provided for you. In this section you create, compile, and execute a `CustomerTest` class. In this test class, you create objects of the Customer class and set values to its member fields.

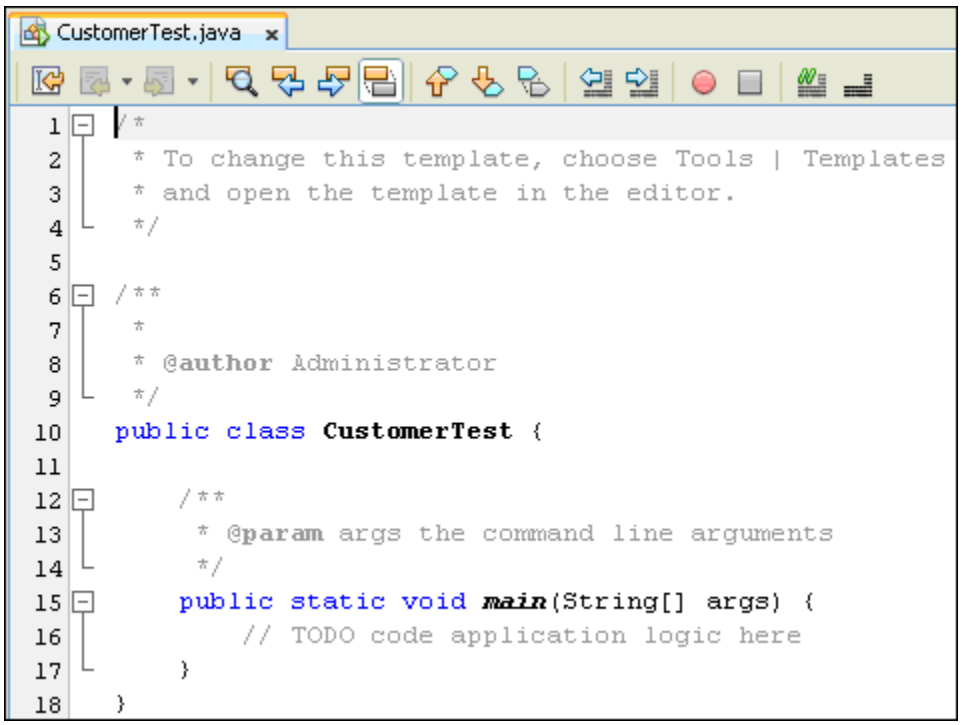
1. Create a new project from existing source called Practice06. Set the **Source Package Folder** to point to `D:\labs\les06`. Remember to also change the Source/Binary Format property. If you need further details, refer to Practice 2-2, Step 3.



2. Open the `Customer.java` file in the editor and examine its member fields and its method. You will use the field information to complete this practice.
3. Create the CustomerTest class as a “Java Main Class” type. Since this class will be run (executed) by the Java executable, it must contain a `main` method. The NetBeans IDE provides the skeleton of a main class for you.
   a. Select File > New File from the NetBeans menu.
   b. In the Choose File Type window select Java Main File from the File Types column. Click **Next**.



c. Name the file “CustomerTest” and click **Finish**.



4. In the `main` method of `CustomerTest`, add code to declare and initialize two instances of the `Customer` class. The table below provides high level instructions for this task. If you need more assistance, refer to the detailed steps following the table.

Step	Window/Page Description	Choices or Values
a.	Declare two fields of type `Customer`	**cust1** **cust2**
b.	Initialize the two instances	Use the `new` operator

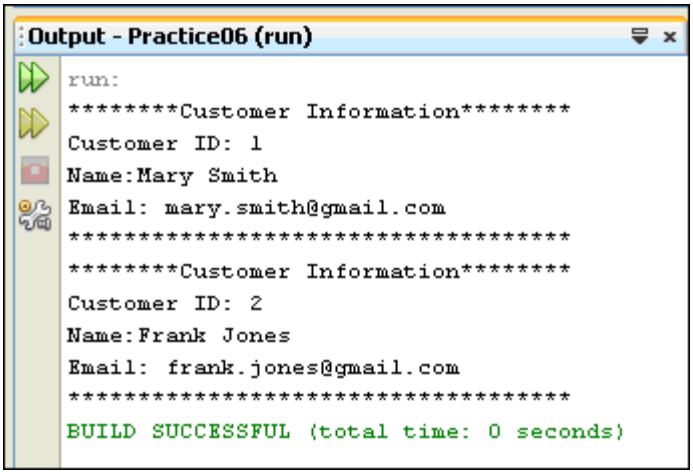
   a. Within the body of the `main` method, declare two fields of type Customer as follows:
   ```
   Customer cust1, cust2;
   ```
   b. Initialize each of the variables using this syntax:
   ```
   <variable name> = new <class name>();
   ```

5. Finish coding the main method as indicated in the following table. More detailed instructions are provided below the table.

Step	Window/Page Description	Choices or Values
a.	Assign values to the member fields of one of the `Customer` objects	Example: `cust1.customerID = 1;`
b.	Repeat for the other Customer object but use a different values for the fields.	
c.	Invoke the `displayCustomerInfo` method of each object	Use the reference variable to qualify the method as you did in step a.

   a. Assign values to all of the member fields of one of the Customer objects. Use the reference variable to qualify the field name as shown below:
   ```
   cust1.customerID = 1;
   ```
   b. Assign different values to each member field of the other `Customer` object.
   c. Invoke the `displayCustomerInfo` method of each object. Example:
   ```
   cust1.displayCustomerInfo();
   ```

6. Click Save to compile.
7. Run the `CustomerTest.java` file. Check the output to be sure that each `Customer` object displays the distinct values you assigned.



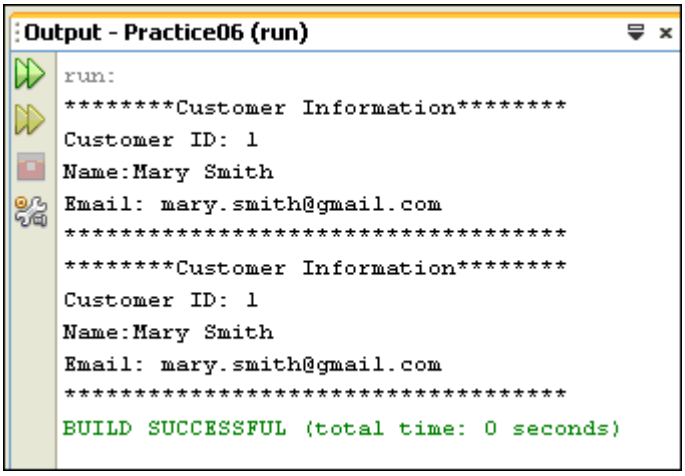
Manipulating Object References

In this section you assign the value of one object reference to another object reference.

8. Edit the `main` method of the `CustomerTest` to assign one object reference to another object reference just above the first line of code that invokes the `displayCustomerInfo` method. For example (assuming that `cust1` and `cust2` are instances of the `Customer` class):

```
cust2 = cust1;
```

9. Save and run the `CustomerTest.java` file. Check the output of the `displayCustomerInfo` methods for both objects. Both of the object references now point to the same object in memory so both of the `displayCustomerInfo` method outputs should be identical.



Practice 6-2: Use the String and StringBuilder Classes

Overview

In this practice you create and initialize String objects and print their contents. You also create and manipulate StringBuilder objects. This practice has two sections. In the first section you work with String objects. In the second section you work with StringBuilder objects.

Assumptions

The `PersonTwoTest.java` file appears in the lab folder for this lesson: `D:\labs\les06`

Creating and Using String Objects

1. Create a new Java class called "PersonTwo".
2. Declare and instantiate two member fields of type `StringBuilder` to hold the person's name and phone number, respectively. For the name field, initialize the capacity of the `StringBuilder` object to 8. Use meaningful field names.

   Example Solution:

```
public class PersonTwo {
     public StringBuilder name = new StringBuilder(8);
     public StringBuilder phoneNumber = new StringBuilder();
}
```

3. Create a new method called "displayPersonInfo".
4. In the body of the `displayPersonInfo` method, populate and then display the `name` object. Ensure that the total number of characters in the name exceeds the initial capacity of the object (8). The following table provides high level steps for this task. More detailed instructions can be found below the table.

Step	Window/Page Description	Choices or Values
a.	Add a first name to the `StringBuilder` object	Use the `append` method of the `StringBuilder` class
b.	Append two more values to the `name` object	a space: " " a last name Note: The total number of characters appended should exceed 8
c.	Display the `String` value of the `name` object	Use the `toString` method of the `StringBuilder` class
d.	Display the capacity of the `name` object with a suitable label	Use the `capacity` method of the `StringBuilder` class
e.	Compile and run the program	Run the `PersonTwoTest.java` file

   a. Use the `append` method of the `StringBuilder` class to append a first name. Example:

```
name.append("Fernando");
```

   b. Use the same method in two separate invocations to add first a space (" "), and then a last name. Ensure that total number of characters that you have added to the `name` object exceeds 8.

Note: You can accomplish the same thing by using a String object and concatenating additional values. However, this would be inefficient because a new String object is created with each concatenation. String object capacity cannot be increased as Strings are immutable.

c. Use the `System.out.println` method to display the entire name value. You can embed the `toString` method of `name` object within the `System.out.println` method.

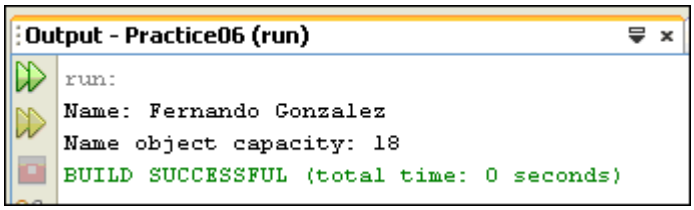
```
System.out.println("Name: " + name.toString());
```

d. Display the capacity of the `name` object, using the `capacity` method. The `StringBuilder` object will have dynamically increased the capacity to contain all of the values that you have appended.

Example Solution:

```
public void displayPersonInfo(){
     name.append("Fernando");
     name.append(" ");
     name.append("Gonzalez");
     // Display the name object
     System.out.println("Name object capacity: " +
        name.toString());
     // Display the capacity
     System.out.println("Capacity: " + name.capacity());
}
```

e. Click Save to compile. Run the `PersonTwoTest.java` file. The output should look similar to the screenshot below. Notice that the capacity has been increased from the initial setting of 8 to accommodate the full name.



5. Populate and manipulate the `phoneNumber` object. Here you append a string of digits and then use the `insert` method to insert dashes at various index locations, achieving the format "nnn-nnn-nnnn". The table below provides high level instructions for this task. More detailed instructions can be found below the table.

Step	Window/Page Description	Choices or Values
a.	Append a 10 digit `String` value to the `phoneNumber` object	Example: "5551234567"
b.	Insert a dash ("-") after the first three characters of the `phoneNumber`.	Use the `insert` method that takes an `int` value for the offset and inserts a `String` value. (Use offset number 3)
c.	Insert another dash after the first seven characters of the `phoneNumber`	Reminder: The previous insertion pushed the remaining characters over one index.
d.	Display the `phoneNumber` object	Use the `toString` method of the `StringBuilder` class

   a. Use the `append` method of the `StringBuilder` class to append a `String` value consisting of ten numbers.

   b. Insert a dash ("-") at offset position 3. This puts the dash at the 4th position in the String, pushing all of the remaining characters over one position. The syntax for this method is shown below:

```
<object reference>.insert(int offset, String str);
```

> Example: Given the following string,
>
> "5551234567"
>
> offset position 3 occurs at the number 1. (Index numbers begin at 0) If the dash is inserted at offset position 3, it pushes the number currently at that position and all remaining numbers over to the next offset position.

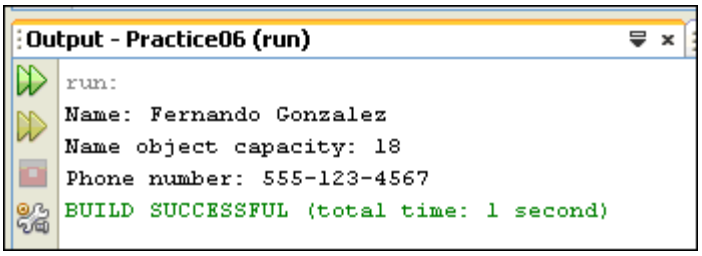
   c. Insert a dash at offset position 7 (where the number 4 is currently placed).

   d. Use `System.out.println` to display the output from the `StringBuilder` object's `toString` method.

Solution:

```
phoneNumber.append("5551234567");
phoneNumber.insert(3, "-");
phoneNumber.insert(7, "-");
System.out.println("Phone number: " +
   phoneNumber.toString());
```

6. Click Save to compile. Run the `PersonTwoTest.java` file. Check the output from the `displayPersonInfo` method. Ensure that the dashes appear between the third and fourth digits and between the sixth and seventh digits.



7. Use the `substring` method of the `StringBuilder` class to get just the first name value in the `name` object. Use the `substring` method that takes the start index and the end index for the substring. Display this value using `System.out.println`.

Syntax:

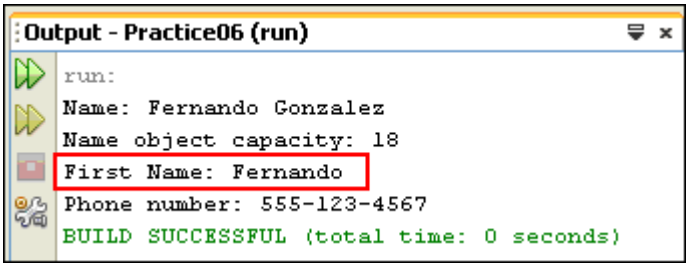
```
<object reference>.substring(int start, int end);
```

> **Note:** Indexes for characters in the StringBuilder class, much like array indexes, are zero-based. The first character in the StringBuilder is located at position (or index) 0.
>
> While the start index of the `substring` method is inclusive (it is the *actual* index of the first character you want returned), the end index is exclusive (it is the index of the character just to the right of the last character of your substring.)

Example Solution:

```
// Assumes the first name "Fernando"
System.out.println("First name: " + name.substring(0,8));
```

8. Save and again run the `PersonTwoTest.java`. Check the output and make any adjustments necessary to the index numbers in order to get the correct first name value.



Practice 6-3: Examine the Java API Specification

Overview

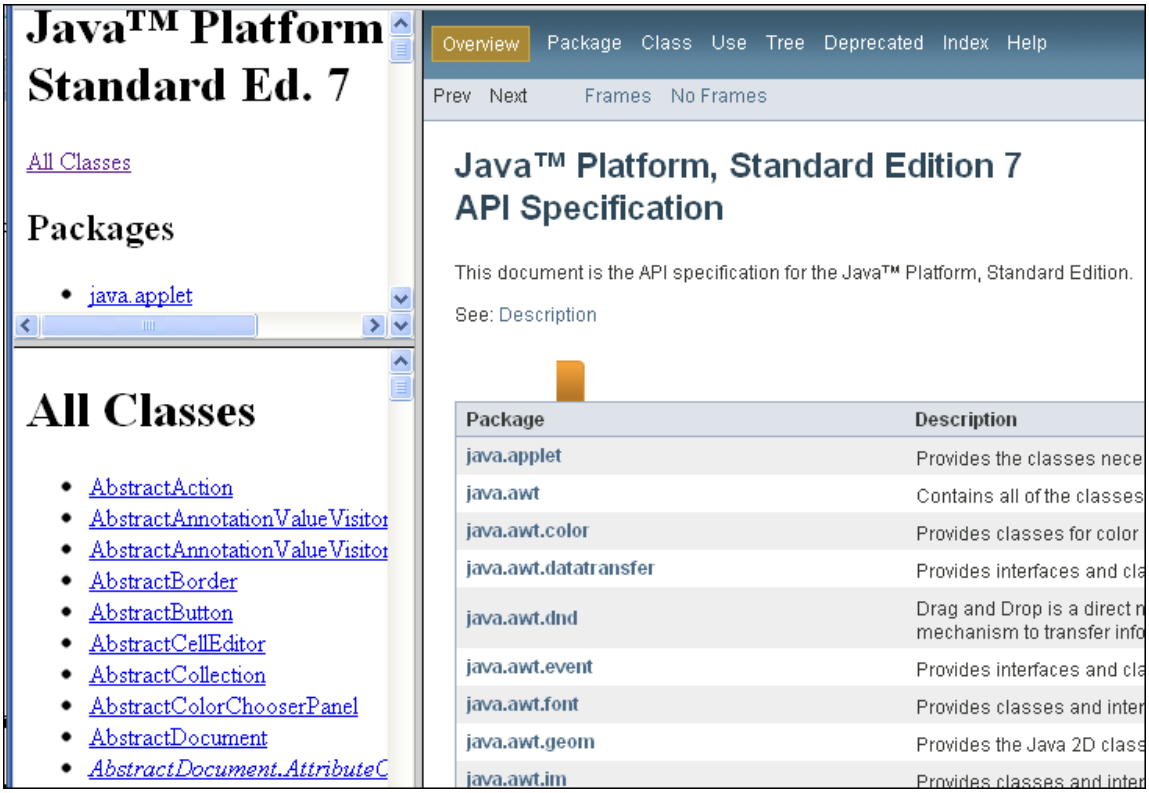
In this practice you examine the Java API specification in order to become familiar with the documentation and how to look up classes and methods. You are not expected to understand everything you see. As you progress through this course, the Java API documentation should make more sense.

Assumptions

The Java SE 7 API specification is installed locally on your machine.

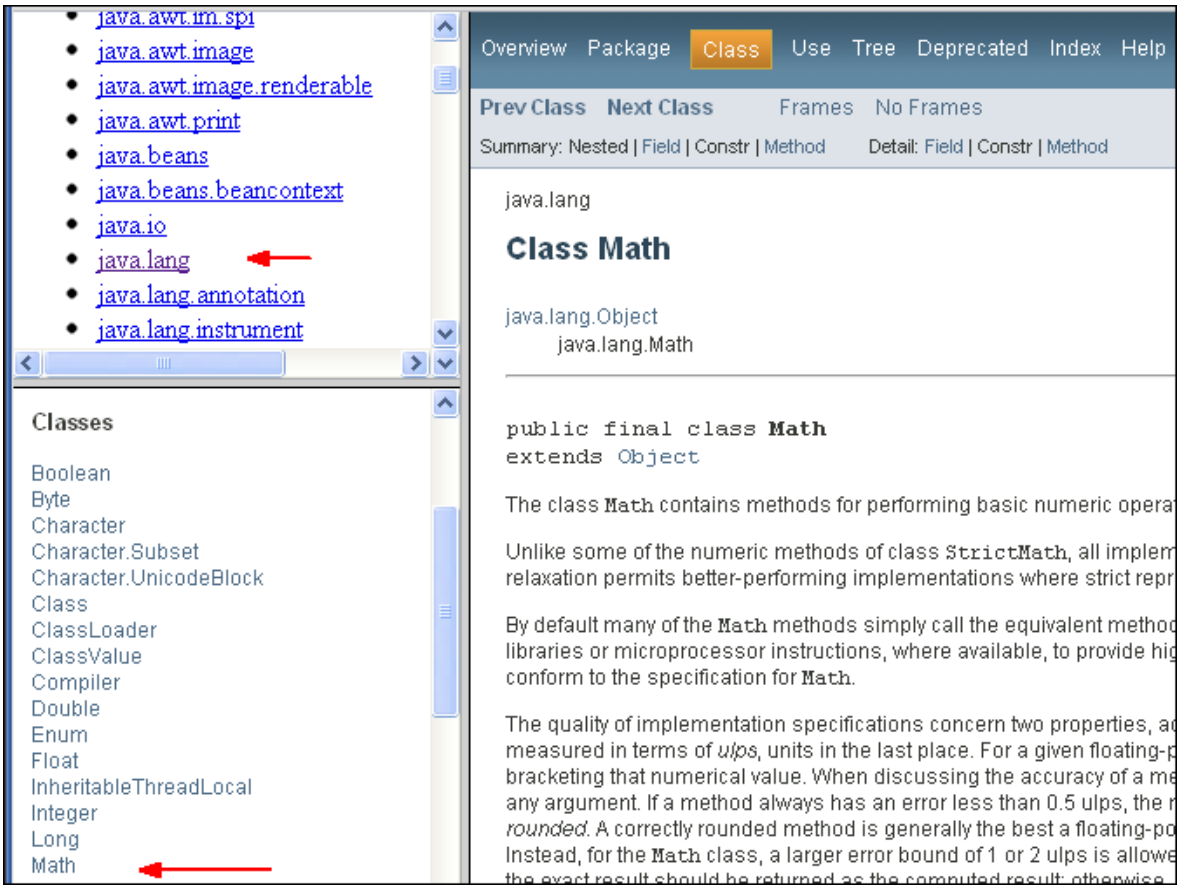
Tasks

1. To view the Java SE7 API specification (also referred to as "javadocs"), double click the shortcut on your desktop (entitled "Java JDK7 1.7.0 API Docs").



The opening page of the javadocs consists of three frames as shown above. It allows you to navigate through the hierarchy of classes in the API by class name or by package. (**Note:** you learn about packages later in this course)

2. Using the **Packages** frame, select the `java.lang` package. The **All Classes** frame will now change to display only classes within that package.
3. Find the `Math` class and click it to display its documentation in the main frame.



4. Answer the following questions about the Math class:
   a. How many methods are there in the `Math` class?_____
   b. How many fields are there in the `Math` class?_____

   Answer:
   a.) 54
   b.) 2

5. Select several other classes in the Classes panel in order to answer this question: What class does every class refer to at the top of the page? Hint: What class is the superclass to all classes? _____

   Answer: `Object`

6. Find the `String` class and identify the methods of `String` class Which methods will enable you to compare two strings?_____

   Answer: `compareTo` and `compareToIgnoreCase`

7. Close the Practice06 project in NetBeans.

Practices for Lesson 7: Using Operators and Decision Constructs

Chapter 7

Practices for Lesson 7

Practices Overview

In these practices, you will create `if` and `if` / `else` constructs and also create a `switch` construct. Solutions for these practices can be found in `D:\labs\soln\les07`.

Practice 7-1: Write a Class that Uses the `if/else` Statement

Overview

In this practice you create classes that use `if` and `if` / `else` constructs. There are two sections in this practice. In the first section, you create the DateTwo class that uses if / else statements to display the day of the week based on the value of a variable. In the second section, you create the Clock class that uses if / else statements to display the part of the day, depending on the time of day.

Assumptions

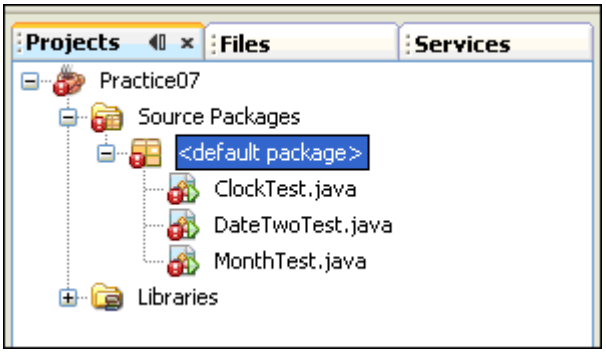
The following files appear in the lab folder for this lesson, `D:\labs\les07`:

- `ClockTest.java`
- `DateTwoTest.java`

Writing a Class that Uses an `if/else` Statement

In this task, you create a DateTwo class that evaluates a numeric field in order to determine the day of the week that corresponds to that number. You use an `if/else` construct to do this.

1. Create a new project from existing sources called Practice07. Set the **Source Package Folder** to point to `D:\labs\les07`. Remember to also change the Source/Binary Format property. If you need further details, refer to Practice 2-2, Steps 3 and 4.



2. Using the New File wizard, create a new Java class called “DateTwo”. Declare and initialize a member field for this class called `dayNumber`. The value assigned should be a number between 1 and 7 (inclusive) where the number 1 represents Monday (beginning of the week) and 7 represents Sunday (end of the week).

   Hint: Use the `int` data type.

3. Create a `displayDay` method in the DateTwo class. High level instructions for this task are provided in the table below. More detailed instructions can be found following the table.

Step	Window/Page Description	Choices or Values
a.	Use an `if/else` construct to inspect the value of `dayNumber`.	In each `if` block, display the corresponding day of the week
b.	Display an error message if an invalid number is found	This should be the last condition you check

   a. The following pseudo code will help you write the body of the `displayDay` method. Each `if` condition should check the value of `dayNumber`. Hint: Use the `==` sign. Within the `if` blocks, print out the day of the week (“Monday”, “Tuesday”, etc.)

```
if (condition1) {
    // print corresponding day
}else if (condition2) {
    // print corresponding day
}else if (condition3)
    …
}else {
    // if none of the conditions is true
}
```

b. If `dayNumber` does not equal a number between 1 and 7 (inclusive), print out an error message. This will be in the final else block.

4. Save, compile, and execute your class by running the DateTwoTest class. Check the output in the Output window.
5. Repeat step 4 several times by assigning different values to the `DateTwo` member field.

Writing Another Class That Uses `if/else` Statements

In this task, you write a class called "Clock" that uses `if/else` statements to display the part of day depending upon the time of day. Use the following table as a guideline.

Time of Day	Part of Day
8:01 to 12:00	Morning
12:01 to 17:00	Afternoon
17:01 to 24:00	Evening
0:01 to 8:00	Early Morning

6. Create a new Java class called "Clock" that contains an `int` field called `currentTime`. Initialize this field to hold the hour of the day. (Example: 400 = 04:00, 1505 = 15:05).
7. In the Clock class, create a `displayPartOfDay` method. Display the part of the day associated with the value of the `currentTime` field. For instance, if the `currentTime` field equals 2100, you would display "Evening". You need not check for values outside the range of 1 to 2400.

   Hint: Use a similar `if/else` construct to what you used in the previous task.

Solution:

```
public void displayPartOfDay() {
     if(currentTime >= 801 && currentTime <= 1200){
          System.out.println("Morning");
     }else if(currentTime >= 1201 && currentTime <= 1700){
          System.out.println("Afternoon");
     }else if(currentTime >= 1701 && currentTime <= 2400){
          System.out.println("Evening");
     }else {
          System.out.println("Early Morning");
     }
}
```

8. Save, compile and execute your program by running the ClockTest class.
9. Repeat Step 8 several times by assigning different values to the `Clock` member variable.

   Note: A leading zero indicates an octal value. Therefore, the program does not compile if you set `currentTime` to 0800. You need to specify `currentTime` as 800 for 8:00 AM to successfully compile the program. No tests have been done for values that lie outside the range of 100 and 2400.

Practice 7-2: Write a Class that Uses the `Switch` Statement

Overview

In this practice you create a class called "Month" that uses switch statements to display the name of the month based upon the numeric value of a field.

Assumptions

The MonthTest.java file appears in the lab folder for this lesson, `D:\labs\les07`

Tasks

1. Create a new Java class called "Month".
2. Declare an `int` field in the Month class called `monthNumber`. Assign a value to the field that is between 1 and 12 (inclusive), where 1 represents the month of January and 12 represents the month of December.
3. Create a new method in the Month class called `displayMonth`. This method uses a `switch` construct to inspect the value of the `monthNumber` field and display the corresponding name of the month. The `displayMonth` method should also display an error message if an invalid number is used.

   Hint: The syntax for a `switch` statement is:

```
switch (<variable>){
     case <value1>:
          // do something
          break;
     case <value2>:
          // do something
          break;
     … // more cases
     default:
          // possibly error checking
          break;
} // end of switch
```

   Solution: Please see the solution file in the `D:\labs\soln\les07` folder.
4. Save, compile and execute your program by running the `MonthTest` class.
5. Repeat step 4 several times assigning different values to the `monthName` field.
6. Close the Practice07 project in NetBeans.

Practices for Lesson 8: Creating and Using Arrays

Chapter 8

Practices for Lesson 8

Practices Overview

In these practices, you will create and populate arrays and ArrayLists. You also use the methods of the ArrayList class to manipulate its values. In the last exercise you create and use a two-dimensional array. Solutions for these practices can be found in `D:\labs\soln\les08`.

Practice 8-1: Creating a Class with a One-dimensional Array of Primitive Types

Overview

In this practice you create an array containing the number of days that an employee at the Duke's Choice company receives, based upon the number of years that the employee has worked for Duke's Choice. The following table shows the vacation scale for Duke's Choice:

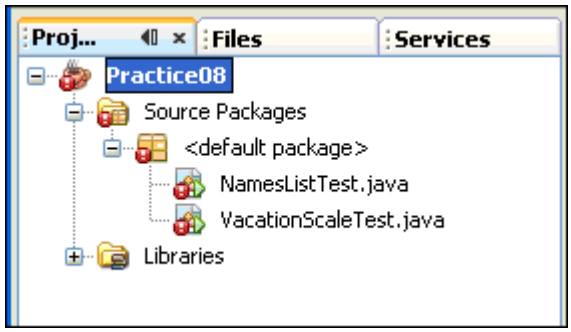
Number of Years of Employment	Number of Days of Vacation
Up to 1 year	10
1, 2, or 3 years	15
4 or 5 years	20
6 or more years	25

Assumptions

The VacationScaleTest.java file appears in the lab folder for this lesson, `D:\labs\les08`.

Tasks

1. Create a new project from existing source called Practice08. Set the **Source Package Folder** to point to `D:\labs\les08`. Remember to also change the Source/Binary Format property. If you need further details, refer to Practice 2-2, Step 3.



2. Create a new Java class called `VacationScale`. Declare but do not initialize two public fields to this class as follows:
   - `int` array called `vacationDays`
   - `int` called `yearsOfService`

   Hint: Use the square brackets ([ ]) next to the data type to indicate that this field is an array.

3. Create a method in the `VacationScale` class called `setVacationScale`. The table below provides high level steps for this task. More detailed steps can be found below the table.

Step	Window/Page Description	Choices or Values
a.	Initialize the `vacationDays` array	Set size of the array to 7
b.	Populate each element of the `vacationDays` array to align a number of years of service with the correct number of vacation days.. (See the table above)	The value = number of vacation days The element index = number of years of service

   a. Use the `new` keyword to initialize the `vacationDays` array. Supply the size of the array within the square brackets as shown below.

      ```
      vacationDays = new int[7];
      ```

   b. Assign each array element, beginning with `vacationDays[0]` with the appropriate number of days of vacation from the table shown above in the overview section. For example, an employee with 0 years of service is entitled to 10 vacation days. Therefore, `vacationDays[0] = 10`. An employee with 1 year of service is entitled to 15 days of vacation. Therefore `vacationDays[1] = 15`.

   Solution:

```
public void setVacationScale(){
     vacationDays = new int[7];
     vacationDays[0] = 10;
     vacationDays[1] = 15;
     vacationDays[2] = 15;
     vacationDays[3] = 15;
     vacationDays[4] = 20;
     // … and so on through element 6
}
```

4. Create a public method called `displayVacationDays` that displays the number of vacation days due to an employee with the years of service indicated in the `yearsOfService` field. Use an `if/else` construct to check for an invalid `yearsOfService` (a negative number) and display an error message in this case.

   Hint: You can use a variable within the square brackets to represent the array index number. For example:

   ```
   vacationDays[yearsOfService]
   ```

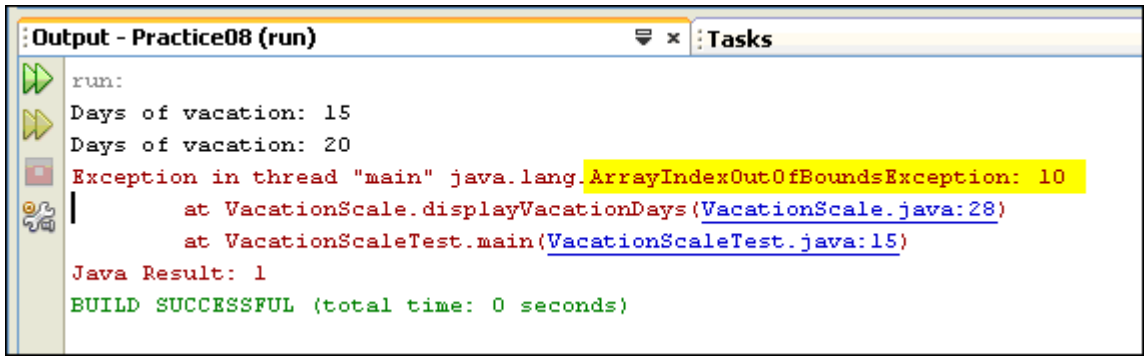
Example:

```
public void displayVacationDays(){
     if(yearsOfService >= 0){
          System.out.println("Vacation days: " +
             vacationDays[yearsOfService]);
     }else {
          System.out.println("Invalid years of service");
     }
}
```

5. Save and compile your program. Run the `VacationScaleTest` class to test your program.

   Note: The program, as currently written, throws an exception (an error). You will fix this problem in the next few steps.

6. The exception thrown by the Java Virtual Machine (JVM) is an `ArrayIndexOutOfBounds` exception. Your Output window should look similar to the screenshot below:



   This exception is thrown when an attempt has been made to access a non-existent index of an array. Notice that the index number that caused the exception is shown in the error message: index #10. Remember that this array has 7 elements, indexed by numbers 0 through 6. Why did the program try to access index 10?

   Answer:

   If you look at the `displayVacationDays` method, you will see that the `yearsOfService` field is used as the array index (as an argument to the `System.out.println` method).

   It is, of course, conceivable that an employee would have more than 6 (the highest index number of the array) years of service. The `displayVacationDays` method needs to be modified to account for >6 years of service.

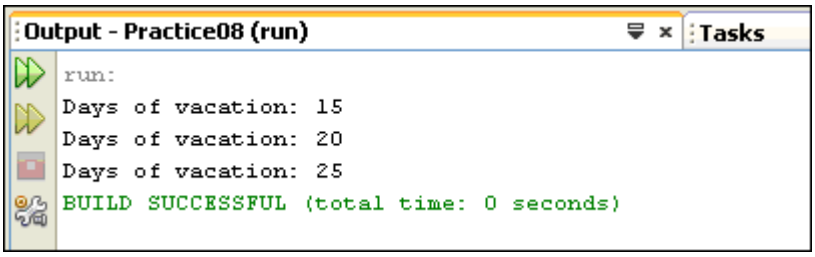
7. Change the `if/else` construct to also check for a `yearsOfService` value that is >=6. All years of service greater than or equal to 6 receive the same number of vacation days.

   Hint: For any `yearsOfService` value between 0 and 5 (inclusive), you can display the value of the array whose index corresponds to that value. For a `yearsOfService` of 6 and above, use the value referenced by the last array index.

Solution:

```
if (yearsOfService >= 0 && yearsOfService < 6) {
     System.out.println("Vacation days: " +
        vacationDays[yearsOfService]);
} else if (yearsOfService >= 6) {
     System.out.println("Days of vacation: " +
        vacationDays[6]);
} else {
     System.out.println("Invalid years of service");
}
```

8. Save and compile the program and then test it again by running the `VacationScaleTest` class. You should now see all three of the test values for `yearsOfService` displayed in the output window.



Practice 8-2: Create and Work With an ArrayList

Overview

In this practice you create the `NamesList` class and the `NamesListTest` class in order to experiment with populating and manipulating ArrayLists. There are two sections in this practice. In the first section, you create the two classes and then add a method to the `NamesList` class to populate the list and display its contents. In the second section, you add a method to manipulate the values in the list.

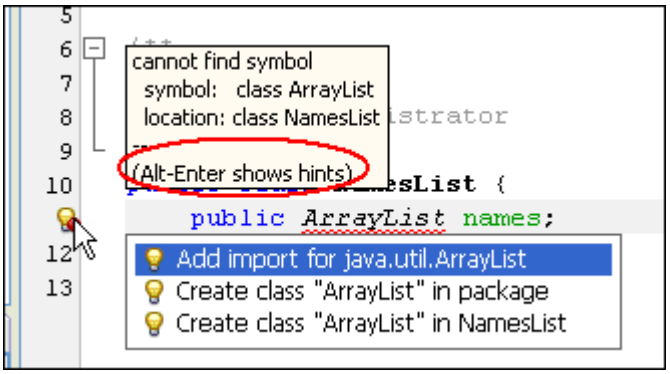
Assumptions

None

Creating and Populating an ArrayList

1. Create a new Java *main* class called `NamesListTest`. Reminder: In the New File wizard, select Java Main Class as the type of Java class. Leave the `main` method empty for the time being. You will add code later.
2. Create a new Java class called `NamesList`.
3. In the `NamesList` class, declare a public `ArrayList` field called `theList`. Do not instantiate the `theList` field.

   Note: When you type the word `ArrayList`, the editor will indicate a warning in the margin of this line. It does not recognize the `ArrayList` class. You must import this class to make it visible to the compiler.
4. Put your cursor over the warning icon in the margin to see the warning description. Click Alt-Enter to view and select from a list of hints to solve this problem. Select **Add import for java.util.ArrayList** as shown below.



The import statement will be placed at the top of the NamesList class (above the class declaration).

```
1
2  import java.util.ArrayList;
3
4  /*
5   * To change this template, choose Tools | Templates
6   * and open the template in the editor.
7   */
8
9  /**
10  *
11  * @author Administrator
12  */
13 public class NamesList {
14     public ArrayList names;
15 }
```

5. Add a new method to the NamesList class called setList. Code the method as described in the table below. More detailed steps can be found below the table.

Step	Code Description	Choices or Values
a.	Instantiate the `theList` object	Use the `new` keyword. Do not specify size.
b.	Add a name (first and last name) to the `theList` object	Use the `add` method Example: `theList.add("Joe Smith");`
c.	Repeat step b three times to add a total of four names to the `theList` object.	
d.	Print the list of names with a suitable label.	You can just print the object, itself. (`theList`)
e.	Print the size of the `theList` ArrayList	Use the `size` method of the `theList` object

a. Use the `new` keyword to instantiate `theList`. Example:

```
theList = new ArrayList();
```

b. Invoke the `add` method of the theList object. Pass a `String` value containing first_name and last_name, separated by a space. (See example in table above)

c. Repeat step b three more times, using a different name in each method invocation.

d. Use `System.out.println` to print out all of the names within the `theList` object. Use a suitable label and concatenate the `theList` object to it.

```
System.out.println("Names list: " + theList);
```

e. Use `System.out.println` to print out the size (number of elements) of the `theList` object. Use the `size` method of the theList object and concatenate a suitable label.

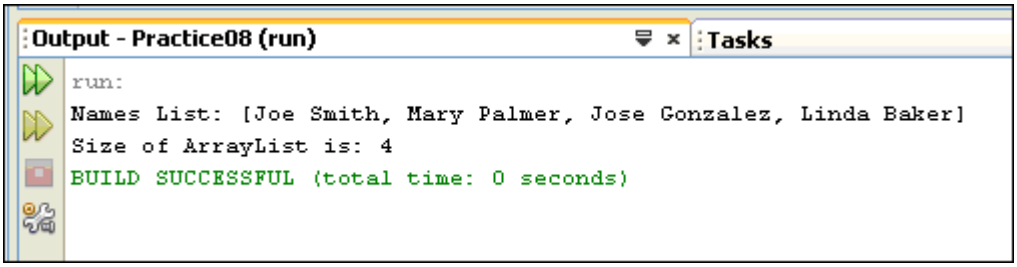
```
System.out.println("Size of ArrayList: " + theList.size());
```

Solution:

```
public void setList(){
     theList = new ArrayList();
     theList.add("Joe Smith");
     theList.add("Mary Palmer");
     theList.add("Jose Gonzales");
     theList.add("Linda Baker");

     System.out.println("Names list: " + theList);
     System.out.println("Size of ArrayList: " + theList.size());
}
```

6. Click Save to compile.
7. Open the `NamesListTest` class in the editor. In the `main` method:
8. Instantiate a NamesList object called "names" using the `new` keyword.
   a. Invoke the `setList` method of the `names` object.
9. Save and compile your program. Run the `NamesListTest` class to test the program.



Manipulating the ArrayList

10. Add another new method to the `NamesList` class called `manipulateList`. Code the method as described in the table below. More detailed steps can be found below the table.

Step	Code Description	Choices or Values
a.	Remove one of the names from the list	Use the `remove(Object obj)` method of the `ArrayList` object. Hint: a `String` literal is an object.
b.	Print the contents of the `theList` object, using a suitable label	
c.	Print the size of the `theList` object	Use the `size` method of the `ArrayList`
d.	Add the name you just removed back into the list in a different location	Use the `add(int index, Object obj)` method of the `ArrayList` object. Hint: Index numbers are zero-based.
e.	Print the contents of the `theList` object	
f.	Print the size of the `theList` object	

a. Remove one of the names in the `ArrayList` using the `remove` method and passing the full name, enclosed in double quotes.
   - Note: This method is defined as taking an `Object` as an argument. A `String` literal, such as the quote-enclosed full name, is an object.

   ```
   theList.remove("Joe Smith");
   ```
b. Use `System.out.println` to print the `theList` object. Use an appropriate label.
c. Use `System.out.println` to print the current size of the `ArrayList`. Use an appropriate label.
d. Use the `add` method of the `ArrayList` to add the name you just removed back into the `ArrayList`, but at a different location in the list than previously.

   Note: The `add` method is "overloaded". That is, it has two different method signatures. One of the `add` methods takes an `Object` and appends it to the end of the `ArrayList`. The other method takes an index number and an `Object`. It inserts the `Object` before the referenced index number, pushing all remaining list elements over one index number. Use the latter `add` method. An example is shown below:

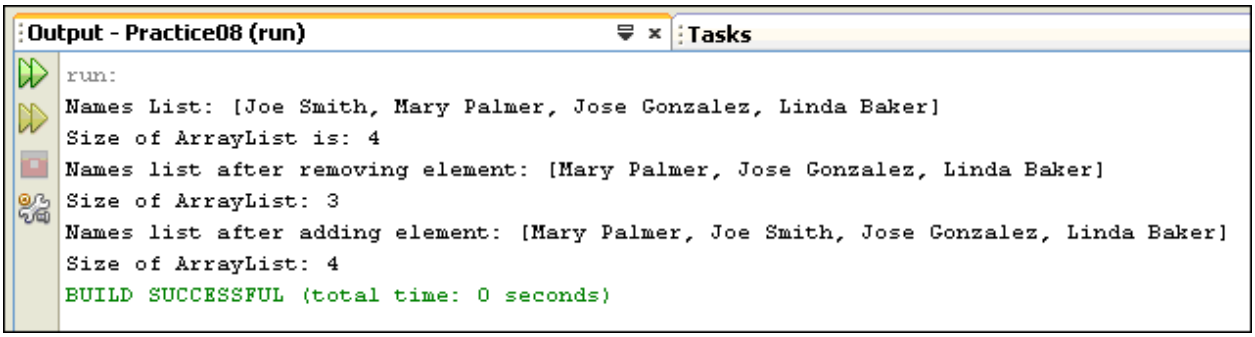
   ```
   theList.add(1, "Joe Smith");
   ```
e. Use a suitable label when printing the newly modified contents of the `theList` object.
f. Use a suitable label when printing the new size of the `theList` object.

Example Solution:

```
theList.remove ("Joe Smith");
System.out.println("Names list after removing element: "
    + theList);
System.out.println("Size of the ArrayList: " +
    theList.size());
theList.add(1, "Joe Smith");
System.out.println("Names list after adding element: "
     + theList);
System.out.println("Size of the ArrayList: " +
    theList.size());
```

11. In the `main` method of the `NamesListTest` class, invoke the `manipulateList` method of the `names` object.

    Note: You may need to click Save so that the compiler can resolve the reference to `manipulateList`.
12. Save and compile the program.
13. Run the `NamesListTest` class to test the program. The output should look similar to the screenshot below, depending upon the name you removed and added, and the index number you used in the `add` method.



Note: In the example shown above, Joe Smith was previously located at index position 0 and Mary Palmer was at index position 1.

After removing Joe Smith, Mary Palmer moved to index position 0 and Jose Gonzalez was at index position 1.

Joe Smith was then added at index position 1, pushing Jose Gonzalez over to index position 2.

Practice 8-3: Use Runtime Arguments and Parse the Args Array

Overview

In this practice you write a guessing game that accepts an argument and displays an associated message. You create a class that accepts a runtime argument between 1 and 5, inclusive. You also randomly generate a number between 1 and 5 in the class and compare the value of the argument with the randomly generated number.

Assumptions

None

Tasks

1. Create a new Java Main Class called `GuessingGame`.
2. In the `main` method, declare two `int` variables as shown below:

```
int randomNum = 0;
int guess;
```

3. Add code to the `main` method to accept a single argument of any number in the range of 1 to 5, inclusive, or the word "help". The high level steps are described in the pseudo code below, followed by helpful hints. If you need additional assistance, follow the steps below the pseudo code and hints. Remember, the solution can also be found in `D:\labs\soln\les08`
4. **Pseudo Code for main Method:**

```
if length of args array == 0 or value of args[0] = "help"
    print a Correct Usage message
else
    randomNum = a generated random number 1 - 5
    guess = integer value of args[0]

    if argument < 1 or > 5
        print an error message (invalid argument)
    else
        if argument == randomNum
            print congratulations message
        else
            print a "Sorry; try again" message
```

Hints:

- Use the `compareTo` method of the `String` class (elements of the args array will always be Strings) to match the `args[0]` to "help".
- To generate the random number 1 – 5, use the following code snippet:
  `randomNum = ((int)(Math.random()*5)+1);`
- Convert the runtime argument to an `int` before assigning it to the `guess` variable. Use the `Integer.parseInt` method to do the conversion.

Detailed Steps

a. If the first argument in the `args` array equals "help" or if the `args` array is empty, display the usage of the program. For example:
   "Usage: java GuessingGame [argument]"
   "Enter a number from 1 to 5 as your guess"

b. If a 1, 2, 3, 4, or 5 is entered:
   - Generate a random number (as shown in Hint above)
   - Convert the `arg[0]` to an `int` and assign it to the `guess` variable.

```
guess = Integer.parseInt(args[0]);
```

   - Compare the `guess` to `randomNum` using a nested `if/else` construct.
   - If they match, display a "Congratulations" message.
   - Else, tell them what the random number was and ask them to "Try again."

Solution:

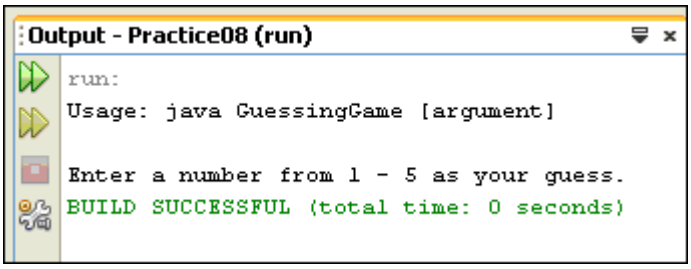
```
public static void main(String[] args){
     int randomNum = 0;
     int guess;
     if(args.length == 0 || args[0].compareTo("help") == 0){
          System.out.println
               ("Usage: java GuessingGame [argument]");
          System.out.println();
          System.out.println
               ("Enter a number from 1 - 5 as your guess.");
     }else {
          randomNum = ((int)(Math.random()*5) +1);
          guess = Integer.parseInt(args[0]);

          if(guess < 1 || guess > 5){
               System.out.println
                ("Invalid argument: Need a number from 1 - 5");
          }else {
               if(guess == randomNum){
                  System.out.println
                     ("Great guess! You got it right!");
               }else {
                    System.out.println
                      ("Sorry.  The number was " + randomNum +
                         ". Try again.");
               }//end of innermost if/else
          } // end of first nested if/else
     }// end of outer if/else
}// end of main method
```

5. Save and compile the program.

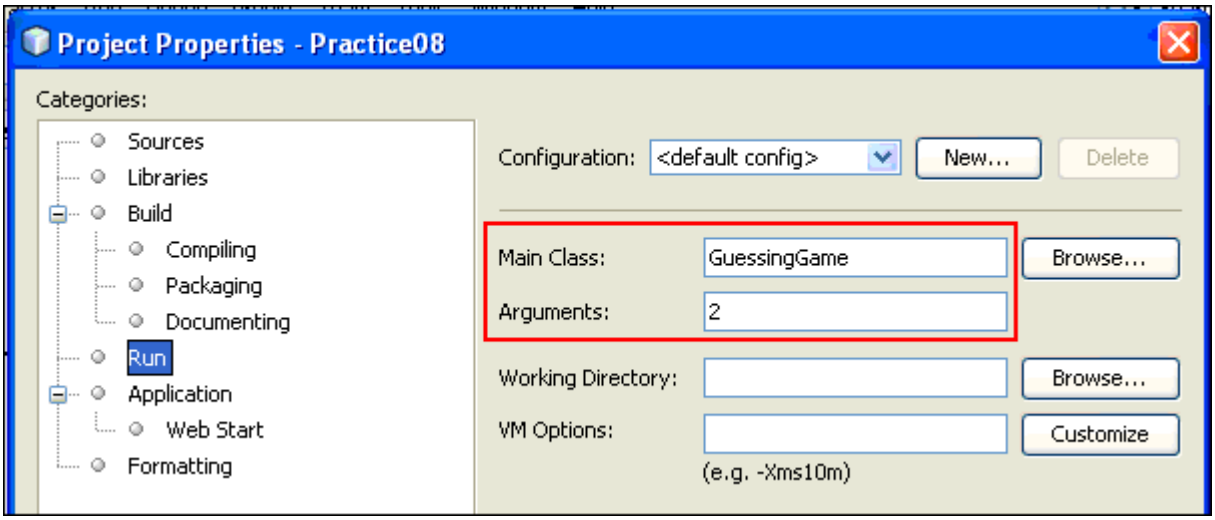
6. Test it by running the `GuessingGame` class.

   Note: Since no runtime parameter was passed to the `args` array, you should get the Usage message as shown here.



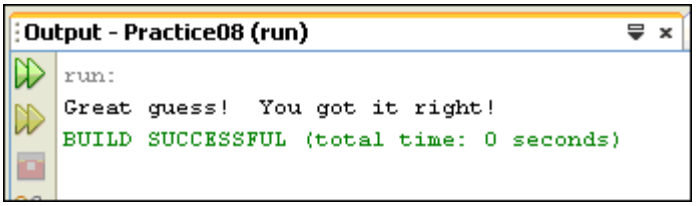
   Note: When using an IDE, you don't have access to the command line to provide runtime parameters. Therefore, you will enter your "guess" (runtime parameter) as a runtime property of the project and then run the entire project, rather than just running an individual file.

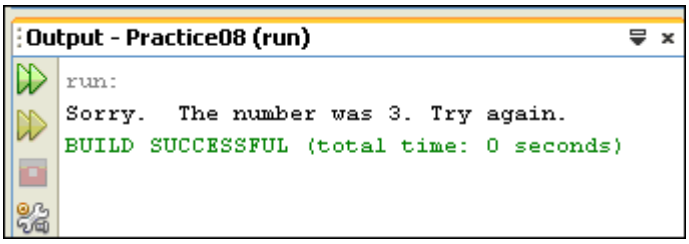
7. Right click the project name in the Projects window and select Properties from the menu.
8. In the Project Properties window, select the **Run** category. Change the **Main Class** to GuessingGame and enter a number from 1 – 5 in the **Arguments** field. Click **OK**.



9. Now run the project by clicking the Run button on the main toolbar.

10. You should receive either the "Congratulations…" message or the "Sorry. …Try again." message. Continue to click the run button to see the different random numbers generated and how the program responds by comparing it with your guess.





11. Close the Practice08 project in NetBeans.

    That's it for Lesson 8 practices. In the practices for the next lesson, you will have an opportunity to work with two-dimensional arrays.

Practices for Lesson 9: Using Loop Constructs

Chapter 9

Practices for Lesson 9

Practices Overview

In these practices, you will use for loops and while loops to process data within arrays or ArrayLists. Two challenge practices are included here for those of you who have extra time and wish to be challenged. Solutions for these practices can be found in `D:\labs\soln\les09`.

Practice 9-1: Writing a Class that Uses a `for` Loop

Overview

In this practice you create the `Counter` class that uses a simple `for` loop to print a sequence of numbers.

Assumptions

The `CounterTest.java` file appears in the lab folder for this lesson, `D:\labs\les09`.

Tasks

1. Create a new project from existing source called Practice09. Set the **Source Package Folder** to point to `D:\labs\les09`. Remember to also change the Source/Binary Format property. If you need further details, refer to Practice 2-2, Step 3.
2. Create a new Java class called "Counter". Declare and initialize a `public final int` field called `MAX_COUNT`. Assign the value 100 to this field.

   Hint: Use the keyword `final` to designate this is as a constant field.
3. Create a method called displayCount that does the following:
   - Counts from 1 to the value of the MAX_COUNT constant, using a `for` loop. Increment the value of the loop variable by 1.
   - Displays the value of the loop variable if it is divisible by 12. Display this on a single line, separated by a space.

   Hints
   - Example of a for loop:
     ```
     for (int i= 1; i < 10; i++) // loops 9 times
     ```
   - Use the modulus operator (%) to check divisibility by 12. If it is divisible by 12, the result of the modulus operation will be zero.
   - Use the `System.out.print` method to keep all displayed values on the same line.

   Solution:
   ```
   public void displayCount(){
       for(int count = 1; count <= MAX_COUNT; count++){
             if (count % 12 == 0) {
                   System.out.print(count + " ");
             } // end if
       } // end for
   } // end method
   ```
4. Save and compile your program. Test it by running the CounterTest class.
5. You should receive the following list of numbers as an output:
   ```
   12 24 36 48 60 72 84 96
   ```

Practice 9-2: Writing a Class that Uses a `while` Loop

Overview

In this practice you write a class named `Sequence` that displays a sequence starting with the numbers 0 and 1. Successive numbers in the sequence are the sum of the previous two numbers. For example: 0 1 1 2 3 5 8 13 21… This sequence is also called the Fibonacci series.

Assumptions

The `SequenceTest.java` file appears in the lab folder for this lesson, `D:\labs\les09` and consequently, in your project.

Tasks

1. Create a new Java class called "Sequence" with three fields called `firstNumber`, `secondNumber`, and `nextNumber`. Assign the values of 0 and 1 to the `firstNumber` and `secondNumber` fields, respectively. Also declare a `public final int` called SEQUENCE_LIMIT. Set its value to 100.
2. Create a method called `displaySequence`. Use the following high level steps to code the method. If you need more help, detailed instructions are provided following these steps:
   a. Print the value of `firstNumber`, `secondNumber` to start with the sequence. Separate all numbers in the sequence by a space.
   b. Calculate the sum of `firstNumber` and `secondNumber` and assign the sum to `nextNumber`.
   c. Create a `while` loop with the following characteristics:
      – *boolean expression*: Repeat if the value of `nextNumber` is less than or equal to SEQUENCE_LIMIT.
      – *code block*:
         – Print the value of `nextNumber`.
         – Assign the value of `secondNumber` to `firstNumber` and the value of `nextNumber` to `secondNumber`.
         – Recalculate the value of `nextNumber` to be the sum of `firstNumber` and `secondNumber`.
   d. After the while loop, use the System.out.println method to create a new line.

   Detailed Instructions:

   a. Before the `while` loop begins, use the `System.out.print` method to print `firstNumber` and `secondNumber`, concatenating a space to the end of each variable in your `print` statements.
   b. Set `nextNumber` equal to `firstNumber` + `secondNumber`.
   c. Start a `while` loop that evaluates the following expression in determining whether to loop again:

      ```
      while(nextNumber <= SEQUENCE_LIMIT)
      ```

      – Within the while block, do the following:
         – Print the `nextNumber` field. Add a space to the end of it.

- Set `firstNumber` equal to `secondNumber`, and `secondNumber` equal to `nextNumber`.
- Set `nextNumber` equal to `firstNumber` + `secondNumber`.

d. Outside the while loop block, use `System.out.println` to create a new line for the "Build Successful…" message that will appear after the sequence.

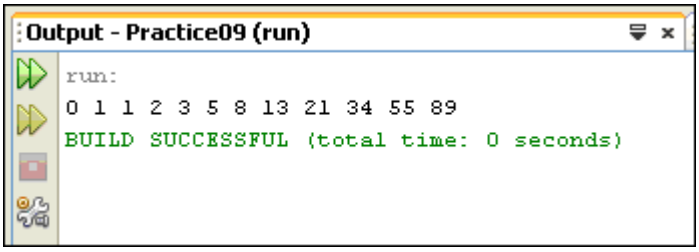
Solution:

```
public class Sequence{
     public int firstNumber = 0;
     public int secondNumber = 1;
     public int nextNumber;
     public final int SEQUENCE_LIMIT = 100;

     public void displaySequence(){
          // Print the first two numbers
          System.out.print(firstNumber + " ");
          System.out.print(secondNumber + " ");
          // Calculate the next number
          nextNumber = firstNumber + secondNumber;

          while(nextNumber <= SEQUENCE_LIMIT){
               // Print the next number of the sequence
               System.out.print(nextNumber + " ");
               firstNumber = secondNumber; // new first number
               secondNumber = nextNumber; // new second number
               // Calculate the next potential number
               nextNumber = firstNumber + secondNumber;
          } // end of while
          // Finish it off with a carriage return
          System.out.println();
     } // end of method
} // end of class
```

3. Save and compile your program. Run the `SequenceTest` class to test it.



Challenge Practice 9-3: Converting a `while` Loop to a `for` Loop

This practice is optional. Check with your instructor for recommendations about which optional labs to do. Only perform this if you are certain that you will have enough time to perform all of the non-optional labs.

Overview

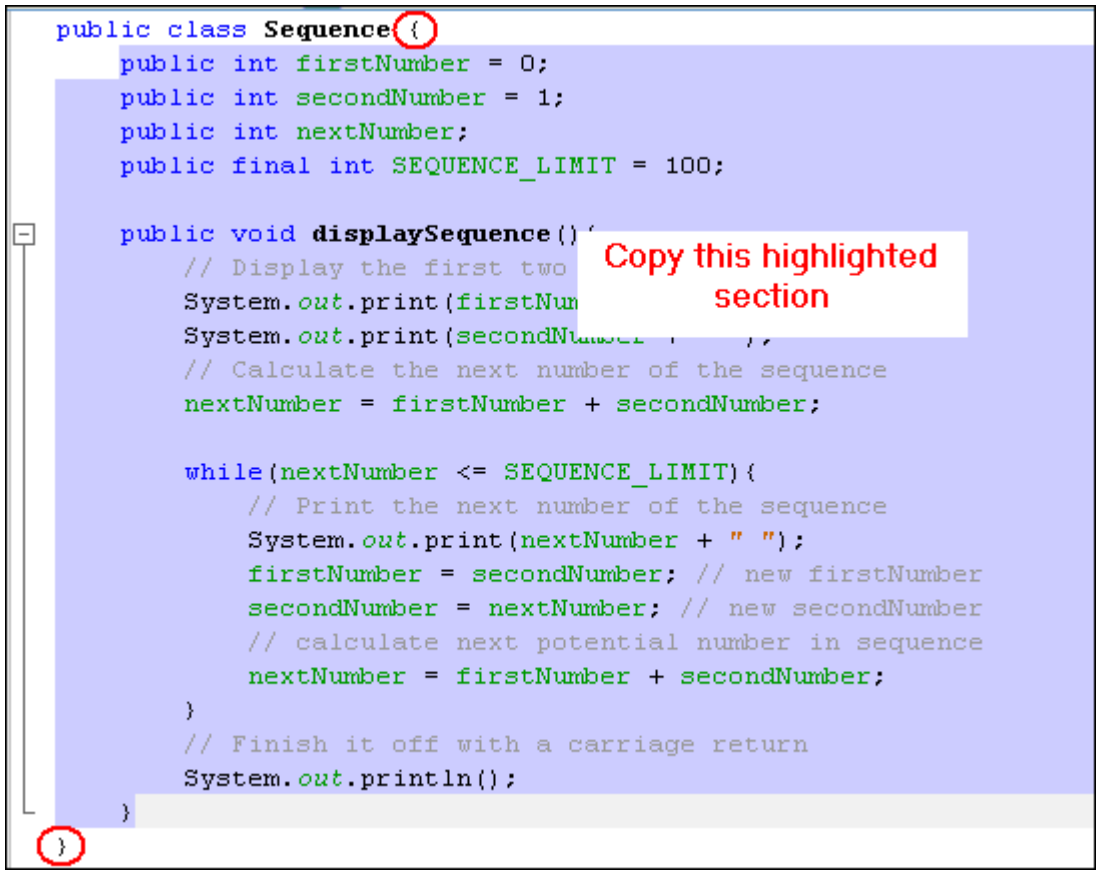
In this practice you create a new class, `ChallengeSequence`, based upon the `Sequence` class you created in the last practice. You modify the `displaySequence` method to use a `for` loop instead of a `while` loop.

Assumptions

This practice assumes that you have completed Practice 9-2. It also assumes that the ChallengeSequenceTest.java file appears in the lab folder for this lesson, `D:\labs\les09` and consequently, in your project.

Tasks

1. Create a new Java class called "ChallengeSequence". Copy all the code that occurs between the outer (class) brackets of the `Sequence` class and paste it inside the outer brackets of the `ChallengeSequence` class.




```
public class Sequence {
    public int firstNumber = 0;
    public int secondNumber = 1;
    public int nextNumber;
    public final int SEQUENCE_LIMIT = 100;

    public void displaySequence(){
        // Display the first two
        System.out.print(firstNum
        System.out.print(secondNu
        // Calculate the next number of the sequence
        nextNumber = firstNumber + secondNumber;

        while(nextNumber <= SEQUENCE_LIMIT){
            // Print the next number of the sequence
            System.out.print(nextNumber + " ");
            firstNumber = secondNumber; // new firstNumber
            secondNumber = nextNumber; // new secondNumber
            // calculate next potential number in sequence
            nextNumber = firstNumber + secondNumber;
        }
        // Finish it off with a carriage return
        System.out.println();
    }
}
```

2. Create an additional final field called `SEQUENCE_COUNT` and assign a value of 10 to it. Be sure that you don't change any of the other field names.
3. In the `displaySequence` method, modify the `while` loop to a `for` loop such that only the first 10 values of the fibonacci series are displayed.

Hints

- Remember that the first two numbers in the sequence are displayed before the loop begins. Your `for` loop must display the remaining 8 values.
- There are a several ways of handling the discrepancy between the `SEQUENCE_COUNT` value and the number of values that need to be displayed within the loop. One approach is to adjust the initial count in the loop.

One Possible Solution:

```
public void displaySequence(){
     System.out.print(firstNumber + " ");
     System.out.print(secondNumber + " ");
     nextNumber = firstNumber + secondNumber;

     for(int count = 2; count < SEQUENCE_COUNT; count++){
     // Start at 2 and loop until you get to 9 (8 numbers)
          System.out.print(nextNumber + " ");
          firstNumber = secondNumber;
          secondNumber = nextNumber;
          nextNumber = firstNumber + secondNumber;
     }
     System.out.println();
}
```

4. Save and compile your program. Run the `ChallengeSequenceTest` class to test your code. Your output should display the following series:

```
0 1 1 2 3 5 8 13 21 34
```

Practice 9-4: Using `for` Loops to Process an ArrayList

Overview

In this practice you create two new methods in two different classes. One of the methods uses a traditional `for` loop to display the values in an `ArrayList`. The other method uses an *enhanced* `for` loop to display the values in the `ArrayList`. This practice contains two sections:

- Using a `for` Loop with the `VacationScaleTwo` Class
- Using an Enhanced `for` loop with the `NamesListTwo` Class

Assumptions

This practice assumes that the following files appear in the lab folder for this lesson, `D:\labs\les09` and consequently, in your project:

- VacationScaleTwo.java
- VacationScaleTwoTest.java
- NamesListTwo.java

Using a `for` Loop with the VacationScaleTwo Class

1. Open the `VacationScaleTwo` class in the editor. This is similar to the `VacationScale` class you wrote in Lesson 8, but an `ArrayList` is used to store vacation days instead of an array in this case.
2. Add a new method called `displayVacationDays`. High level instructions for this task are provided in the table below. More detailed instructions can be found following the table.

Step	Code Description	Choices or Values
a.	Use a `for` loop to loop through the elements of the `vacationDays` ArrayList.	Use the `size` method of the `ArrayList` in the boolean expression that determines the end of the loop.
b.	Within the loop, display each value of the `ArrayList` and its position in the list with a suitable label.	Use the `get` method of the `ArrayList` together with `System.out.println` to display the value.

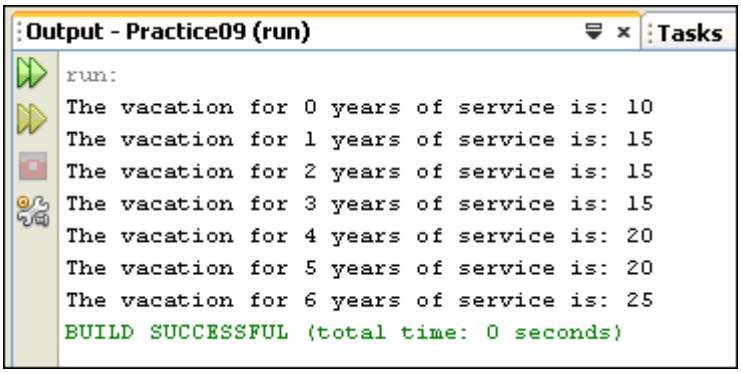
   a. In the `displayVacationDays` method, add a `for` loop with the following criteria:

```
for(int years = 0; years < vacationDays.size(); years++)
```

   b. In the `for` loop block, use `System.out.println` to print the value of each `ArrayList` element. Use the `get` method of the `ArrayList`, passing the `years` variable as an argument. It references the current index number of the `vacationDays` list.

```
System.out.println("The vacation for " + years +
 " years of service is: " + vacationDays.get(years));
```

3. Save and compile your program, then run the `VacationScaleTwoTest` class to test it. You should see an output similar to this:



Using an Enhanced `for` Loop with the NamesListTwo Class

4. Open the `NamesListTwo` class in the editor. This is similar to the `NamesList` class that you saw in Lesson 8. It has only one method, `setList`, that initializes the `ArrayList` and then prints the size of the list.
5. Add a new method to the `NamesListTwo` class called `displayNames`. You will use an *enhanced* `for` loop in this method to process the `ArrayList`. High level instructions for this task are provided in the table below. More detailed instructions can be found following the table.

Step	Code Description	Choices or Values
a.	Display an introductory message to describe the list that will follow.	
b.	Start the enhanced `for` loop (remember that an `ArrayList` is defined to hold elements of type `Object`)	`for (Object name : names)`
c.	In the `for` block, display the current element of the `ArrayList`.	`Use the name reference`

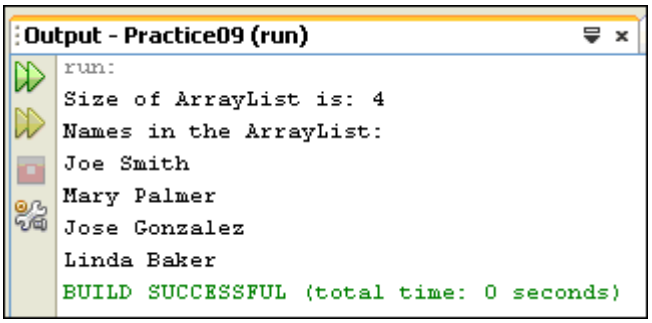
   a. Use the `System.out.println` method to print the message:
      `"Names in the ArrayList: "`
   b. Start the enhanced `for` loop as follows:
      `for (Object name : names)`

      Note: The `name` variable will be a reference to the current element in the `names` `ArrayList` for each iteration of the `for` loop.
   c. Within the `for` loop block use system.out.println to print the `name` reference.

Solution:

```
public void displayNames(){
     System.out.println("Names in the ArrayList: ");
     for(Object name : names){
          System.out.println(name);
     }
}
```

6. Create a new Java Main Class called `NamesListTwoTest`.
7. In the `main` method, do the following:
   a. Declare and initialize a local variable of type `NamesListTwo` called `namesList`.
      `NamesListTwo namesList = new NamesListTwo();`
   b. Invoke the `setList` method of the `namesList` object.
   c. Invoke the `displayNames` method of the `namesList` object.
8. Save and compile your program. Run the `NamesListTwoTest` class to test it.
9. You should see an output from the program similar to the screenshot below:



Practice 9-5: Writing a Class that Uses a Nested `for` Loop to Process a Two Dimensional Array

Overview

In this practice you create and process a two-dimensional array using a nested `for` loop (one loop within another loop). This practice is based on the scenario of a classroom. A classroom has 12 desks arranged in a rectangular grid comprised of three rows and four columns. Students are allocated a desk at the position found vacant first, by traversing each row.

The following table shows the class map as a grid. Each cell represents a desk. Each cell contains the coordinates of the desk position in the class map.

XXXX	Column 1	Column 2	Column 3	Column 4
Row 1	0,0	0,1	0,2	0,3
Row 2	1,0	1,1	1,2	1,3
Row 3	2,0	2,1	2,2	2,3

Assumptions

This practice assumes that the `ClassMapTest.java` file appears in the lab folder for this lesson, `D:\labs\les09` and consequently, in your project.

Tasks

1. Create a new Java class called "ClassMap" .
2. In the class, declare two public fields as follows:

```
public String[][] deskArray;
public String name;
```

3. Create a new method called `setClassMap`. In this method, initialize the `deskArray` to have three rows and four columns:

```
deskArray = new String[3][4];
```

4. Create another new method called `setDesk`. This method will assign a new student (identified by the `name` field which will be set by the `ClassMapTest`) to an empty desk in the class map. Define the method according to the steps below:
   a. Traverse the `deskArray` to identify the first vacant element in it. Use a nested `for` loop for this purpose. For example:

```
for(int row=0; row<3; row++){
    for(int col=0; col<4; col++){
        if(deskArray[row][col]==null){
```

   b. If you find a `null` value in the `deskMap` (in other words, if you find an empty desk), assign the value of the `name` field to the vacant element.
   c. Print the position of the desk for the student and exit out of the loops. Use a `break` statement to branch out of a running loop.

Solution:

```
public void setDesk() {
     boolean flag= false;
     for(int row=0; row<3; row++){ // start of row loop
        for(int col=0; col<4; col++){ // start of column loop
          if(deskArray[row][col]==null){
             deskArray[row][col] = name;
             System.out.println
                (name +" desk is at desk set at position: Row:"
                  + row + ", Column:"+col);
             flag = true;
             break; // drop out of column loop
          } // end of if
        } // end of inner/column for loop
        if (flag == true){
          break; // drop out of row loop
        } // end of if
     } // end of row for loop
     if (flag == false){
        System.out.println("All desks occupied.");
     } // end of if
} // end of method
```

Note: You will test this code a little later.

5. Create another new method called `displayDeskMap`. In this method, traverse the `deskArray` in the same way you did in the last step. For each element in the array, print the name in that element (or print "null"). The output should be in the form of grid.

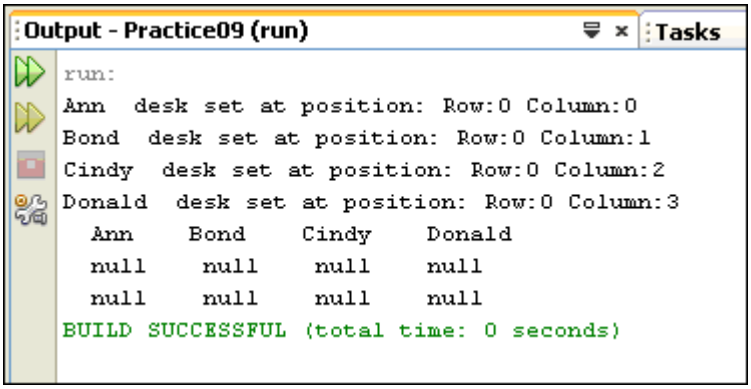
   Hint: Use a combination of `print` and `println` method calls to achieve the grid format. The grid should look similar to this:

   Ann Bond Cindy Donald
   null null null null
   null null null null

Solution:

```
public void displayDeskMap() {
     for(int row=0; row<3; row++){
        for(int col=0; col<4; col++){
            System.out.print(" "+ deskArray[row][col] +" ");
        }
        System.out.println(); // carriage return between rows
     }
}
```

6. Save and compile your program.
7. Open the `ClassMapTest` class and examine the code in the `main` method. It first calls `setClassMap` to initialize the array. Next it assigns a value to the `name` field of the `myClassMap` object and then invokes `setDesk`. It does this four times, with a different `name` value each time. Finally, it invokes `displayDeskMap`.
8. Run the `ClassMapTest` class to test your program.



9. If you do not plan to perform the Practice 9-6 (an optional challenge practice), please close the Practice09 project in NetBeans now.

Challenge Practice 9-6: Adding a Search Method to the ClassMap Program

This practice is optional. Check with your instructor for recommendations about which optional labs to do.

Overview

In this practice you add another method to the `ClassMap` class. This method searches through the `deskArray` to find a certain name.

Assumptions

This practice assumes that you have completed Practice 9-5.

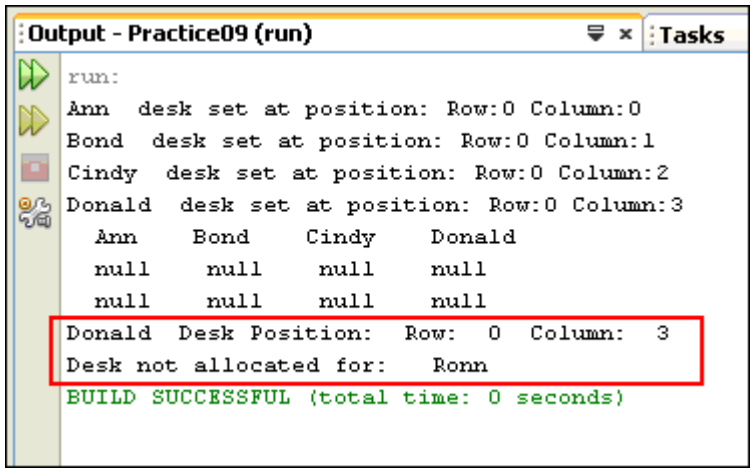
Tasks

1. In the `ClassMap` class, add another new method called `searchDesk`.
2. In the `searchDesk` method, do the following:
   a. Create a nested `for` loop to traverse through the `deskArray`.
   b. If the array element is not `null`, compare the value of the `name` field with the value of the element. For example:
   ```
   if(deskArray[row][col] != null &&
        deskArray[row][col].equals(name)){
   ```
   c. Print the position of the desk if the names are equal.
   d. Print an error message if the name is not found in the `deskArray`.
   e. Use the `break` statement to branch or exit out of the loops wherever required.

Solution:

```
public void searchDesk() {
     boolean flag= false;
     for(int row=0; row<3; row++){
        for(int col=0; col<4; col++){
          if(deskArray[row][col] != null &&
             deskArray[row][col].equals(name)){
                System.out.println
                (name +" Desk Position: Row: "+row+" Column: "
                   +col);
             flag = true;
             break;
          } // end of if
        } // end of column loop
        if (flag == true){
          break;
        } // end of if
     } // end of row loop
     if (flag == false){
        System.out.println("Desk not allocated for "+name);
     } // end of if
} // end of method
```

3. In the `ClassMapTest` class, uncomment the lines of code that set the `name` value of `myClassMap` object and invoke its `searchDesk` method (this combination occurs twice).
4. Save and compile your program. Run the `ClassMapTest` class to test the program.



5. This is the conclusion of lesson 9 practices. Please close the Practice09 project now.

Practices for Lesson 10: Working With Methods and Method Overloading

Chapter 10

Practices for Lesson 10

Practices Overview

In these practices, you will create and use methods with arguments. In a challenge exercise, you create an overloaded method. Solutions for these practices can be found in `D:\labs\soln\les10`.

Practice 10-1: Writing a Method that Uses Arguments and Return Values

Overview

In this practice you create a class to order more than one shirt and then display the total order value of the shirts.

Assumptions

This practice assumes that the following files appear in the lab folder for this lesson, `D:\labs\les10`:

- Order.java
- Shirt.java

Tasks

1. Create a new project from existing source called Practice10. Set the **Source Package Folder** to point to `D:\labs\les10`. Remember to also change the Source/Binary Format property. If you need further details, refer to Practice 2-2, Steps 3 and 4.
2. Open the Shirt class and examine the member fields and the method it contains.
3. Open the Order class and examine its member fields and method.
4. Create a new Java Main Class called "OrderTest".
5. Add code to the main method that will add a shirt to a new order and display the total amount of the order. The high level steps for this are provided in the table below. More detailed assistance is given following the table.

Step	Code Description	Choices or Values
a.	Create and initialize two objects	Object type: `Shirt` Object type: `Order`
b.	Declare and initialize a local variable of type double	Variable Name: `totalCost` Value: 0.0
c.	Assign a value to the `price` field of the `Shirt` object	Value: 14.99
d.	Invoke the `addShirt` method of the `Order` object	Method argument: the `Shirt` object Method return: assign to `totalCost`
e.	Display the return value with a suitable label	Example output: "Total amount for the order is $14.00"

Further Details:

The documentation for the `addShirt` method is as follows:

Adds a shirt to a list of shirts in an order

Syntax:

```
public double addShirt (Shirt s)
```

Parameters:

s – An object reference to a Shirt object

Returns:

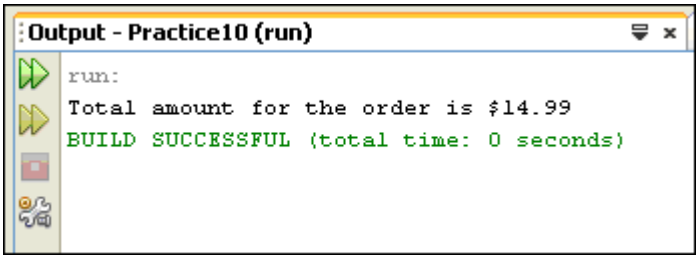
A total current amount for the order

Solution:

```
public static void main(String[] args){
    Order order = new Order();
    Shirt shirt = new Shirt();
    double totalCost = 0.0;

    shirt.price = 14.99;
    totalCost = order.addShirt(shirt);
    System.out.println("Total amount for the order is $" +
        totalCost);
}
```

6. Save and compile your program. Test the order process by running the OrderTest class.



7. In the `main` method of OrderTest, create additional `Shirt` objects, assign values to the `price` field of each new `Shirt` object, and add the `Shirt` objects to your order by invoking the `addShirt` method.

   Note: Remember that the `addShirt` method adds the price of the shirt argument object to the `totalPrice` field of the `Order` object. Therefore the `totalPrice` value grows with each addition of a shirt. You only need to capture the return value of the final `addShirt` method call to get the `totalCost` value.

Solution:

```
public static void main(String[] args) {
        Order order = new Order();
        Shirt shirt = new Shirt(),
              shirt2 = new Shirt(),
              shirt3 = new Shirt();
        double totalCost = 0.0;

        shirt.price = 14.99;
        shirt2.price = 23.55;
        shirt3.price = 49.99;
        order.addShirt(shirt);
        order.addShirt(shirt2);
        totalCost = order.addShirt(shirt3);
        System.out.println("Total amount for the order is $" +
    totalCost);
}
```

8. Save and compile the program and once again, run the OrderTest class to test it. Make sure that the amount displayed is the total of all of the shirt prices.

Challenge Practice 10-2: Writing a Class that Contains an Overloaded Method

This practice is optional. Check with your instructor for recommendations about which optional labs to do. Only perform this if you are certain that you will have enough time to perform all of the non-optional labs.

Overview

In this practice you write a Customer class with an overloaded method called setCustomerInfo.

Assumptions

This practice assumes that the CustomerTest.java file appears in the lab folder for this lesson, `D:\labs\les10`, and consequently also in the Practice10 project.

Tasks

1. Create a new Java Class called "Customer". Declare the following fields and initialize them as shown in the table below:

Field	Type	Initial Value
`customerID`	int	`0`
`name`	`String`	`"-name required-"`
`address`	`String`	`"-address required-"`
`phoneNumber`	`String`	`"-phone required-"`
`eMail`	`String`	`"-email required-"`

2. **High level instructions:** Within the Customer class, add two overloaded methods called `setCustomerInfo`. Depending upon how the `setCustomerInfo` method is called, it does one of the following:
   - Sets the ID, name, address, and phone number for a Customer object. (This is the minimum information needed for a new customer.)
   - Sets the ID, name, address, phone number, and email address for a Customer object.

   Detailed Instructions:
   - The two method signatures should be as follows:

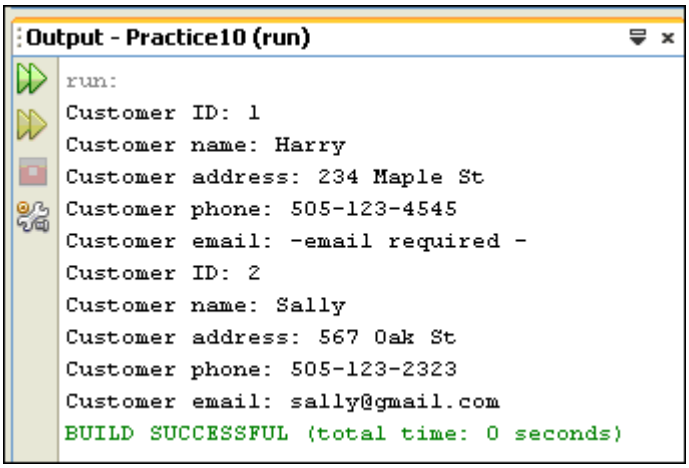
     ```
     public void setCustomerInfo(int Id, String nm, String addr,
         String phNum)
     public void setCustomerInfo(int Id, String nm, String addr,
         String phNum, String email)
     ```
   - Within each method, assign each argument in the method to its corresponding field.
3. Create a `display` method to display the values of all the member fields of the Customer class.
4. Save and compile the program.

5. Open the CustomerTest class. Modify the `main` method as follows so that it can be used to test the overloaded methods of the Customer class.
   - Create two object references to different Customer objects.
   - Use each variation of the `setCustomerInfo` method to provide information for each Customer object.
   - Display the contents of each Customer object.

   Solution:

```
public static void main(String[] args){
     Customer c1 = new Customer(),
              c2 = new Customer();
     c1.setCustomerInfo(1,"Harry","234 Maple St",
               "505-123-4545");
     c2.setCustomerInfo(2,"Sally","567 Oak St",
               "505-123-2323","sally@gmail.com");
     c1.display();
     c2.display();
}
```

6. Once again, save and compile the program. Run the CustomerTest file to test the program. Make sure that the output is different for each of the display methods.



Practices for Lesson 11: Using Encapsulation and Constructors

Chapter 11

Practices for Lesson 11

Practices Overview

In these practices, you will experiment with field access and encapsulation, and create and use overloaded constructors. A challenge practice is included here for those of you who have extra time and wish to be challenged. Solutions for these practices can be found in `D:\labs\soln\les11`.

Practice 11-1: Implementing Encapsulation in a Class

Overview

In this practice you create a class containing private attributes and try to access them in another class. This practice has two sections.

- Implementing encapsulation in a class
- Accessing encapsulated attributes of a class

Assumptions

This practice assumes that the following files appear in the lab folder for this lesson, `D:\labs\les11`, and consequently also in the Practice11 project.

- DateOneTest.java
- DateTwoTest.java
- DateThreeTest.java

Implementing Encapsulation in a Class

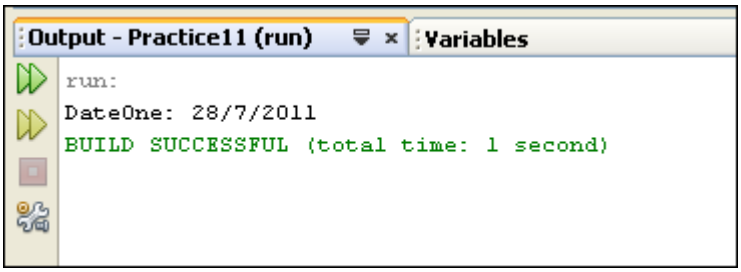
1. Create a new project called "Practice11". Refer to the Practice 2-2, Steps 3 and 4 if you need assistance.
2. Create a new Java Class called "DateOne". Declare three member fields of type `int` named: `day`, `month`, and `year`. Give public access to all the member fields.
3. Open the DateOneTest class. In the `main` method, do the following:
   a. Create and initialize an object of type `DateOne`.
   b. Assign different numeric values to the member fields of the `DateOne` object.
   c. Display the value of the member fields of the `DateOne` object. Concatenate them into a single String with your choice of date formatting.

   Note: The back slash (\) character is a special character in the Java language called an "escape character". If you wish to use it as part of your date format, use two back slashes together in order to have the back slash appear in the String. Example: day + "\\" + month. There are no restrictions for using a forward slash.

   Solution:

```
public static void main(String args[]){
     DateOne date = new DateOne();
     date.day = 28;
     date.month = 07;
     date.year = 2011;
     System.out.println("DateOne: "+date.day+ "/" +date.month+
          "/" +date.year);
}
```

4. Save and compile your program.
5. Run the DateOneTest class to test the program.



6. Create another new Java Class called “DateTwo” similar to DateOne with three member fields (`day`, `month`, `year`).
7. Set the access modifier for the member fields to `private`.
8. Open the DateTwoTest class and perform the same steps as in Step 2, however in this case, create an instance of the DateTwo class instead of the DateOne class. The other lines of code remain the same.

   Note: NetBeans warns you with an error icon next to each line that references the fields of the DateTwo object.

```
1   public class DateTwoTest {
2
3     public static void main(String args[]) {
4         DateTwo date = new DateTwo();
          date.day = 28;
          date.month = 07;
          date.year = 2011;
```

   Examine the warning message by putting your cursor over any of the red icons. It says that “`day` has private access in DateTwo” (similar message for each field). While NetBeans will let you click Save without issuing a compiler error, it only saved the file. It did not compile the code or create the `DateTwoTest`.class file.

Accessing Encapsulated Attributes of a Class

In this task, you create a class with private attributes but enable them to be manipulated from another class.

9. Create a new Java Class called “DateThree” and add the same three private fields as the `DateTwo` class.
10. Add a `public get` method for the day field. This method should return an `int`. In the body of the method, return the day field. Example:

```
public int getDay(){
   return day;
}
```

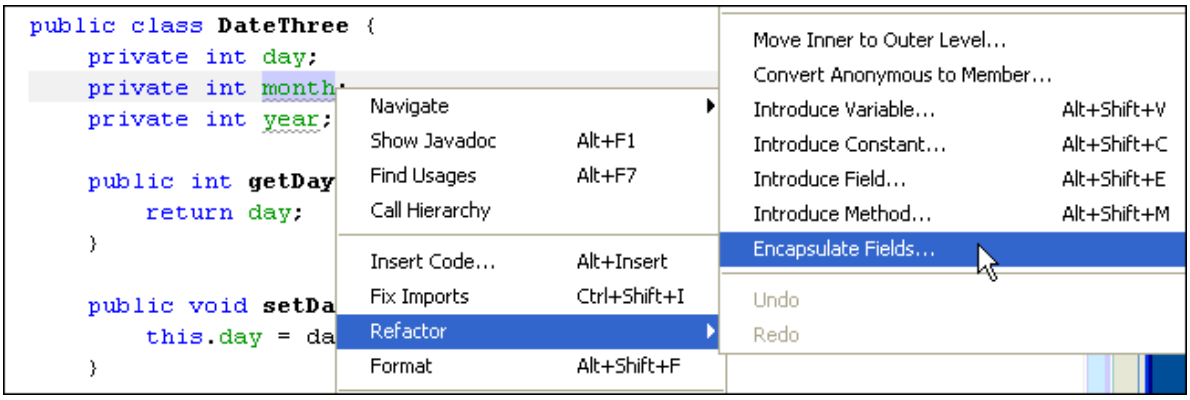
11. Add a `public set` method for the `day` fields. This method returns `void` but takes an argument of type `int`. In the body of the method assign the argument to the `day` field. Example:

```
public void setDay(int d){
     day = d;
}
```

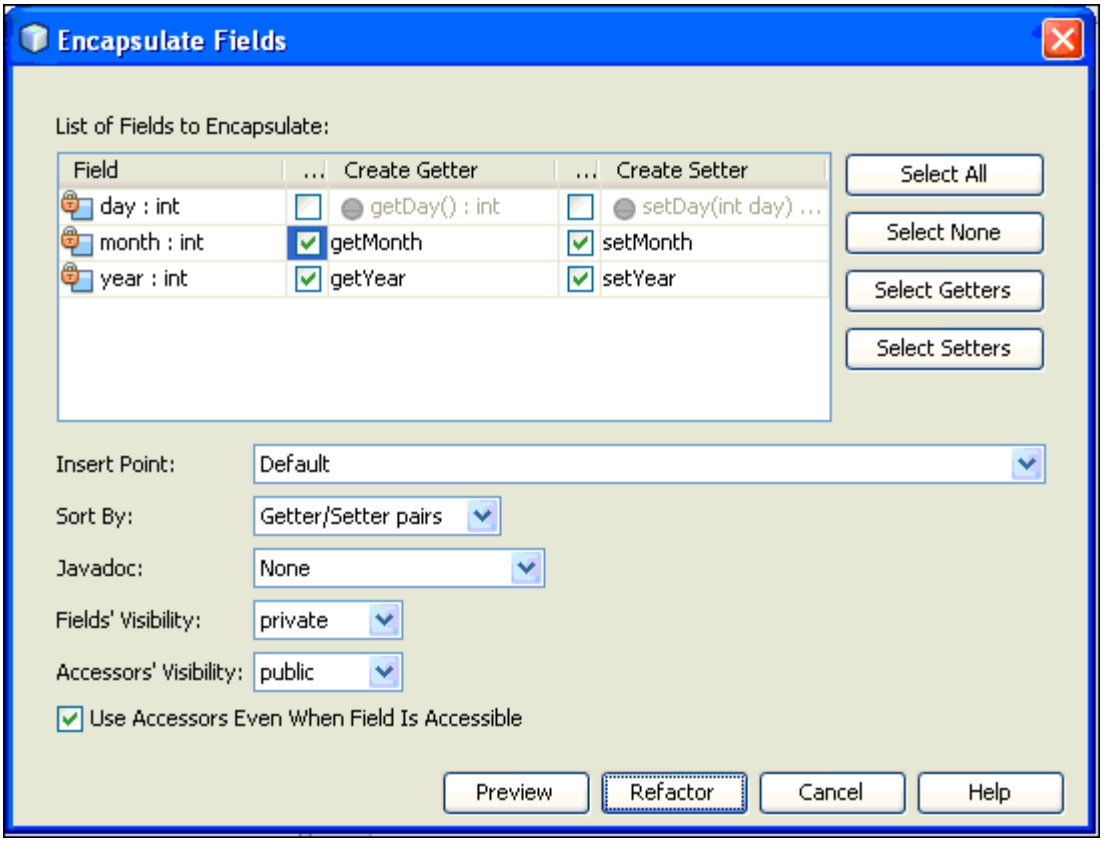
12. Add a similar `get` and `set` method for both the `month` and the `year` fields. Read the Hint below first to save some time.

Hint

Most IDEs will automatically create the `get` and `set` methods for private fields in a class. This is part of feature called "Refactoring". In NetBeans, you can take advantage of this feature by selecting (highlighting) one of the private fields and right-clicking it. Select **Refactor > Encapsulate Fields** from the context menu.



The **Encapsulate Fields** window opens. Select the get and set method check boxes for the remaining fields. You may wish to also set the **Javadoc** setting to **None** in order to streamline your method code. Click **Refactor** to close the window and create the methods.



```
public int getMonth() {
    return month;
}

public void setMonth(int month) {
    this.month = month;
}

public int getYear() {
    return year;
}

public void setYear(int year) {
    this.year = year;
}
```

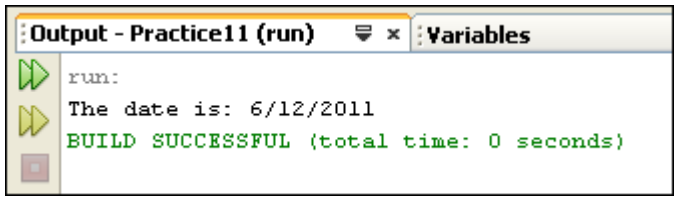
13. Open the DateThreeTest class.
14. In the `main` method, declare and an object of type DateThree called "dateObj". Create an instance of the DateThree class.
15. Using the `DateThree` object reference, invoke the `setMonth`, `setDay`, and `setYear` methods of the `DateThree` object to set the three values of a date. Example:

```
dateObj.setMonth(6);
dateObj.setDay(12);
dateObj.setYear(2011);
```

16. Complete the `main` method by displaying the entire date in the format of your choice. For example:

```
System.out.println(The date is :"+ dateObj.getMonth() +
  "/" + dateObj.getDay() + "/" + dateObj.getYear());
```

17. Save and compile your program. Run the DateThreeTest class to test it.



Challenge Practice 11-2: Adding Validation to the DateThree Class

This practice is optional. Check with your instructor for recommendations about which optional labs to do. Only perform this if you are certain that you will have enough time to perform all of the non-optional labs.

Overview

In this practice you add a `setDate` method to the DateThree class that performs validation on the date part values that are passed into the method.

Assumptions

This practice assumes that you have finished Practice 11-1.

Tasks

1. In the DateThree class, add a public `setDate` method that takes three arguments of type `int`. These values will be assigned to the `day`, `month`, and `year` fields respectively. For example:

   ```
   public void setDate(int d, int m, int y)
   ```

   Perform the validation indicated in the table below before assigning the argument values to the fields. Detailed steps are provided after the table.

Step	Code Description	Choices or Values
a.	Valid values for the `year` field	Between 1000 and 10000
b.	Valid values for the `month` field	1 – 12
c.	Valid values for the `day` field	30, 31, 28, 29 Depends on the month

   Note

   Use a `switch case` statement to determine the `month`. Use `if/else` statements to perform the validation. Display an error message if the value is invalid.

   a. In the `setDate` method, add the following `if / else` statement to check the validity of the year argument. Note: The `year` field is set to 0 in the case of an invalid year argument. You will check for a 0 year value later.

   ```
   if (y > 1000 && y < 10000){
       this.year = y;
   } else {
       System.out.println(y + " is not a valid year.");
       this.year = 0;
   }
   ```

   b. Create a `switch` statement that evaluates the month argument. Months 1, 3, 5, 7, 8, 10, and 12 have 31 days. Check for these values first in the `switch` statement. If the month argument equals any of these cases, assign the month argument to the `month` field, then include an `if / else` statement to test the value of the day argument. It should be between 1 and 31. If it is not, display an error message and set the `day` field to 0.

Example:

```
switch (m) {
     case 1:
     case 3:
     case 5:
     case 7:
     case 8:
     case 10:
     case 12:
          this.month = m;
          if (d > 0 && < 32){
             this.day = d;
          } else {
            System.out.println(d + " is an invalid day for "
            + m);
            this.day = 0;
          }
          break;
//(switch statement continues in step c)
```

c. Use the following logic to complete the `switch` statement. In the `case block` for the month of February (case 2), you must also test for a leap year if the day argument is 29. The logic for rest of the months is similar to what you wrote for months containing 31 days.

```
...
case 2:
     this.month = m;
     if(d > 0 && d < 29) {
        this.day = d;
     } // check if year is a leap year when d=29 and m=2
     else if (d == 29){
        if(((y%4 == 0) && !(y%100 == 0)) || (y%400 == 0)){
           this.day = d;
        } else {
           System.out.println("Invalid day. " +
            "Day cannot be 29 unless the year is a leap year.");
        }
     break;
case 4:
case 6:
case 9:
case 11:
     this.month = m;
     if(d > 0 && d < 31){
        this.day = d;
     } else {
        System.out.println("Invalid day. Must be 1 to 30.");
        this.day = 0;
     }
     break;
default:
     System.out.println(m + " is an invalid month.");
     this.month = 0;
     break;
} // end switch
```

2. Add one more method called `displayDate`. In this method, first check for values of (zero) in `day`, `month`, or `year`. If any of these has a 0 value, print an "Invalid date" message. Otherwise, display the date using a date format of your choice. Example:

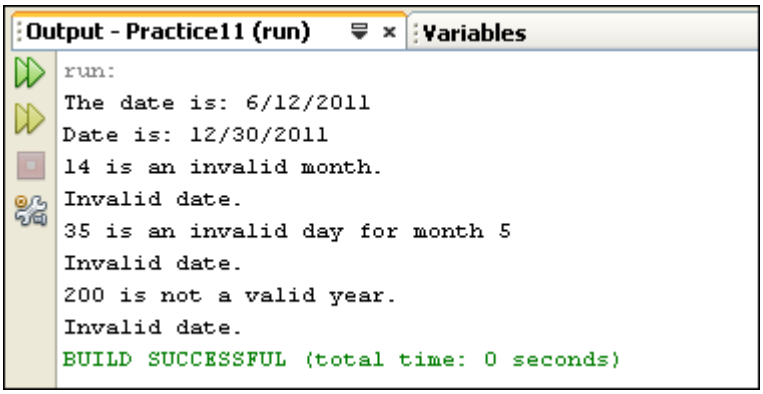
```
public void displayDate(){
     if(day == 0 || month == 0 || year == 0){
          System.out.println("Invalid date.");
     }
     else {
          System.out.println("Date is: " + month +"/"+
             day +"/"+ year);
     }
}
```

3. Open the DateThreeTest class and, using the `setDate` and `displayDate` methods, write code to perform the following tests:
   - Test with valid values for month, day and year
   - Test with invalid value for month 14
   - Test with invalid value for day 35
   - Test with invalid year 200

   Example for the first test:

```
dateObj.setDate(30,12,2011);
dateObj.displayDate();
```

4. Save and compile your program and run the DateThreeTest class. You should see an output similar to the following:



Compare the output to your code in order to match up the messages with the particular test that was run.

Practice 11-3: Creating Constructors to Initialize Objects

Overview

In this practice you create a class and use constructors to initialize objects.

Assumptions

This practice assumes that the RectangleTest.java file appears in the lab folder for this lesson, `D:\labs\les11`, and consequently also in the Practice11 project.

Tasks

1. Create a new Java Class called "Rectangle". Add two `private` fields of type `int` and name them `width` and `height`.
2. Add a constructor with no arguments (a "no args constructor"). The following table provides the high level steps to create this constructor. If you need more help, use the detailed instructions below the table.

Step	Description	Choices or Values
a.	Syntax for declaring a no args constructor	`public <class_name>()`
b.	Print a message	"Default rectangle created: width = 25, height = 10"
c.	Initialize the private fields	`width = 25` `height = 10`

   a. In the Rectangle class, declare a `public` no args constructor as follows:
   ```
   public Rectangle(){
   }
   ```
   b. Use `System.out.println` to display the message shown in Step b of the table above.
   c. Assign the `width` field to the value 25 and the `height` field to the value 10.
3. Add a second constructor that accepts two `int` arguments: `w` and `h` (for "width" and "height"). The following table provides the high level steps to complete this constructor. If you need more help, use the detailed instructions below the table.

Step	Code Description	Choices or Values
a.	Set height to `h` and width to `w` after validating the argument values	`h` and `w` should be > 0 and < 30
b.	Display a message for each condition	Error message if the numbers are invalid Message indicating that a rectangle has been created (show the height and width)

   a. In the constructor, add an `if / else` statement to ensure that the values passed into the constructor are within the acceptable range of 1 through 29. If both arguments are valid, assign the argument to its respective member field.
   b. After assigning the values, print a message that indicates that a rectangle has been created with the designated values. Include the `width` and `height` values in the message. If the argument values are not valid, display an error message.

Solution:

```
public Rectangle(int w, int h){
     if(w > 0 && w < 30 && h > 0 && h < 30){
          width = w;
          height = h;
          System.out.println("Rectangle created: width = "
            +width+ " and height = "+height);
     }
     else {
          System.out.println("Invalid width and/or height. "+
             "Each must be positive and less than 30.");
     }
}
```

4. Create a `getArea` method that calculates and returns the area of the rectangle (width * height).

Solution:

```
public int getArea(){
     return width * height;
}
```

5. Create a `draw` method that prints the rectangle shape, as determined by its width and height, in a series of rows containing asterisks (*). The following steps provide more detailed instructions:
   a. Create a nested `for` loop to draw the rectangle using asterisks.
   b. The outer `for` loop iterates through the rows of the rectangle. The number of rows corresponds to the value of the `height` field.
   c. The inner `for` loop iterates through the columns of each row. The number of columns corresponds to the value of the `width` field.

Solution:

```
public void draw(){
     for(int rowCounter=0;rowCounter<height;rowCounter++){
          for(int colCounter=0;colCounter<width;colCounter++){
             System.out.print("*");
          } // end of each row
          System.out.println();  // create a new line
     } // end of all rows
} // end of draw method
```

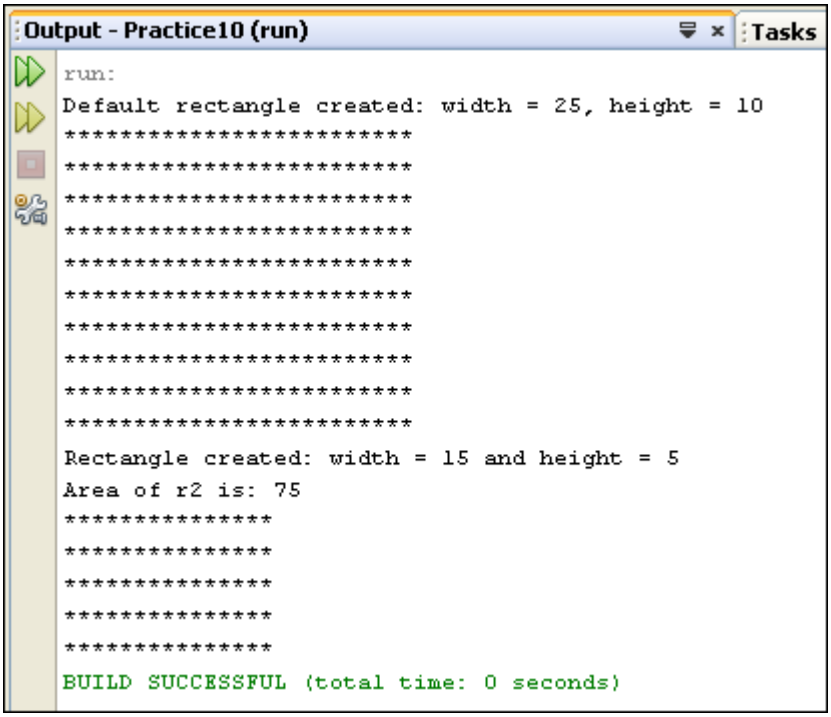
6. Save and compile your program.
7. Open the RectangleTest class. In the `main` method, declare and create two Rectangle objects, `r1` and `r2`, such that:
   - `r1` is created with the no args constructor
   - `r1` is drawn immediately after it is created (use the draw method)

- `r2` is created using the constructor with arguments
- `r2` is drawn and the area is printed

Solution:

```
public static void main(String args[]){
     // Rectangle with default values (no args)
     Rectangle r1 = new Rectangle();
     r1.draw();
     //Rectangle from args constructor
     Rectangle r2 = new Rectangle(15,5);
     System.out.println("Area of r2 is: "+r2.getArea());
     r2.draw();
}
```

8. Save and compile your program. Run the RectangleTest class to test it. The output should look similar to this:



Practices for Lesson 12: Describing Advanced Object-Oriented Concepts

Chapter 12

Practices for Lesson 12

Practices Overview

In these practices, you will design and create a class hierarchy for the Employee Tracking System of the Marketing department of Duke's Choice Company. You will also create an interface and implement it in the classes you created. Solutions for these practices can be found in `D:\labs\soln\les12`.

Practice 12-1: Creating and Using Superclasses and Subclasses

Overview

In this practice you design and then create a class hierarchy that will form the basis for an Employee Tracking System of the Marketing department in the Duke's Choice Company. This practice comprises two sections. In the first section, you create a simple design model for the class hierarchy. In the second section, you create the actual classes and test them.

Assumptions

This practice assumes that the following files appear in the lab folder for this lesson, `D:\labs\les12:`

- EmployeeTest.java

Design the Class Hierarchy

In this section, you design subclasses and superclasses using the information in the following paragraphs.

The Marketing department of the Duke's Choice Company has employees in several different positions. Some of these positions are: Technical Writers, Graphic Illustrators, Managers, and Editors.

Marketing wants you to create a program for tracking information about each of its workers. This information consists of: the worker's name, job title, employee ID and level (1, 2, or 3). Additionally:

- Managers must have a list of employees that they manage.
- Technical Writers, Graphic Illustrators, and Editors must have a list of skills that they possess.
- Editors must have a value indicating whether they prefer to do electronic editing or paper-based editing.
- There must be a means by which to display all the information for a given employee type.

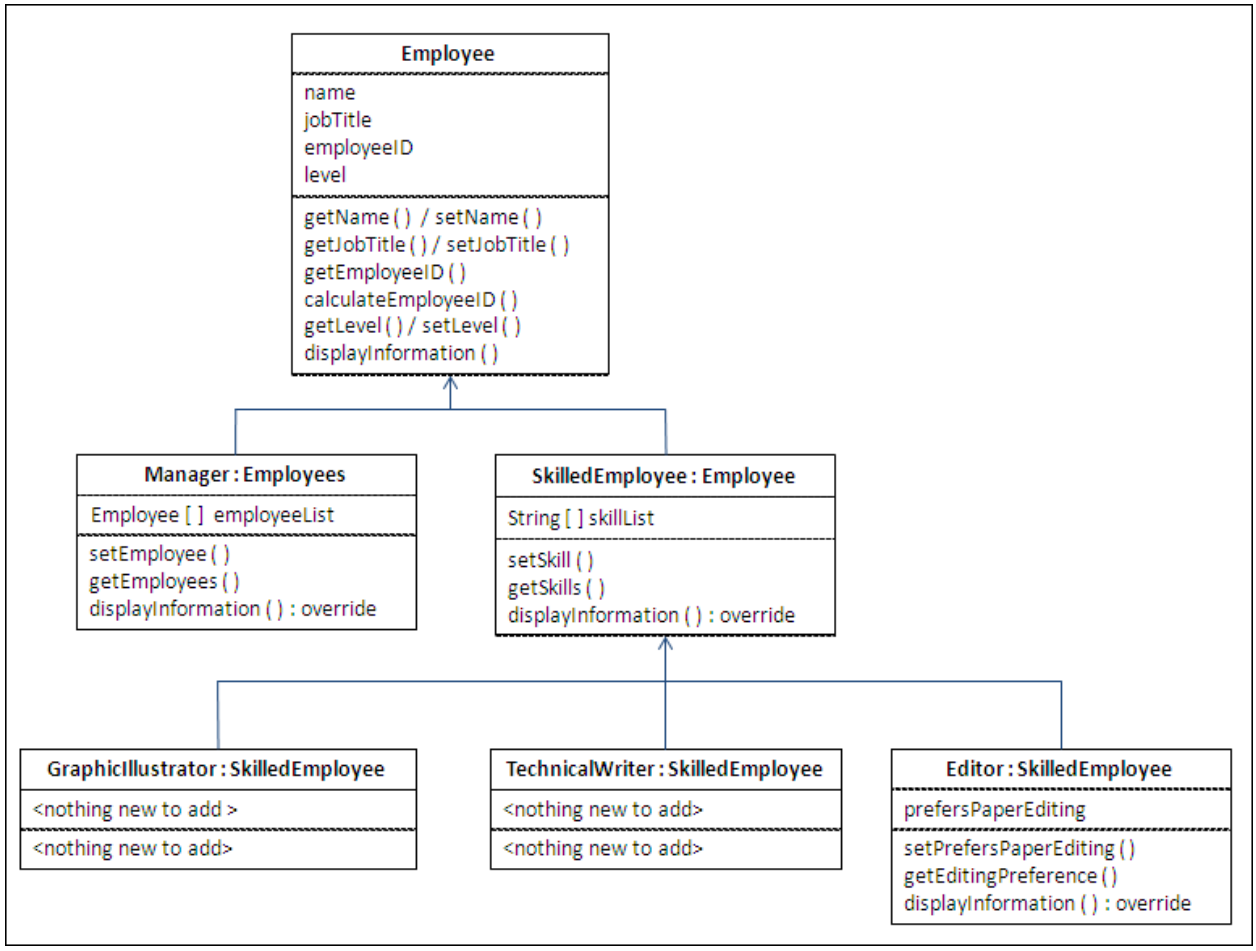
1. Create a class hierarchy of superclass and subclass relationships for the employees of the Marketing department. Draw the diagram on a piece of paper. Or, if you prefer, you may use the UMLet tool on your desktop.

   Hints

   - **Use the "is a" phrase –** Ask yourself if all or many of the job types have some of the same attributes (fields) and operations (methods). For instance, all of the different job types mentioned above can also be called Employees (in the general sense). They share certain fields and operations. Therefore, a Manager "is a(n)" Employee. An Editor has an "is a(n)" Employee.
   - **Consider an interim superclass –** If you find that certain employee types share common fields and/or operations that are *not* shared by other employee types (for instance a list of skills), yet they are all "Employees", consider creating a common superclass for these employees – inherited from the top level superclass: Employee.
   - **Displaying information –** Remember that many of the fields that would be displayed are shared in common by all these employees (for instance: name, job title, employeeID). You might be able to display this common information from the top level superclass. In the subclass, simply "add to" what was displayed by the superclass, showing the fields that are unique to this particular employee type.

  - **Note:** This is done by overriding the method from the superclass and calling it from within the subclass method of the same name which then adds more code to display additional fields.
- **Encapsulation –** Demonstrate encapsulation for each of the classes in your design by including `get` and `set` methods for each private field, according to the type of access required.
- **Modeling –** Model the class hierarchy using class diagrams similar to those you saw in this lesson.

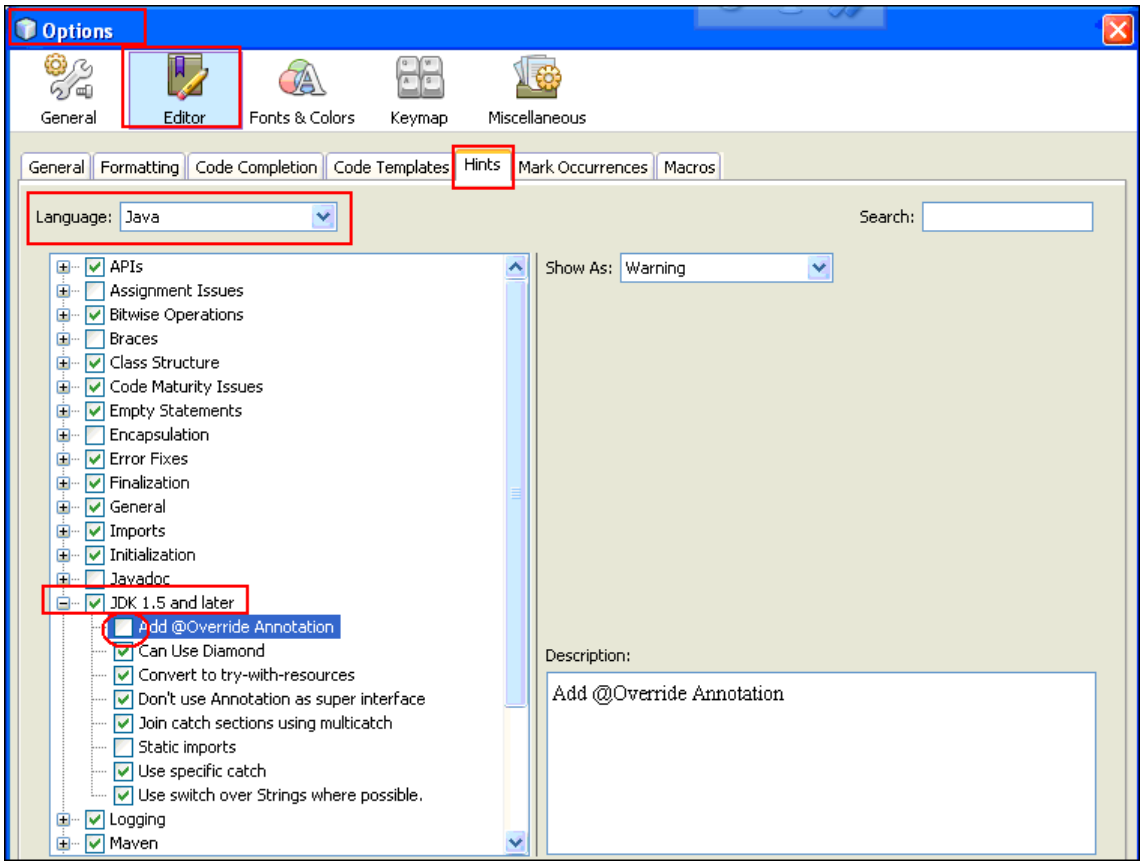
Solution:



Create the Classes

2. In NetBeans, create a new project from existing sources called Practice12. Set the Source Package Folder to point to `D:\labs\les12`. Remember to set the Binary Source Format property of the project. If you need further details, refer to Practice 2-2, Steps 3 and 4.
3. Before you begin creating the classes, change a property of the NetBeans IDE. The **Add @Override Annotation** property of the editor is useful when you are creating javadocs for your application. This property is applied when you override a method in the superclass. Since we are not creating javadocs in this course, you will turn off this property as it is merely distracting for our purposes. Follow the steps below to make this change:
   a. Select Tools > Options from the main menu.

b. In the Options window, click the **Editor** toolbar button and then click the **Hints** tab.
c. Change the **Language** to **Java**. The hints in the left column will change accordingly.
d. Expand the **JDK 1.5 and later** node. Beneath this node, deselect **Add @Override Annotation**.
e. Click **OK** to save the change and close the Options window.



4. Create the `Employee` class shown in the diagram above. The following steps will provide more details.
   a. All of the fields shown in the diagram should be private. Be sure to follow the same naming pattern that you have been using (camelCase).
   b. Use the Refactor feature of NetBeans to encapsulate these fields (create `get` methods for each field and `set` methods for each field). Change the access modifier for the `setEmployeeID` method to `private`.
      – **Note:** Employee IDs will be *calculated* to ensure uniqueness and you must restrict public *write* access to this field so that the IDs will always be unique. ID values are only set by the `calculateEmployeeID` method.
   c. Add another field, not shown in the diagram, called `employeeIDCounter`. Make it a `protected static int` field and initialize it to 0 (zero).

   Note

   A `static` field is a "class" field. There is only one value for this field that is shared by all instances of this class. The `static` field will be used here to store an integer value that is incremented from within the `calculateEmployeeID` method to generate the next ID value. The `employeeIDCounter` is accessed and incremented by all instances of the

`Employee` and its subclasses, thus ensuring that no duplicate employee IDs are generated.

In a real business application, this technique would not be robust enough to guarantee unique IDs. Instead, a database would probably generate the IDs. However, this technique suffices for our simple application.

d. Create the `calculateEmployeeID` method. It takes no arguments and does not return a value. In the body of this method, increment the `employeeIDCounter` and then set the new value in the `employeeID` field (use the `set` method of the field).

e. Create the `displayInformation` method. It takes no arguments and does not return a value. In this method, print out the value of each field of the class with a suitable label.

Solution:

```
public class Employee {
    protected static int employeeIDCounter = 0;
    private int employeeID;
    private String name;
    private String jobTitle;
    private int level;

    public void calculateEmployeeID() {
          employeeIDCounter++; // inc so employeeID's unique
          setEmployeeID(employeeIDCounter);
    }

    public void displayInformation() {
          System.out.println("Name: " + getName());
          System.out.println("Job Title:" + getJobTitle());
          System.out.println("Employee ID: " +
             getEmployeeID());
          System.out.println("Level: " + getLevel());
    }

    // The set and get methods are not shown here
}
```

f. Click Save to compile the class.

5. Create the Manager class from the diagram. The steps below provide more details.
   a. After creating the new Java Class file, add the following bolded phrase to the class declaration to indicate that it is a subclass of Employee:
      ```
      public class Manager extends Employee {
      ```
   b. Declare and instantiate the employeeList field as a private ArrayList (instead of the array of type Employee that is indicated in the diagram). This will be simpler to work with than an array.
      ```
      private ArrayList employeeList = new ArrayList();
      ```
   c. Add the necessary `import` statement to import the `java.util.ArrayList` class.

   Hint: Click on the error icon in the left margin and let NetBeans add the import statement for you.

   d. Add a public `setEmployee` method to add a single employee to the `employeeList`. The method takes an argument of type `Employee`. Use the `add` method of the `ArrayList` to add the `Employee` object to the `employeeList` object.
      ```
      public void setEmployee(Employee emp){
          employeeList.add(emp);
      }
      ```

   Question: What validation might you need to do in this method in a real-world application?

   e. Add a public `getEmployees` method that simply returns the `employeeList`.
      ```
      public ArrayList getEmployees(){
          return employeeList;
      }
      ```
   f. Add a `displayInformation` method to override the method in the `Employee` class. In this method, you will invoke the `displayInformation` method in the superclass and then display additional information specific to the `Manager` class.
      - Declare the method with the exact same signature as in the superclass method (returning void and accepting no arguments). NetBeans will display a green circle icon in the margin as you have finished typing the method declaration. This indicates that this method overrides the superclass method. Clicking the green circle opens the `Employee` class in the editor to show you the ancestor method. This can be helpful sometimes.
      - In the method block, invoke the superclass method using the `super` keyword as a reference to the `Employee` class.
      - Display the following message: "Manager has the following employees: "
      - Now iterate through the `employeeList` using an enhanced `for` loop. Remember that the `employeeList` is an `ArrayList` that holds `Objects`. The compiler does not know that these `Objects` happen to be `Employee` objects. Therefore, in order to get the `name` field from each object to display it, you have to cast the `Object` to an `Employee` (an `Employee` "is a(n)" `Object`). Declare a local variable at the top of this method of type `Employee`. This will hold the cast value. The code for this method is provided for you here:

```
public void displayInformation(){
     Employee emp;
     // Invoke the ancestor method
     super.displayInformation();
     System.out.println
       ("The manager has the following employees: ");

     for(Object obj : employeeList){
          // Cast the object as an Employee
          emp = (Employee)obj;
          // print the name, indented by a tab
          System.out.println("\t" + emp.getName());
     }
}
```

g. Save and compile your program.

6. Create the `SkilledEmployee` class from the diagram. This class should also extend `Employee`.
   a. Use an `ArrayList` instead of a `String` array when you declare the `skillList` field. Instantiate the field to an empty `ArrayList`.
      `private ArrayList skillList = new ArrayList();`
   b. Add the necessary `import` statement to import for the `ArrayList` class.
   c. Add a public `setSkill` method to add a single skill to the `skillList`. The method takes an argument of type `String`. Use the `add` method of the `ArrayList` to add the `String` to the `skillList` object.
   d. Add a public `getSkills` method that returns the `skillList`.
   e. Override the `displayInformation` method as you did in the `Manager` class. After invoking the superclass method, display the following message: "Employee has the following skills: ". Iterate through the `skillList` using an enhanced `for` loop, displaying each skill, indented by a tab as you did in the `Manager` class.
      – **Note:** The `skillList` object contains `String` objects. In this case, you can directly print the `Object` reference from the `ArrayList` without casting it to a `String`. The reason for this is that every `Object` has a `toString` method and the `println` method will invoke this for you, resulting in the display of the `String` value (i.e. the skill).

f. Click Save to compile the program.
   - **Note:** Consult the solution file for the `SkilledEmployee` class if you need help.

7. Create the `Editor` class as a subclass of `SkilledEmployee`.
   a. Declare the `prefersPaperEditing` field as a `private boolean`. (It will be initialized to a default value of false.)
   b. Add a `setPrefersPaperEditing` method that takes a `boolean` argument and returns `void`. Assign the argument to the private field.
   c. Add a `getEditingPreference` method that returns a `String` value. Use an `if/else` construct to check the value of `prefersPaperEditing` and set the return value to either "Paper" or "Electronic".
   d. Override the `displayInformation` method as you did in the `Manager` class, invoking the superclass method first and then displaying the return value of `this.getEditingPreference()` with a suitable label.
   e. Click Save to compile the program.
      - **Note:** Consult the solution file for the `Editor` class if you need help.
8. Create the remaining two classes from the diagram: `GraphicIllustrator` and `TechnicalWriter`. Both of these classes extend the `SkilledEmployee` class. It is not necessary to add any additional fields or methods, nor is it necessary to override the `displayInformation` method.
9. Save and compile the program.
10. Open the EmployeeTest class in the editor and examine the code.

    Note

    If there are any error indicators, check to make sure that you have spelled all of your method names the same way they are spelled in this class. If there are still error indicators after making any changes, try clicking the Save button again and/or try just clicking on a line in EmployeeTest that indicates an error. This will remind the syntax checker in NetBeans to try resolving the references once more.
11. Run the EmployeeTest class to test your program. You should see an output similar to the following screenshot:

```
Name: Fred Hanson
Job Title:Editor
Employee ID: 1
Level: 1
Employee has the following skills:
        technical editing
        typing
Editing preference: Paper
**** *****
Name: Frank Moses
Job Title:Graphic Illustrator
Employee ID: 2
Level: 3
Employee has the following skills:
        technical illustration
        video production
        media authoring
**** *****
Name: James Ralph
Job Title:Technical Writer
Employee ID: 3
Level: 1
Employee has the following skills:
        technical writing
**** *****
Name: Susan Smith
Job Title:Manager
Employee ID: 4
Level: 2
Manager has the following employees:
        Fred Hanson
        Frank Moses
        James Ralph
BUILD SUCCESSFUL (total time: 0 seconds)
```

Practice 12-2: Using a Java Interface

Overview

In this practice you create an interface called Printable and implement it within the class hierarchy that you built in Practice 12-1. You also examine and run another small application that uses the same Printable interface in order to better understand the benefits of using interfaces.

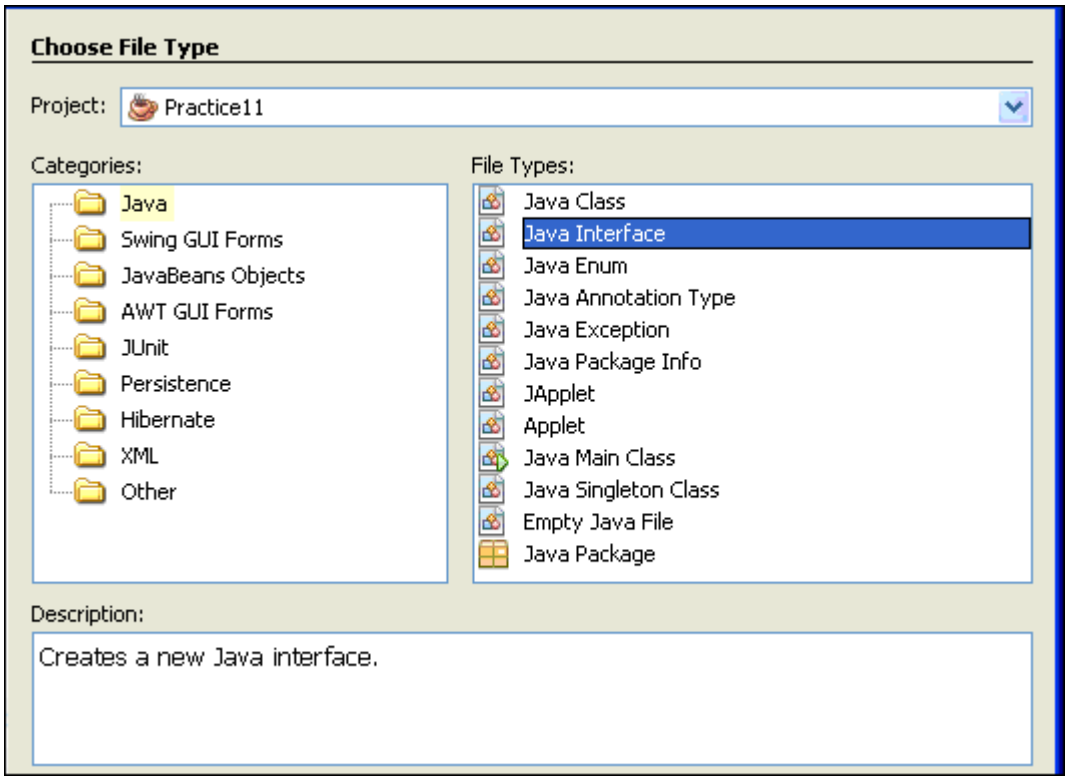
Assumptions

This practice assumes that the following files appear in the lab folder for this lesson, `D:\labs\les12:`

- Printer.java
- Country.java
- Region.java
- Company.java
- CompanyTest.java

Tasks

1. Create a new Java Interface using the NetBeans **New File** wizard.
   a. Click File > New.
   b. On the Choose File Type page, select **Java Interface** in the File Types column.



   c. Click **Next**. Name the interface “Printable”.
   d. Click **Finish**.

2. In the `Printable` interface, declare a `public abstract` method called `print`. It should return `void` and accept zero arguments.

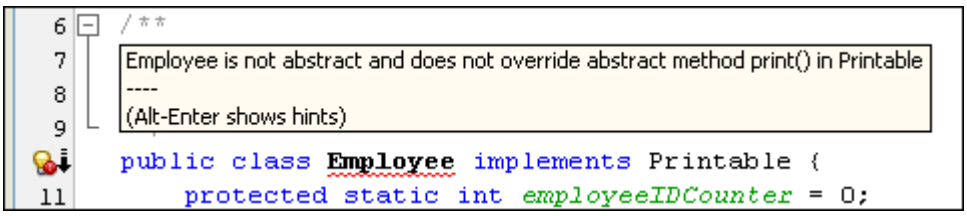
```
public abstract void print();
```

3. Click Save.
4. Implement the `Printable` interface in the `Employee` class.

   Note: Remember that all of the other classes in this hierarchy are subclasses of `Employee`, therefore, they also now implement `Printable` through inheritance.

```
public class Employee implements Printable {
```

5. The syntax checker now shows an error icon in the margin of this line. Move your cursor over the error icon to see the potential compilation error that it recognizes.



   Explanation: Any non-abstract classes that implement an interface must also implement all of the abstract methods of the interface. In this case, the only abstract method in `Printable` is `print`.

6. Change name of the `displayInformation` method to `print`.
7. Make this same change (`displayInformation` to `print`) in each of the following classes to ensure that they also implement the `print` method. You also need to change the name of the superclass method called in the first line of the new `print` method. (It is no longer called `displayInformation`.)
   - Manager
   - SkilledEmployee
   - Editor
8. Open the `Printer` class in the editor and examine its only method: `printToScreen`. Notice that this method takes an argument of type `Printable`. Any class that implements `Printable` would be accepted as an argument. This method invokes the `print` method of the `Printable` object.

```
public void printToScreen(Printable p){
    p.print();
}
```

9. In the `EmployeeTest` class, make the following changes:
   - Declare and create an instance of the `Printer` class.
   - For *every* invocation of the `displayInformation` method, comment out the line and instead, invoke the `printToScreen` method of the `Printer` object. Pass in a reference to the `Printable` object as shown below:

```
//myManager.displayInformation();
myPrinter.printToScreen(myManager);
```

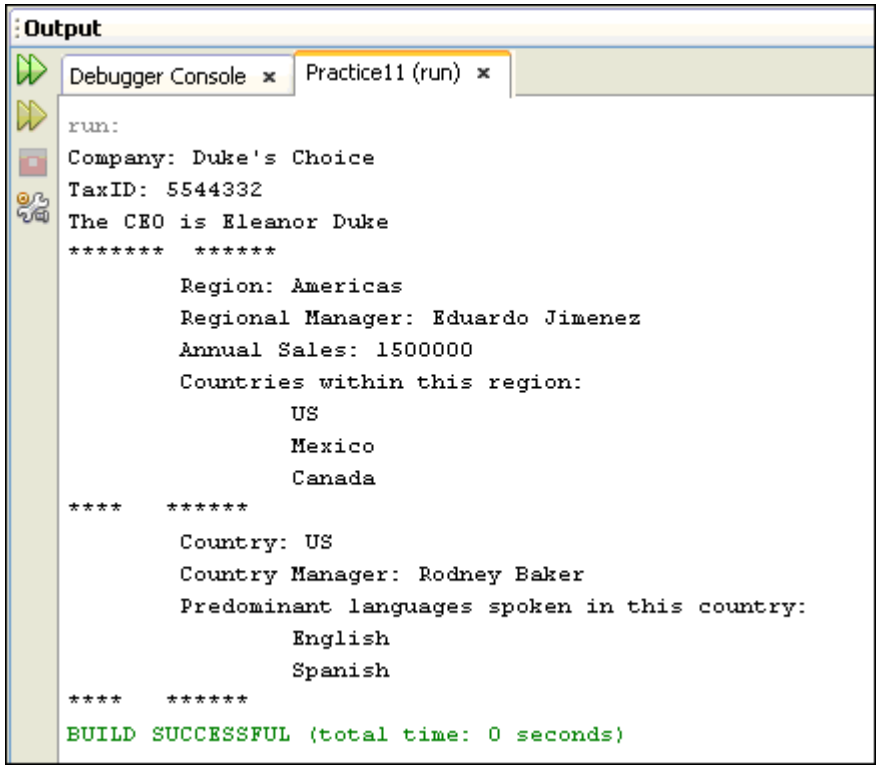
10. Save and compile your program.
11. Run the `EmployeeTest` class and examine the output. It should be identical to the output you saw before implementing the interface.

Discussion:

One of the benefits of using interfaces is that you can abstract functionality that is used in different applications and different class hierarchies. This functionality is moved into the interface and can then be used anywhere that the functionality is required. For example, in this practice, the ability to display class fields with labels and formatting has been moved into the `Printable` interface.

Now you test the cross-application benefit by running a different application that also implements `Printable`. The `Company` class hierarchy displays information about Duke's Choice top level management, as well as that of its regional and divisional management. The code is very similar to what you saw in the `Employee` hierarchy.

12. Close all of the classes you have been working on and open the following classes in the editor:
    - Company
    - Region
    - Country
    - CompanyTest
13. Examine the `Company` class first. This is the superclass of `Region` and `Country`. Notice that it implements the same `Printable` interface that you used in the `Employee` hierarchy.
14. Examine the `Region`, `Country` and `CompanyTest` classes as well.
15. Run the `CompanyTest` class to view the output of this application.



16. Close the Practice12 project in NetBeans.

You have now had an introductory exposure to Java Interfaces, one of the most valuable tools of the Java language. This topic is covered in much more detail in the Java SE7 Programming class.

Practices for Lesson 13: Handling Errors

Chapter 13

Practices for Lesson 13

Practices Overview

In these practices, you will experiment with handling checked exceptions. In the first practice, you handle an exception thrown by one of the Java foundation classes. In the second practice, you catch and throw a custom exception class. Solutions for these practices can be found in `D:\labs\soln\les13`.

Practice 13-1: Using a `try/catch` Block to Handle an Exception

Overview

In this practice you use the Java API documentation to examine the SimpleDateFormat class and find the exception thrown by its `parse` method. Then you create a class that calls the `parse` method, using a `try/catch` block to handle the exception.

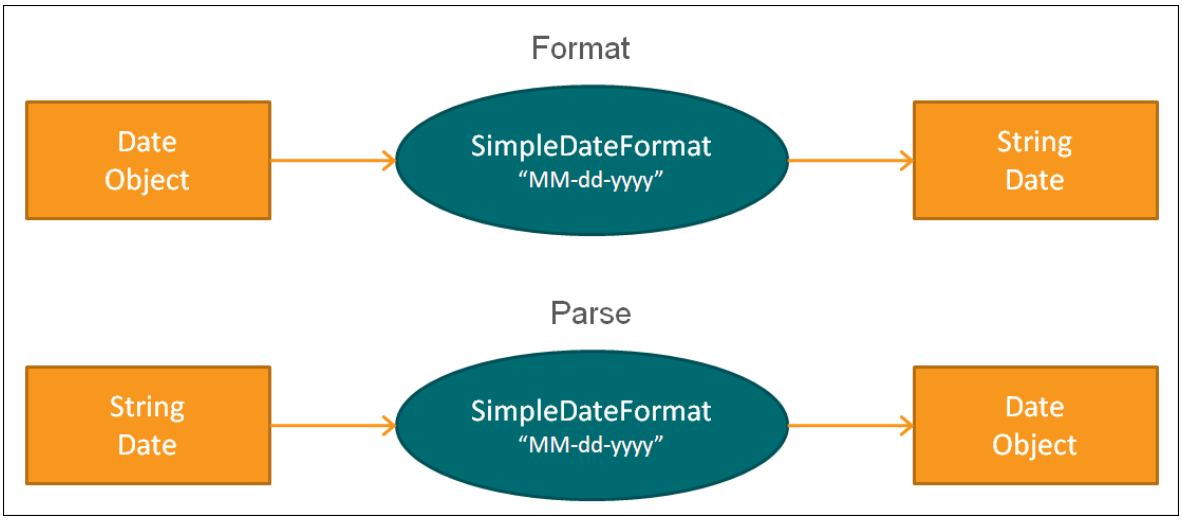
Assumptions

This practice assumes that the following files appear in the lab folder for this lesson, `D:\labs\les13:`

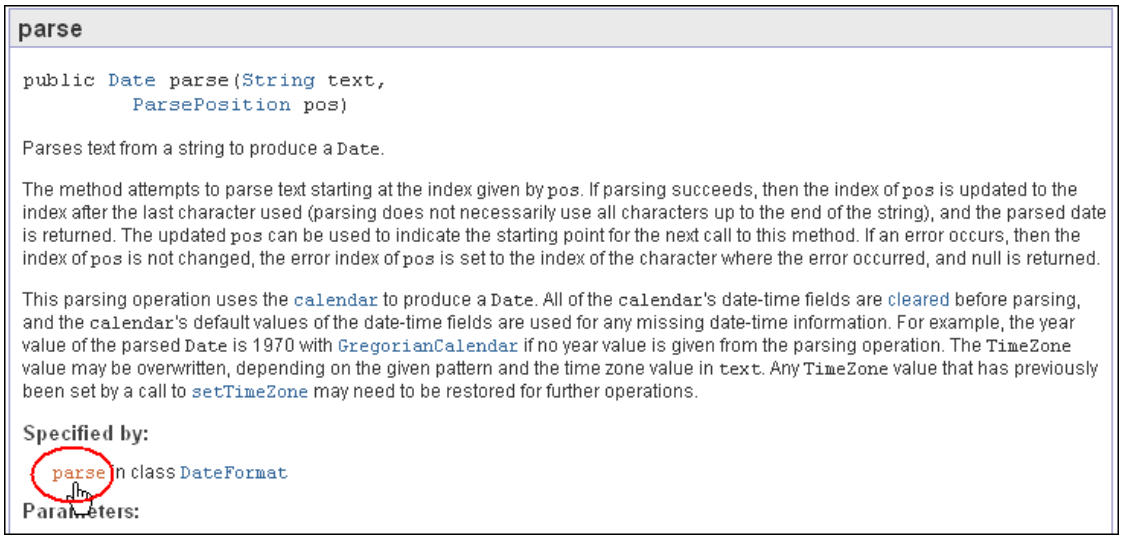
- DateTest.java

Tasks

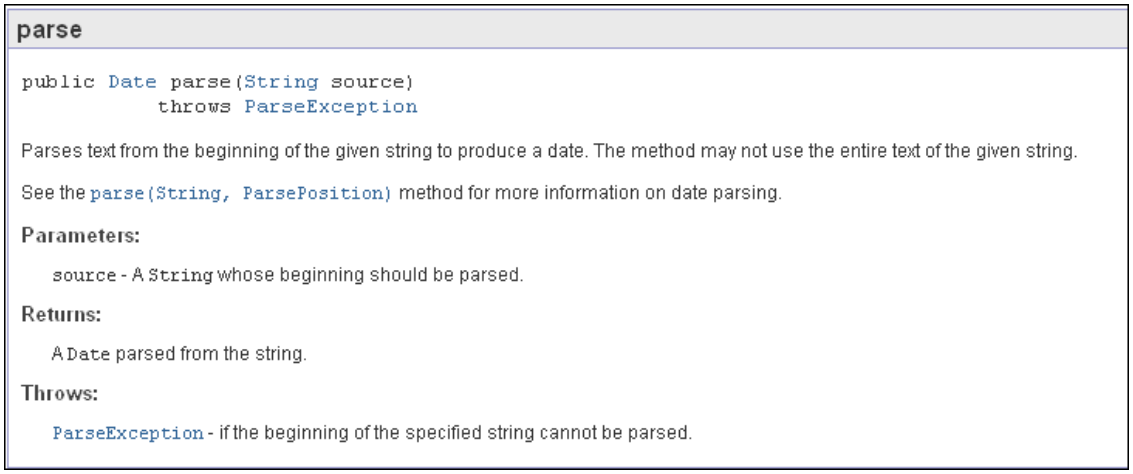
1. In NetBeans, create a new project from existing sources called Practice13. Set the Source Package Folder to point to `D:\labs\les13`. Remember to set the Binary Source Format property of the project. If you need further details, refer to Practice 2-2, Steps 3 and 4.
   There are many files in this project. Only the DateTest class is relevant for this practice.
2. Open the Java API Specification documentation by using the shortcut on the desktop.
3. Find the SimpleDateFormat class in the `java.text` package. This class allows you to pick a standard date format that will then be applied during both formatting and parsing. For instance, you *format* the `String` output of a `Date` object, and you *parse* (or create) a `Date` object based on a formatted `String` representation of the date.



4. The steps below will guide your examination of the SimpleDateFormat documentation.
   a. Find and click the `parse` method. As you can see, this method has two arguments. In this practice, you will invoke a simpler `parse` method that belongs to the superclass, `DateFormat` instead of this `parse` method you see here. The superclass method is not `private` and is therefore, available to a SimpleDateFormat object.
   b. In the **Specified by** section, click the `parse` link as shown below to go to the DateFormat documentation for this method.

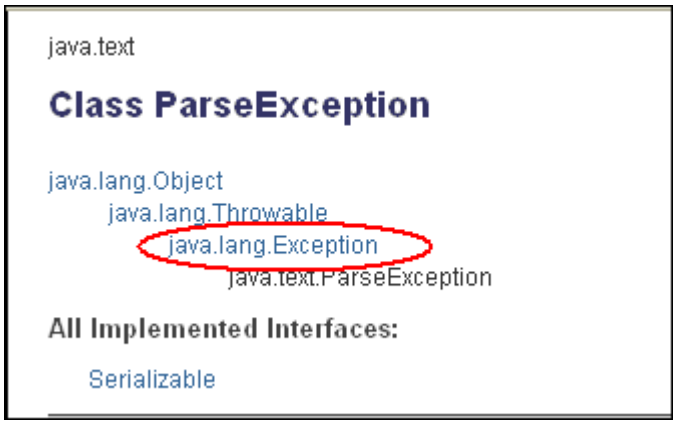


c. The javadocs now display a similar two-argument `parse` method in the DateFormat class. Scroll up to the one-argument `parse` method directly above this one.



d. Notice that this `parse` method accepts a single `String` argument and returns a `Date` object. What, if any, exceptions does it throw?

e. Is the ParseException a checked exception (one that must be caught or thrown)? Click the ParseException link to see its class hierarchy. Is it a subclass of Exception? If so, it is a checked exception and must be handled in the program.



f. Click the browser's Back button twice to return to the SimpleDateFormat documentation.

g. Find the `format` method. This `format` method accepts three arguments. Once again, there is a simpler version of this method defined in the DateFormat class that you will use in this practice. Click the `format` link under the **Specified by** heading to view the DateFormat documentation for this method.

h. Scroll down to find the one-argument `format` method. Notice that it accepts a single `Date` object argument and returns a `String` value. Does this `format` method throw any exceptions? (*Answer:* It does not.)

5. In NetBeans, create a new Java Class called "DateManipulator". Provide field declarations as indicated in the table below. More detailed instructions follow the table.

Step	Code Description	Choices or Values
a.	Declare a field of type `Date`	Name: `myDate`
b.	Declare and instantiate a field of type `SimpleDateFormat` specifying its default format in the constructor.	Name: `simpleDF` Default format: `"MM/dd/yyyy"`
c.	Add the necessary import statements	`java.util.Date` `java.text.SimpleDateFormat`

a. Declare a field of type `Date`, using the variable name `myDate`.

b. Declare a field of type `SimpleDateFormat` called `simpleDF`. Use the `new` operator to instantiate (create an object) the `SimpleDateFormat` field in the same line as the declaration. Specify the default format for this object by passing the format String to the object constructor as shown below.

```
SimpleDateFormat simpleDF =new SimpleDateFormat("MM/dd/yyyy");
```

c. Click the error icons that appear next to each of these lines of code. Select the option to add the required `import` statements. There are two possible `Date` objects that can be imported. Choose the `java.util.Date`.

6. Add a public method called `parseDate` that accepts a `String` argument called `dateString` and returns `void`.

Note: This method creates a `Date` object instance by invoking the `parse` method. It formats the `Date` object according to the default format of the `SimpleDateFormat` object and displays the resulting string. It also displays the native date format of the `Date` object for comparison. In addition to this, the method handles the ParseException using a `try / catch` block.

7. Follow the high level steps in the table below to code the `parseDate` method. More detailed steps are provided following the table.

Step	Code Description	Choices or Values
a.	Declare a local `String` variable.	Name: `formatDate`
b.	Invoke the `parse` method of the `SimpleDateFormat` object. Ignore the error sign for now.	Pass the `dateString` as the `String` argument. Assign the return value to the `myDate` field.
c.	Display a message indicating that the `parse` method was successful	

Step	Code Description	Choices or Values
d.	Display the natively formatted `date` object	Print the `Date` object, itself, with a suitable label.
e.	Invoke the `format` method of the `SimpleDateFormat` object, passing `myDate` as the method argument.	Assign the return value to `formatDate`
f.	Display the formatted date String with a suitable label	

g.	Enclose all of the above code in a `try` block	`try{`
		`  // lines of code here`
		`}`

h.	Catch the `ParseException` and display the exception object	`catch (ParseException ex) {`
		`  // display the ex object here`
		`}`

i.	Add the missing import statement	`java.text.ParseException`

a. Declare a local `String` variable called `formatDate`. This will be used to hold the String representation of the formatted `Date` object.

b. Invoke the `parse` method of the `simpleDF` object, passing the method's `dateString` argument to the `parse` method. This method returns a `Date` object so assign the return value to `myDate`. You will, no doubt, notice an error icon in the left margin for this line of code. Put your cursor over it to see the warning message. You will add a try/catch block later to this fix this.

c. Use `System.out.println` to print the message "Parse successful".

d. Again use `System.out.println` to print `myDate` along with the message "Date with native format: ". (Hint: the Java Virtual Machine will invoke the `toString` method of the `Date` object.)

e. Invoke the `format` method of the `simpleDF` object. Pass `myDate` as the argument to the method. Assign the return value to `formatDate`.

```
formatDate = simpleDF.format(myDate);
```

f. Display `formatDate` with a suitable label. Suggestion "Formatted date: " + `formatDate`

g. Now you fix the error regarding the missing `try/catch` block. Surround all of the above lines of code in a `try` block.

   – Hint: Right click anywhere in the editor and select **Format** to correct the indentation of your code.

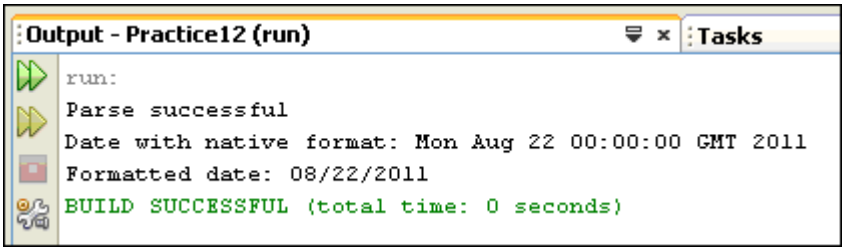
h. On the next line after the closing brace of the `try` block, add a `catch` block that catches the ParseException and displays the exception object to the screen.

i. Right click the error icon in the left margin and allow NetBeans to provide the missing import statement (java.text.ParseException).

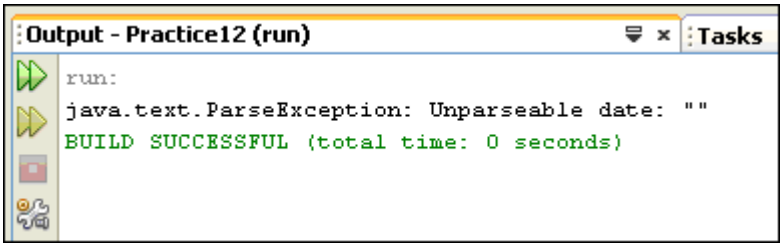
Solution:

```
public void parseDate(String myDate){
     try{
        String formatDate;
        myDate = simpleDF.parse(dateString);
        System.out.println("Parse successful");
        System.out.println("Date with native format: "
                + myDate);
        formatDate = simpleDF.format(myDate);
        System.out.println("Formatted date: "
              + formatDate);
     }catch (ParseException ex) {
         System.out.println(ex);
     }
}
```

8. Save and compile your program.
9. Open the DateTest class and examine it. Substitute your own date between the quotation marks in the `parseDate` method call. Use the format, "MM/dd/yyyy", as specified in the comment.
10. Click Save to compile.
11. Run the DateTest and check the output. If your date was formatted correctly, the ParseException will not appear in the output. You should, however, see the difference between the native Date formatting and the effect of the SimpleDateFormat class on the formatting of that same date.



12. Now change the argument value of the `parseDate` method in DateTest so it reverts to being an empty string (""). Save and compile the program.
13. Run the DateTest class again. The ParseException will be thrown this time and you should see the message from the exception object in the output.



Note: You will notice that the "Parse successful" message does not appear. That particular display occurred in the line immediately following the `parse` method call. When the `parse` method threw the exception, the program went immediately to the `catch` block and the remaining lines of code in the `try` block were not executed.

Practice 13-2: Catching and Throwing a Custom Exception

Overview

In this practice you use a custom exception called "InvalidSkillException". You use this with the Employee Tracking application that you designed and built in Practices 12-1 and 12-2. You throw the InvalidSkillException in one method and catch it in the calling method.

A new set of Java source files for the Employee hierarchy are provided for your use in this practice.

Assumptions

This practice assumes that the following files appear in the lab folder for this lesson and consequently, in the Practice13 project:

- Editor.java
- Employee.java
- EmployeeTest.java
- GraphicIllustrator.java
- InvalidSkillException.java
- Manager.java
- Printable.java
- Printer.java
- SkilledEmployee.java
- TechnicalWriter.java

Tasks

Assume that there is a list of valid skill types that can be associated with a particular job role in the Employee Tracking system. In the `setSkill` method (belonging to the SkilledEmployee class), some validation is necessary in order to determine whether the skill argument passed into the method is valid for the employee's job title. If the skill is not valid, the method will throw an InvalidSkillException. The calling method in the EmployeeTest class must then catch this exception.

Note: In our simple example, the validation in the `setSkill` method will be greatly simplified and does not represent the robust, thorough type of validation that would occur in a "real world" application. Our purpose here is to focus on catching and throwing exceptions.

1. Open the InvalidSkillException class and examine the code. It is very simple. The only thing that makes this function as an exception is that it extends `Exception`. You see a public no-args constructor and also a public constructor that accepts a String argument. That argument is the message that will be displayed when this Exception object is printed.
2. Open the SkilledEmployee class and modify the `setSkill` method as described in the steps below. The solution for the `setSkill` method is shown following the steps if you need further assistance.
   a. Add a `throws` clause to the method declaration so that it `throws` an `InvalidSkillException`.
   b. As the first line of code in the method, declare a `boolean` variable called "valid" and initialize it to `true`

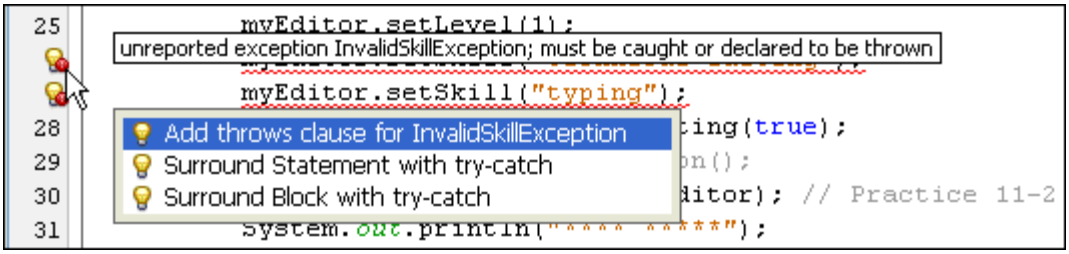
c. Use an `if/else` construct to set the value of the `valid` variable to `false` if the `skill` argument is `null` or has a length of less than 5. Otherwise, set `valid = true`.

d. Use another `if/else` construct to test the value of the `valid` variable.
   - If it is `true`, add the skill to the `skillList`.
   - If it is `false`, throw a `new InvalidSkillException`, using the constructor that takes a `String` argument for the exception message.
   - The message should show the `skill` argument that caused the exception and concatenate that to a string literal that indicates that this is an invalid value for an employee with this particular job. Also display the employee's job title, using `this.getJobTitle()`.

Solution:

```
public void setSkill(String skill) throws
     InvalidSkillException {
     Boolean valid = true;
     if(skill == null | skill.length() < 5){
        valid = false;
     }
     else {
        valid = true;
     }
     if (!valid) {
        throw new InvalidSkillException(skill +
          " is not valid for the " +
          this.getJobTitle() + " job.");
     }
     else {
        skillList.add(skill);
     }
}
```

3. Save and compile your program
4. Open the EmployeeTest class. You should now see red error icons in the left margin for every line of code that calls the `setSkill` method. Click one of the error icons to read the error description and see the options it offers to help you solve the problem.



When you compiled the SkilledEmployee class, you made NetBeans aware of the fact that the `setSkill` method throws an InvalidSkillException. The compiler is checking this (remember, this is a "checked exception") and expects you to either catch it or re-throw it

whenever you invoke `setSkill`. None of the suggested options will work well in this particular situation, so you will add the `try/catch` block yourself.

5. Surround each *set* of `setSkill` method invocations with a `try/catch` block. In the `catch` block, display the exception object. You will have to add `try/catch` blocks for `myEditor`, `myGI`, and `myTW`.

   Example: The two invocations for the `myEditor` object can all go within a single block.

```
try{
     myEditor.setSkill("typing");
     myEditor.setSkill("technical editing");
}
catch(InvalidSkillException ex){
     System.out.println(ex);
}
```

6. Change the String argument in one of the `setSkill` calls to a shortened String (less than 5 characters) so that it will be deemed invalid and the exception will be thrown.
7. Save and compile your program. Run the EmployeeTest class and examine the output.

```
**** *****
InvalidSkillException: tw is not a valid skill for the Technical Writer job.
Name: James Ralph
Job Title:Technical Writer
Employee ID: 3
Level: 1
**** *****
Name: Susan Smith
Job Title:Manager
Employee ID: 4
Level: 2
Manager has the following employees:
        Fred Hanson
        Frank Moses
        James Ralph
BUILD SUCCESSFUL (total time: 0 seconds)
```

Note: The catching of an InvalidSkillException did not prevent the remainder of the method from executing. It only prevented the saving of an invalid skill for this employee. Handling checked exceptions in this way also offers an opportunity to write the error information to a log file or to prompt the user to enter the skill again (assuming a different user interface than we are using here).