Practices and Solutions

Table of Contents

Practices for Lesson 1

Practice 1*: Introducing the Java and Oracle Platforms*

There is no practice for this lesson.

Practice 2*: Basic Java Syntax and Coding Conventions*

Goal

The goal of this practice is to create, examine, and understand Java code. You start by editing and running a very simple Java application. In addition, in this practice you become acquainted with the Order Entry application, which you will use throughout this course. You will use a UML model of the Order Entry application as a guide to creating additional class files for it, and you will run some simple Java applications, fixing any errors that occur.

The practices in this and the next two lessons are written to help you understand the syntax and structure of Java. Their sole purpose is to instruct rather than to reflect any set of best practices for application development. The goals of the practices from Lesson 5 to the end of the course are different. Starting in Lesson 5, you use JDeveloper to build an application by using techniques you would use during real-world development. The practices continue to support the technical material presented in the lesson while incorporating some best practices that you use while developing a Java application.

Your Assignment

In this practice you edit and run a very simple Java application. You then start to get familiar with the Order Entry application.

Editing and Running a Simple Java Application

Note: If you close a DOS window or change the location of the `.class` files, you must set the `CLASSPATH` variable *again*.

1. Open a DOS window, navigate to the **`C:\labs\\temp`** directory (or the location specified by your instructor), and create a file called **`HelloWorld.java`** using **Notepad** with the following commands:

```
cd \labs\\temp

notepad HelloWorld.java
```

2. In **Notepad**, enter the following code, placing your name in the comments (after the double slashes). Also, ensure that the case of the code text after the comments is preserved (remember that Java is case-sensitive):

```
// File:   HelloWorld.java
// Author: <Enter Your Name>
   public class HelloWorld {
     public static void main(String[] args)
     {
```

Practice 2: Basic Java Syntax and Coding Conventions (continued)

```
            System.out.println("Hello World!");
        }
    }
```

3. Save the file to the `C:\labs\\temp` directory by using the File > Save menu option, but keep Notepad running in case of compilation errors that require you to edit the source to make corrections.
4. Compile the `HelloWorld.java` file (file name capitalization is important).
   a. In the DOS window, ensure that the current directory is **`C:\labs\\temp`** (or the directory specified by your instructor) and that the **PATH** system variable references **`JDeveloper\jdk\bin`**.
   b. Check that the Java source file is saved to disk.
   (**Hint:** Enter the command **`dir Hello*`**.)
   c. Compile the file using the command **`javac HelloWorld.java`**.
   d. Name the file that is created if you successfully compiled the code.
   (**Hint:** Enter the command **`dir Hello*`**.)
5. Run the `HelloWorld` application (Again, remember that capitalization is important.), and examine the results.
   a. Run the file using the command `java HelloWorld`.
   b. What is displayed in the DOS window?
6. Modify the `CLASSPATH` session variable to use the directory where the **`.class`** file is stored. In the DOS window, use the **`set CLASSPATH=C:\labs\\temp`** command to set the variable. The variable will be set for the duration of the DOS session. If you open another DOS window, you must set the `CLASSPATH` variable again.
7. Run the `HelloWorld` application again.
   a. Use the command `java HelloWorld`.
   b. What is displayed in the DOS window?
8. Close Notepad but do *not* exit the DOS window because you continue to work with this environment in the following practice exercises.

Creating Order Entry Class Files (Examining the Customer Class)

The practices throughout this course are based on the Order Entry application. Turn to the end of Lesson 2 in the student guide to see the UML model of the classes in the Order Entry application.

In this practice you examine some of the class files used in the application.

Practice 2: Basic Java Syntax and Coding Conventions (continued)

1. Copy the **`Customer.java`** file from the `c:\labs\` directory to your **`C:\labs\\OrderEntry\src\oe`** directory.
2. In the **DOS window**, change your current working directory to: **`C:\labs\\OrderEntry\src\oe`**.
3. Using Notepad, review the `Customer` class and provide answers to the following:
   a. Name all of the instance variables in `Customer`.
   b. How many instance methods are there in `Customer`?
   c. What is the return type of the method that gets the customer's name?
   d. What is the access modifier for the class?
4. Close the file and at the DOS prompt, compile the `Customer.java` file using the following command as a guide:

   `javac -d C:\labs\\OrderEntry\classes Customer.java`

   Where is the compiled `.class` file created?

   (**Hint:** Enter **`cd ..\..\classes\oe`**, and then type **`dir`**.)

Incorporating `Order.java` into your Application Files

Add the `Order.java` file to your application structure, review the code, and compile it.

1. In Notepad, open the `\labs\\Order.java` file and save it to the directory for your OE package source code (`C:\labs\\OrderEntry\src\oe` or the directory specified by your instructor). The attributes are different from those in the UML model. The customer and item information are incorporated later.
2. Notice that two additional attributes (getters and setters) have been added:
   - shipmode (String) is used to calculate shipping costs.
   - status (String) is used to determine the order's place in the order fulfillment process.
3. Ensure that you are in the `C:\labs\\OrderEntry\src\oe` directory. Use the following command to compile the Order.java file, which places the .class file in the directory with the compiled version of the `Customer` class:

   `javac -d C:\labs\\OrderEntry\classes Order.java`

Creating and Compiling the Application Class with a `main()` Method

1. Create a file called `OrderEntry.java` containing the main method as follows. Place the source file in the source directory that contains the `java` files (`C:\labs\\OrderEntry\src\oe`). This file is a skeleton that is used for launching the course application. Use the following code to create the file:

Practice 2: Basic Java Syntax and Coding Conventions (continued)

```
package oe;
public class OrderEntry {
   public static void main(String[] args)
   {
   System.out.println("Order Entry
Application");
   }
}
```

2. Save and compile **`OrderEntry.java`** with the following command line:

```
javac -d C:\labs\\OrderEntry\classes
OrderEntry.java
```

3. Run the `OrderEntry` application.

   a. Open a DOS window and use the `cd` command to change the directory to `C:\labs\\OrderEntry\classes`.

   b. Run the file using the command **`java oe.OrderEntry`**.

      Note: To ensure that the correct version of code is run, irrespective of the working directory, include classpath information in the run command as follows:

```
java -classpath C:\labs\\OrderEntry\classes
     oe.OrderEntry
```

Practice 3*: Exploring Primitive Data Types and Operators*

Goal

The goal of this practice is to declare and initialize variables and use them with operators to calculate new values. You also categorize the primitive data types and use them in code.

Note: If you have successfully completed the previous practice, you should continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les03` directory and continue with this practice.

Remember that if you close a DOS window or change the location of the `.class` files, you must set the `CLASSPATH` variable again.

Your Assignment

Add some code to the simple `main()` method in the `OrderEntry` class created in the last practice: declare variables to hold the costs of some rental items, and after displaying the contents of these variables, perform various tests and calculations on them and display the results.

Note: Ensure that the `CLASSPATH` variable points to the location of your `.class` files (`C:\labs\\OrderEntry\classes` or the location specified by your instructor).

Modifying the OrderEntry Class and Adding Some Calculations

1. Declare and initialize two variables in the `main()` method to hold the cost of two rental items. The values of the two items are 2.95 and 3.50. Name the items anything you like, but do not use single-character variable names; instead, use longer meaningful names such as `item1` and `item2`. Also, think about your choice of variable type.

   Note: Recompile the class after each step, fix any compiler errors that may arise, and run the class to view any output.

   a. Use four different statements: two to declare your variables and two more to initialize them, as follows:
   ```
   double item1;
   double item2;
   item1 = 2.95;
   item2 = 3.50;
   ```

   b. However you can also combine the declaration and initialization of both variables into a single statement:
   ```
   float item1 = 2.95, item2 = 3.50;
   ```

Practice 3: Exploring Primitive Data Types and Operators (continued)

2. Use `System.out.println()` to display the contents of your variables. After recompiling the class, run the class and see what is displayed. Then, modify the code to display more meaningful messages.

   a. To simply display the contents of the variables:
   ```
   System.out.println(item1);
   System.out.println(item2);
   ```

   b. To display more useful information:
   Hint: Use the **+** operator.
   ```
   System.out.println("Item costs " + item1);
   System.out.println("Item costs " + item2);
   ```

3. Now that you have the costs for the items, calculate the total charge for the rental. Declare and initialize a variable to hold the number of days and to track the line numbers. This variable holds the number of days for which the customer rents the items, and initializes the value to `2` for two days. Display the total in a meaningful way such as `Total cost: 6.982125`.

   a. Create a variable to hold the item total:
   ```
   double itemTotal;
   ```

   b. Declare and initialize variables to hold the number of days (initialized to 2) and to keep track of line numbers:
   ```
   int line = 1, numOfDays = 2;
   ```

   c. Calculate the total charge for the rental:
   ```
   itemTotal = ((item1 * numOfDays) + (item2 *
   numOfDays));
   System.out.println("Total cost: " + itemTotal);
   ```

4. Display the item total in such a way that the customer can see how it has been calculated. To do so, display the item total as the item cost multiplied by the number of rental days:
   ```
   System.out.println(
      "Item " + line++ +" is " + item1 + " * " +
   numOfDays + " days = " +(item1 * numOfDays));

   System.out.println(
     "Item " + line++ +" is " + item2 + " *   " + numOfDays
   + " days = " +(item2 * numOfDays));
   ```

5. Compile and run the **`OrderEntry`** class. Ensure that the `.class` file has been placed in the correct directory (**`C:\labs\\\OrderEntry\classes\oe`**).

Practice 4*: Controlling Program Flow*

Goal

The goal of this practice is to make use of flow-control constructs that provide methods to determine the number of days in a month and to handle leap years.

Note: If you have successfully completed the previous practice, continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les04` directory and continue with this practice.

Remember that if you close a DOS window or change the location of the `.class` files, you *must* set the `CLASSPATH` variable again.

Your Assignment

Create a program that calculates the return date of a rental item based on the day it was rented and on the total number of days before it must be returned. You must determine how many days are in the month and whether the year is a leap year.

Modifying the `OrderEntry` Class to Calculate Dates

1. Modify the number of days in a month. Use a `switch` statement to set an integer value to the number of days in the month that you specify. For now, add all of the code in the `main()` method of the `OrderEntry.java` application.

   a. Declare three integers to hold the day, month, and year. Initialize these variables with a date of your choice.
   ```
   int day = 25, mth = 5, yr = 2000;
   ```

   b. Add a simple statement to display the date. Select a format that you prefer, such as day/month/year or month/day/year.
   ```
   System.out.println(day + "/" + mth + "/" + yr);
   ```

   c. Declare a variable to hold the number of days in the current month. Then, using a `switch` statement, determine the value to store in this variable. Use **`daysInMonth`** as the name of the variable.

Note: The hardest part of this exercise is remembering how many days there really are in each month. Here is a reminder if you need it: There are 30 days in September, April, June, and November. All other months have 31 days, except for February, which has 28 days (ignore leap years for now).

```
int daysInMonth;
switch (mth) {
  case 4:
  case 6:
  case 9:
  case 11: daysInMonth = 30;
           break;
  case 2:  daysInMonth = 28;
           break;
  default: daysInMonth = 31;
           break;

    }
```

d. Add a simple statement to display the number of days for the current month.

```
System.out.println(daysInMonth + " days in
month");
```

2. Ensure that your `CLASSPATH` is set correctly (`C:\labs\\OrderEntry\classes`), and then compile and test the program. Experiment with different values for the month. What happens if you initialize the month with an invalid value, such as 13?

   For January 27, 2000, the output should look something like the following:

```
27/1/2000
31 days in the month
```

3. Use a `for` loop to display dates.

   a. Using a **`for`** loop, extend your program so that it prints out all of the dates between your specified day/month/year and the end of the month. Here is an example:

      If your day variable is 27, your month variable is 1 (January), and your year variable is 2000, then your program must display all of the dates between January 27 and January 31 (inclusive) as follows:

```
27/1/2000
28/1/2000
29/1/2000
30/1/2000
31/1/2000
```

Hint: You must use the result of the `switch` statement in step 1 to determine the last day in the month.

```
System.out.println("Printing all days to end of
month using for loop...");
for (int temp1 = day;
         temp1 <= daysInMonth;
         temp1++) {

     System.out.println(temp1 + "/" + mth + "/" +
yr);

}
```

b. Compile and test your program, making sure that it works with a variety of dates.

c. Modify your program so that it outputs a maximum of 10 dates. For example, if your day/month/year variables are 19/1/2000, the output must now be as follows:

```
19/1/2000
20/1/2000
21/1/2000
22/1/2000
23/1/2000
24/1/2000
25/1/2000
26/1/2000
27/1/2000
28/1/2000
```

Ensure that your program works for dates near the end of the month, such as 30/1/2000. In this situation, it must output only the following:

```
30/1/2000
31/1/2000
```

To do this, use the following code:

```
// Print maximum of 10 dates, using a for
loop

System.out.println("Printing maximum of 10
days using for loop...");
  for (int temp3 = day, iter = 0;
     temp3 <= daysInMonth && iter < 10;
     temp3++, iter++) {

     System.out.println(temp3 + "/" + mth +
"/" + yr);
}
```

d. Compile your program. Then, test it with a variety of dates to ensure that it still works.

4. Determine whether the `year` you specify is a leap year. Use the `boolean` operators `&&` and `||`.

   a. Build a `boolean` statement that tests `year` to see whether it is a leap year. A year is a leap year if it is divisible by 4, *and* if it is either not divisible by 100 *or* it is divisible by 400.

```
boolean isLeapYear = (yr % 4 == 0) &&
     // year divisible by 4?  AND
          (  (yr % 100 != 0) ||
// year not divisible by 100 OR
               (yr % 400 == 0)  );
// year divisible by 400
```

   b. Modify your `switch` statement from step 1 to apply to leap years. Remember that February has 29 days in leap years and 28 days in nonleap years.

```
switch (mth) {
 case 4:
 case 6:
 case 9:
 case 11: daysInMonth = 30;
          break;
 case 2:  daysInMonth = ((isLeapYear) ? 29 : 28);
          break;
 default: daysInMonth = 31;
          break;
}
```

   c. Build and test your program with a variety of dates. The following table includes some sample leap years and nonleap years that you may want to use as test data:

Leap years	Nonleap years
1996	1997
1984	2001
2000	1900
1964	1967

5. Calculate the date on which each rental is due. The due date is the current date plus three days. For this test, you use a number of different dates for the current date, not just today's date.

   a. Declare three variables to hold the due date.
   ```
   int dueDay, dueMth, dueYr;
   ```

   b. Add a variable to hold the rental period of three days.
   ```
   int rentDays = 3;
   ```

Practice 4: Controlling Program Flow (continued)

c. Add to your program the due date calculation that adds the rental period to the date you used in step 1. Display your original date and the due date in a meaningful way. The output should look something like:

```
Rental Date:              27/2/2001
Number of rental days:    3
Date Due:                 2/3/2001
```

```
dueDay = rentDays + day;
System.out.println("Rental Date:   " + day + "/"
                   + mth + "/" + yr);
System.out.println("Number of rental days: " +
rentDays);
System.out.println("Date Due back:       " +
dueDay + "/" + dueMth + "/" + dueYr);
```

d. Test your routine with several dates. For example, try February 29, 2001.

e. What are the problems you must address?

f. Modify your program to catch input dates with invalid months (not 1–12).

```
// Determine invalid months
if ((mth > 0 )& (mth <13))
  System.out.println (mth + " is a valid month");
else
  System.out.println (mth + " is not a valid month");
```

6. When building a software solution to a problem, you must determine the size and scope of the problem and address all of the pertinent issues. One of the issues is what to do if the rental period extends beyond the current month. For example, if the rental date is August 29 and the rental is for three days, the return date must be September 1, but with the current solution, it is August 32, which is an obvious error. Acme Video store rents items only for 10 or fewer days.
To handle such issues, follow these steps:

a. Add code to test whether the calculation results in a valid day.

```
// is dueDay valid for the current month?
if (dueDay <= daysInMonth)
     System.out.println(dueDay + "/" + dueMth + "/" +
     dueYr);
```

b. If the rental period crosses into a new month, be sure to increment the month.

```
else {
// set dueDay to a day in the next month
dueDay = (dueDay - daysInMonth);
// increment the month
dueMth = (dueMth + 1);
```

c. If the rental period crosses into a new year, be sure to increment the year.

```
// is the new month in a new year
if (dueMth > 12) {
dueMth = 1;
dueYr += 1;
}
```

d. Test your routine with various dates.

Optional (Do if you have time.)

1. Replace the `for` loop that prints all days to the end of the month with a `while` loop.

   Print all days to the end of the month using a `while` loop:

```
// initialize temp2 to day of the month
int temp2 = day;
System.out.println("Printing all days to end of month
using while loop...");
while (temp2 <= daysInMonth) {
   System.out.println(temp2 + "/" + mth + "/" +
yr);
   temp2++;
}
```

2. Replace the `for` loop that prints a maximum of 10 days using a `while` loop.

   Print a maximum of ten days using a `while` loop:

```
System.out.println("Printing maximum of 10 days using
while loop...");
// initialize temp4 to day of the month
int temp4 = day;
int numSoFar = 0;
while (temp4 <= daysInMonth) {
   System.out.println(temp4 + "/" + mth + "/" +  yr);
   temp4++;
   if (++numSoFar == 10)
     break;
}
```

Practices for Lesson 5

Practice 5*: Building Java with Oracle JDeveloper 11g*

Starting in Practice 5, you use JDeveloper to build an application using techniques you would use during real-world development. The practice supports the technical material presented in the lesson and incorporates best practices to use while developing a Java application.

Goal

In this practice, you explore using the Oracle JDeveloper IDE to create an application and a project so that you can manage your Java files more easily during the development process. You learn how to create one or more Java applications and classes using the rapid code-generation features.

More importantly, you now start using JDeveloper for most of the remaining lab work for this course (occasionally returning to the command line for various tasks). By the end of the course, you will have built and deployed the course GUI application while continuing to develop your Java and JDeveloper skills.

In this practice, you use the files found in the **`C:\labs\\les05`** directory (or the location specified by your instructor). They are similar to the ones you created in earlier practices (with subtle differences).

Your Assignment

- In the first section, you explore JDeveloper's rapid code-generation features by using the default JDeveloper paths to create a new default application. You then create a project from existing code in the `C:\labs\\les05` directory. You also view a UML diagram that displays the classes that have been created up to this point in the course.
- In the optional section, you run and test the application with the debugger.

Creating an Application and Project

Launch Oracle JDeveloper 11*g* from the desktop icon provided, or ask your instructor how to start JDeveloper. (In this practice, you must use the `C:\labs\\les05` directory.)

1. Create a new application.
   a. In the Applications Navigator, on the left hand side of the screen, click **New Application**.
   b. In the Create Application dialog, enter the following application name: **`OrderEntryApplication`**.
   c. Change the Directory Name field to **`C:\labs\\les05`** (or the directory specified by your instructor). You can use the Browse button to locate the directory.

Practice 5: Building Java with Oracle JDeveloper 11g (continued)

d. In the Application Template field, ensure that the default value **Generic Application** is selected. Click **OK** to create your application definition.

e. Click **Finish**. You create a project explicitly in the next step.

2. Create a new project in the new application, and populate the project with files from the `C:\labs\\les05\src\oe` directory.

   a. Notice that the new `OrderEntryApplication` application has been created and appears in the Navigator. Right-click `OrderEntryApplication` and select the **New Project** menu item. The New Gallery dialog displays. Select **Project from Existing Source** in the Items section of the New Gallery window and click **OK** to invoke the Create Project from Existing Source wizard.

   b. Click the **Next** button on the Welcome screen. In the Location page of the wizard, change the name of the project to **`OrderEntryProject`** and select the **`C:\labs\\les05OrderEntryProject`** directory (or the directory you have been using). Click the **Next** button.

   c. On the Specify Source page of the wizard, click the **Add** button next to Java Source Paths. Navigate to the subdirectory containing the Java source files (which are in the `src\oe` subdirectory of the `C:\labs\\les05` directory tree). Click **Select**.

   d. On the Included tab, ensure that the **Include Content from Subfolders** check box is checked. Confirm that the output directory is **`C:\labs\\les05\classes`**, and then click **Add**. Check that all the `.java` files in the `C:\labs\\les05\src\oe` directory are to be included, and click the **Finish** button.

The new project is displayed in the Navigator. Double-click the name to invoke the Project Properties dialog. In the **Project Source Paths** page, set the Default Package field to **`oe`** and click **OK**.

   e. Note that the `OrderEntryApplication` and `OrderEntryProject` names are in italics in the Navigator. This is because the application is not yet saved. Select it and then select **File > Save All**.
   The font reverts to normal after the application is saved. Expand the application and project nodes to examine their contents.

   f. Compile the files in the project. Right-click **`OrderEntryProject`** and select the **Rebuild** option.
   Observe the compilation progress in the Log window.

   g. Right-click the project again and select **Run** from the context menu. In the Choose Default Run Target dialog box, browse to the **`oe`** package and select **`OrderEntry.java`**. Click **OK**.
   View the output results of your application in the Log window.

Practice 5: Building Java with Oracle JDeveloper 11g (continued)

Examining a UML Diagram

View a UML diagram showing the classes that were created in the lessons up to this point in the course.

1. In the Applications Navigator, click **Open Application**.
2. Browse to locate **`C:\labs\\les05`**, select **`OrderEntryWorkspaceLes05.jws`** and click Open. If you get a message asking if you want to migrate application files, click **Yes**.
3. In the Navigator select the **`OrderEntryProjectLes05`** project, and then select **File > Open.**
4. In the **Open** dialog box, double-click **model**, and then double-click **`oe`**.
5. Select **`UML Class Diagram1.java_diagram`** and click **Open**. The diagram displays the classes created up to this point in the course.

Optional: Debugging the Course Application

Run the `OrderEntryApplication` application in debug mode and examine how the debugger works.

1. Expand the Application Sources and oe nodes in the Navigator, and then open the **`Order.java`** file in the Code Editor by double-clicking the file name.
2. Scroll down to lines 67 and 68. Remove the comment marks from the `System.out.println`, and then set breakpoints on the following two statements:

   ```
   item1Total = item1.getItemTotal();
   System.out.println("Item 2 Total: " +
   item2Total);
   ```

   Note: To set a breakpoint on a line, click the left margin next to the line.
3. In the Navigator, select the **`OrderEntry.java`** file, right-click, and then select **Debug** from the context menu.
   JDeveloper creates a new debugger tab that opens at the lower-right portion of the JDeveloper window. The execution of the code stops at your first breakpoint, as indicated by a red arrow. The red arrow indicates the next line that is about to be executed when you resume debugging.
   The Log/Debug window is modified to contain two tabs—a Log tab and a Breakpoints tab—in which you can view all of the breakpoints that you have set. The Log tab must display the output results generated by the application. Resize the windows if required.
4. Visually select the **Smart Data** tab in the lower-right window, which is called the Debug window.
   Note: If the Debug window is not visible, display it by selecting the View > Debugger > Smart Data menu item. The check box next to the Data item must be selected to make it visible. Otherwise, the tab is removed from the Debug window.

Practice 5: Building Java with Oracle JDeveloper 11g (continued)

5. Locate the **`item1`** variable in the Smart Data tab and expand it. Using the values of **`quantity`** and **`unitPrice`**, calculate the **`item1Total`** of the order. What is the present value of **`item1Total`**?
   (**Hint:** The value for `quantity` is displayed as 2 and the value `for unitPrice` is displayed as 2.95.) However the value for `item1Total` displays as "null" in the Smart Data window.
6. Select **Debug > Step Over** (alternatively, press F8 or click the appropriate toolbar icon) to calculate `item1Total`. Note the changes to the `item1Total` instance variable in the Smart Data tab of the Debug window. Was your calculation in the previous step correct?
7. In the toolbar at the top of the screen, click **Resume** (F9 or select Debug > Resume). The red arrow in the Code Editor advances and highlights the line with the next breakpoint detected in the code-execution sequence.
8. Continue by selecting the **Debug|Resume** menu (press the F9 key, or click the toolbar button) until the program is completed. You need to select it only once.
9. Remove the breakpoints from the **`Order.java`** source file by clicking each breakpoint entry (red dot) in the margin for each line with a breakpoint.

Practices for Lesson 6

Practice 6*: Creating Classes and Objects*

Goal

The goal of this practice is to complete the basic functionality for existing method bodies of the Customer class. You then create customer objects and manipulate them by using their public instance methods. You display the Customer information back to the JDeveloper message window.

Note: For this practice you need to change to the `les06` directory, and load the `OrderEntryApplicationLes06` application. (This workspace file contains the solution to the previous practice, Practice 5.)

Your Assignment

In this practice, you begin refining the application for the Order Processing business area. These classes continue to form the basis for the application that you are building for the remainder of the course. After creating one or more Customer objects, you associate a customer with an order.

Refining the Customer Class

1. In the `OrderEntryProjectLes06` in the Application Navigator, make the following changes to the Customer class:
   a. Make all instance variables private. To do this, double-click the **`Customer.java`** file to open it in the Source Editor. Make your changes directly in the code.
   b. Assign each of the **`setXXX()`** methods to its appropriate field.
   c. The `get()` methods must be assigned. Confirm whether the **`getXXX()`** methods return their appropriate field values.

      Note: The naming convention—such as `setId(), setName(),` and so on—for these methods makes the classes more intuitive and easier to use.
2. At the moment, there is no way to display most or all details for a Customer object by calling one method. You need to address this deficiency.
   a. Add a new **`toString()`** public method to the class, without arguments, and return a String containing the customer's ID, name, address, and phone number. The resultant string should be a concatenation of the attributes that you want to display, as in the following example:

```
public String toString() {
return property1 + " " + property2;
     }
```

      Note: The `toString()` method is a special method that is called whenever a String representation of an object is needed. The `toString()` method is very

useful to add to any class, and thus it is added to almost all of the classes that you create. When adding the `toString` method, a dialog box appears with the message, "OK to override method." Click **Yes**.

b. Save the Customer class and compile it to remove any syntax errors. Compile by right-clicking the **`Customer.java`** file and selecting the **Make** option.

Creating Customer Objects (`OrderEntry` Class)

3. Modify the `main()` method in the `OrderEntry` class to create two customer objects.

   a. In the `main()` method of `OrderEntry.java` create two customer objects using the **`new`** operator, assigning each one to a different object reference (use `customer1` and `customer2`).

   ```
   Customer customer1 = new Customer();
   Customer customer2 = new Customer();
   ```

   b. At the end of the `main()` method, initialize the state of each customer object by calling its public **`setXXX()`** methods to set the ID, name, address, and phone. Use the data in the following table:

Id	Name	Address	Phone
1	Gary Williams	Houston,TX	713. 555. 8765
2	Lynn Munsinger	Orlando, FL	407.695.2210

```
customer1.setId(1);
customer1.setName("Gary Williams");
customer1.setAddress("Houston, TX");
customer1.setPhone("713.555.8765");

customer2.setId(2);
customer2.setName("Lynn Munsinger");
customer2.setAddress("Orlando, FL");
customer2.setPhone("407.695.2210");
```

   c. Print the two customer objects created, under a printed heading of "**Customers:**" by calling the **`toString()`** method inside the argument of the `System.out.println(…)` method. For example:

   ```
   System.out.println("\nCustomers:");
   System.out.println(customer1.toString());…
   ```

   Note: Alternatively, you can just print the customer object reference variable to achieve the same result, as in the following example:

   ```
   System.out.println(customer1);
   ```

   The latter technique is a feature of Java that is discussed in a subsequent lesson.

   d. Save the `OrderEntry` class and compile and run the class to view the results.

Practice 6: Creating Classes and Objects (continued)

Modifying `OrderEntry` to Associate a Customer to an Order

4. In the `main()` method of the `OrderEntry` class, associate one of the customer objects with the order object, and then display the order details.

   a. Call the **`setCustomer()`** method of the `order` object passing in the object reference of **`customer1`** (or **`customer2`**).
   `order.setCustomer(customer1);`

   b. After setting the customer, call the **`showOrder()`** method of the **`order`** object.
   `order.showOrder();`

   c. Save, compile and run the **`OrderEntry`** class.

Practice 7: *Object Life Cycle Classes*

Goal

The goal of this practice is to provide experience with creating and using constructors, class-wide methods, and attributes. You also use an existing `DataMan` class to provide a data-access layer for finding customers and products in the OrderEntry application. Part of the practice is designed to help you understand method overloading by creating more than one constructor and/or method with the same name in the same class.

Note: If you have successfully completed the previous practice, continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les07` directory, load the `OrderEntryApplicationLes07` application, and continue with this practice. (This application contains the solution to the previous practice, Practice 6).

Viewing the model: To view the course application model up to this practice, expand `OrderEntryApplicationLes07 application – OrderEntryProjectLes07 – Application Sources – oe`, and double-click the UML Class Diagram1 entry. This diagram displays all of the classes created up to this point in the course.

Your Assignment

Create one or more suitable constructors to properly initialize the customer objects when instantiated. Examine the `Order` class and the new instantiations. Copy and examine the `DataMan` class to provide class-wide (static) attributes of customer objects to be used by the OrderEntry application when it associates a customer object with an order.

Modifying Customer Information

1. Create two constructors for the `Customer` class. Create a `no-arg` constructor to provide default initialization, and another constructor to set the actual name, address, and phone properties. The `no-arg` constructor is invoked by the second constructor.
   a. Add a `no-arg` constructor to the `Customer` class; the constructor is used to generate the next unique ID for the customer object by first declaring a class variable called **`nextCustomerId`** as a private static integer initialized to zero.
      `private static int nextCustomerId = 0;`
   b. In the `OrderEntry` class, comment out the **`customer.setId`**, **`customer.setName`**, **`customer.setAddress`** and **`customer.setPhone`** statements for both `customer1` and `customer2`.

c. In the `Customer` class, create a `no-arg` constructor, increment the **`nextCustomerId`**, and use the **`setId()`** method with **`nextCustomerId`** to set the ID of the customer.

```
public                                          Customer()
{
    nextCustomerId++;
    setId(nextCustomerId);
}
```

d. Add a second constructor that accepts a **name, address**, and **phone** as `String` arguments. This constructor must set the corresponding properties to these values.

```
public     Customer(String     theName,     String
theAddress, String thePhone)
```

e. In the first line of the second constructor, chain it to the first constructor by invoking the `no-arg` constructor by using the **`this()`** keyword. This is done to ensure that the ID of a customer is always set regardless of the constructor used.

```
{
  this();
  name                  =                  theName;
  address               =               theAddress;
  phone                 =                 thePhone;
}
```

f. Save, compile, and run the OrderEntry class to check the results. Including the order and item details that are displayed as output, you should see "**`Customer: 1 null null null`**".

Replacing and Examining the `Order.java` File

1. In Windows Explorer, copy the `Order.java` class from the `C:\labs\\Les07Adds` directory into your current working `...\src\oe` directory. For example, if you are working in `les07` directory, copy the files under `C:\Labs\\les07\src\oe`.

   a. In the Application Navigator, select your application (**OrderEntryApplication**) and select the **File > Open** menu option. Navigate to your current `...\src\oe` directory and select the **`Order.java`** file. Click the **Open** button. The file is now included in the list of files. If needed, select **View > Refresh** to see the new file in the navigator.

2. The new version of the Order class also has one constructor. Examine the way in which the order date information is managed.

   a. Note that the **`OrderDate`** variable that was commented out is now a `private` variable.

b. After the package statement at the top of the class, notice the import statements (before the class declaration):

```
import java.util.Date;
import java.util.Calendar;
```

c. Note that the `orderDate` type is `Date` instead of `String`, and that the three `integer` variables (`day, month`, and `year`) have been removed.

3. Examine the methods that depend on the three integer date variables to use `orderDate`.

a. The return type and value of the **`getOrderDate()`** method have been replaced as follows:

```
public Date getOrderDate()
{
return orderDate;
}
```

Also included is an overloaded void `setOrderdate()` method that accepts a `Date` as its argument and sets the `orderDate` variable.

b. The **`getShipDate()`** method had used the `Calendar` class to calculate the ship date. The body of `getShipDate()` has been replaced with the following:

```
int daysToShip = Util.getDaysToShip(region);
Calendar c = Calendar.getInstance();
c.setTime(orderDate);
c.add(Calendar.DAY_OF_MONTH, daysToShip);
return c.getTime().toString();
```

c. The **`setOrderDate()`** method body is coded to set the `orderDate` by using the `Calendar` class methods, using the three input arguments. The following date initialization code has been deleted:

```
day = 0;
month = 0;
year = 0;
```

d. Note that the **`setOrderDate(int,int,int)`** method has been modified. The following three bold lines of code:

```
if ((m > 0 && m <= 12) && (y > 0  )) {
 day = d;
 month = m;
 year = y;
}
```

have been replaced with these three lines of code:

```
Calendar c = Calendar.getInstance();
c.set(y, m - 1, d);
orderDate = c.getTime();
```

4. A `no-arg` constructor has been created to initialize the order number, date, and total. Note the following:

- A new class variable, **`nextOrderId`** has been declared and initialized to **100**.
- In the `no-arg` constructor, the **ID** of the order is set to the value in **`nextOrderId`**, and then the **`nextOrderId`** value is incremented by **1**. The **`orderTotal`** value is set to **0**, and the **`orderDate`** value is set as follows:
  `orderDate  = new Date;`

Loading the `Dataman.java` Class File into JDeveloper

The `DataMan` class is used to create the data that is used to test the application. The file creates the customer objects and later is used to access a database for information. This class is really a convenience class that simplifies your application testing. However, after this class is completed, it can be changed to retrieve data from a database without impacting your application.

1. In Windows Explorer, copy the **`DataMan.java`** class from the `C:\labs\` directory into your current working …`\src\oe` directory.
2. Select your application and select the **File > Open** menu option. Navigate to your current **`...\src\oe`** directory and select the **`DataMan.java`** file. Click **OK**. The file is now included in the list of classes. If needed, select **View > Refresh** to see the new file in the navigator.
3. Save and compile the **`DataMan`** class.
   Note: You can compile DataMan.java by right-clicking the file and selecting the Make menu option.
4. Save, compile, and run the **`OrderEntry`** class to verify that the code still works. You can compile `OrderEntry.java` by right-clicking the file and selecting the **Make** menu option.

Modifying `OrderEntry` to Use `DataMan`

Modify the `main()` method in `OrderEntry` to use customer objects from the `DataMan` class.

1. Use the class name **`DataMan.`** as the prefix to all customer reference variables customer1 and customer2. For example, change the code:
   `order.setCustomer(customer1);`
   to become:
   `order.setCustomer(DataMan.customer1);`

   Note: You are accessing a class variable via its class name—that is, there is no need to create a `DataMan` object. In addition, the customer variables in `DataMan` are visible to `OrderEntry` because they have default (package) access.
2. Save, compile and run the **`OrderEntry`** class to verify that the code still works. Replace **`customer1`** with **`customer3`** or **`customer4`** from `DataMan` to confirm that your code now uses the customer objects from `DataMan`.

Practices for Lesson 8

Practice 8: *Using Strings and the `StringBuffer`, Wrapper, and Text-Formatting Classes*

Goal

The goal of this practice is to modify the `Util` class to provide generic methods to support formatting the order details, such as presenting the total as a currency and controlling the date string format that is displayed. This should give you exposure to using some of the java text formatting classes.

In this practice, you use the `GregorianCalendar` class. This class enables you to obtain a date value for a specific point in time. You can specify a date and time and see the behavior of your class respond to that specific date and time. The class can then be based on the values you enter rather than on the system date and time.

Note: If you have successfully completed the previous practice, continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les08` directory, load the `OrderEntryApplicationLes08` application, and continue with this practice.

Viewing the model: To view the course application model up to this practice, load the OrderEntryApplicationLes08 application. In the Applications – Navigator node, expand OrderEntryApplicationLes08 – OrderEntryProjectLes08 - Application Sources – oe, and double-click the UML Class Diagram1 entry. This diagram displays all of the classes created up to this point in the course.

Your Assignment

Create a method called `toMoney()` to return a currency-formatted string for the order total. Also create a method called `toDateString()` that formats the date in a particular way. Then, modify the `Order` class to use these methods to alter the display of order details, such as the order date and total.

Adding Formatting Methods to the `Util` Class

1. Create a static method called `toMoney()` that accepts an amount as a double and returns a `String`.

   a. Open `Util.java` and add the following import statement to the class:
   ```
   import java.text.DecimalFormat;
   ```

   b. Add the following `toMoney()` method code to the class to format a double:
   ```
   public static String toMoney(double amount) {
   DecimalFormat df = new DecimalFormat("$##,###.00");
   return df.format(amount);
   }
   ```

   c. Save and compile the **`Util`** class.

Practice 8: Using Strings and the StringBuffer, Wrapper, and Text-Formatting Classes (continued)

2. Us the static `toDateString()` method to format a date.

   a. Add the following import statements to the **`Util`** class:
   ```
   import java.util.Date;
   import java.text.SimpleDateFormat;
   ```

   b. Use the following code for your method:
   ```
   public static String toDateString(Date d) {
   SimpleDateFormat df = new SimpleDateFormat("dd-  MMMM-
   yyyy");
   return df.format(d);
   }
   ```

   c. Save and compile the **`Util`** class.

3. In this step, you use the `GregorianCalendar` class. This class enables you to obtain a date value for a specific point in time. You can specify that date and time and then see the behavior of your class based on the values you enter (and not simply the system date and time).

   Create another static method called `getDate()` that accepts three integers representing the day, month, and year, and returns a `java.util.Date` object representing the specified date (for example, month = 1 represents January on input). Because many of the methods in the `Date` class that could have been used are deprecated, you use the `GregorianCalendar` class to assist with this task.

   a. Import the **`java.util.GregorianCalendar`** class.

   b. Use the following for the method:
   ```
   public static Date getDate(int day,int month,int year)
   {
        // Decrement month, Java interprets 0 as
        // January.
     GregorianCalendar gc =
         new GregorianCalendar(year, --month, day);
    return gc.getTime();
   }
   ```

   c. Save and compile the **`Util`** class.

Using the Util Formatting Method in the `Order` Class

In the `Order` class, modify the `toString()` method to use the `Util` class methods `toMoney()` and `toDateString()` altering the display format.

1. In the **`toString()`** method, replace the return value with the following text. When `shipMode` is not specified, you do not need to display the information for "`Shipped:` ".
   ```
   return "Order: " + id +
     " Date: " + Util.toDateString(orderDate) +
   ```

```
        " Shipped: " + shipMode +
        " (" + Util.toMoney(getOrderTotal()) + ")";
```

2. Save and compile the **Order** class, and then run **OrderEntry** to view the changes to the displayed order details.
3. Import the **java.text.MessageFormat** class in the Order class and use this class to format the **toString()** return value as follows:

```
import java.text.MessageFormat;
Object[] msgVals = {new Integer(id),
  Util.toDateString(orderDate), shipMode,
  Util.toMoney(getOrderTotal()) };
return MessageFormat.format(
                  "Order: {0} Date: {1} Shipped:
     {2} (Total: {3})",msgVals);
```

4. Save and compile the **Order** class, and then run the OrderEntry class to view the results of the displayed order. The change to the displayed total should appear.

Optional: Using Formatting in the OrderItem Class

In the OrderItem class, modify the toString() method to use the Util.toMoney() methods to alter the display format of the item total.

1. In the **toString**() method, replace the **return** statement with the following:

```
return lineNbr + " " + quantity + " " +
                 Util.toMoney(unitPrice);
```

2. Save and compile the **OrderItem** class, and then run the **OrderEntry** class to view the changes to the order item total.

Optional: Using Util.getDate() to Set the Order Date

1. In the OrderEntry class, alter the **second order object creation statement** to use the **Util.getDate()** method to provide the value for the first argument in the constructor. Select the previous day's date for the values of the day, month, and year arguments supplied to the Util.getDate() method.

   The call to the constructor should look like the following:

```
Order order2 = new Order(Util.getDate(7, 3,   2002),
"overnight");
```

2. Save, compile and run the **OrderEntry** class to confirm that the order date has been set correctly.

Practice 9: *Using Streams for I/O*

Goal

The goal of this practice is to use some of the byte- and character-based stream classes to read and write application data. You also use Object Serialization to save and restore objects.

Note: If you have successfully completed the previous practice, continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les09` directory, load the `OrderEntryApplicationLes09` application, and continue with this practice.

Viewing the model: To view the course application model up to this practice, load the `OrderEntryApplicationLes09` application. In the Applications – Navigator node, expand `OrderEntryApplicationLes09 – OrderEntryProjectLes09 – Application Sources – oe` and double-click the `UML Class Diagram1` entry. This diagram displays all of the classes created up to this point in the course.

Your Assignment

In this practice you use some of the stream classes to manipulate data. First you use the `PrintWriter` class to write a file containing customer information. Then, you use various classes – `FileInputStream, InputStreamReader` and `Scanner` - to read from this file and output the values. Finally, you use object serialization to save and restore customer and order information. Import the necessary I/O classes when prompted to do so by JDeveloper.

Using PrintWriter to Create a File Containing Customer Data

Create a file to contain the customer information that is hard-coded in the `DataMan` class.

1. At the end of the `OrderEntry` class `main` method, declare a `String` variable for the name of the file that will hold the customer information. Call the file **`Customers.txt`**.
   ```
   String fileName = "customers.txt";
   ```
2. Declare an instance of the **`PrintWriter`** class to write to the file.
   ```
   PrintWriter pw = new PrintWriter(fileName);
   ```
3. Write a record for each of the customers in the `DataMan` class, using the value returned by the **`toString()`** method.
   Note that you do not have to explicitly use the `toString()` method.
   ```
   pw.println(DataMan.customer1);
   pw.println(DataMan.customer2);
   pw.println(DataMan.customer3);
   pw.println(DataMan.customer4);
   ```

4. Add the following statement to the `main` method declaration. (Exception handling is discussed in Lesson 14).
   ```
   throws Exception
   ```
5. Close the instance of **`PrintWriter`**.

Using Different Classes to Read the `Customers.txt` File and Print the Values.

Notice the different syntax used in the following steps, and (in one case) the different output.

1. Use `FileInputStream` to read and output the contents of the Customers file. Remember that `FileInputStream` is a class that is used for byte-based streams.
   a. Declare an instance of **`FileInputStream`** to read the file that you created.
      ```
      FileInputStream fis = new   FileInputStream(fileName);
      ```
   b. Declare a **variable** of type `int` to hold the size of the buffer and set it to the value returned by the `available()` method.
      ```
      int fileSize = fis.available();
      ```
   c. Create a **`byte` array** of the size of the buffer to store the bytes read in from the file.
      ```
      byte[] bbuf = new byte[fileSize];
      ```
   d. Read the file in from the buffer.
   e. Close **`FileInputStream`**.
   f. Try to print the buffer as a `String`.
   g. Run **`OrderEntry`**.
      What does the output look like? The result is unrecognizable as customer information because output from `FileOutputStream` is not a Java character string.
   h. Replace the print buffer instruction with a loop to print out each byte from the array individually.
      ```
      for (int cx = 0; cx < fileSize; cx++) {
         System.out.print(bbuf[cx]);
       }
      ```
   i. Rerun **`OrderEntry`**.
      What does the output look like this time? The output is simply a list of the decimal values of the byte stream returned (the stored ASCII characters). In order to view the output as text, you would need to cast each byte to `char`.

2. Use `InputStreamReader` to read and output the values from the Customers file. Remember that `InputStreamReader` handles character-based data.

   a. Declare an instance of **`InputStreamReader`** that reads an input stream containing the **Customers** file.
   ```
   InputStreamReader isr = new InputStreamReader(new
   FileInputStream(fileName));
   ```

   b. Using the buffer size stored in 2b (above), create a character array to hold the file contents.
   ```
   char[] cbuf = new char[fileSize];
   ```

   c. Read the file into the buffer and print it as a single unit.
   ```
   isr.read(cbuf);
   System.out.println(cbuf);
   ```

   d. Close the instance of **`InputStreamReader`**.

   e. Run `OrderEntry`. You should now see customer information correctly displayed as earlier in the program.

3. Use `Scanner` to read and print the contents of the Customer file.

   a. Declare an instance of **`Scanner`** to read the file you created earlier.
   ```
   Scanner sc = new Scanner(new File(fileName));
   ```

   b. Define **a loop** to read in and print one line of the file at a time until the end of file is reached. (Use the `Scanner nextLine()` method).
   ```
   while (sc.hasNext()) {
     System.out.println(sc.nextLine());
    }
   ```

   c. Close your instance of **`Scanner`**.

   d. Run **`OrderEntry`**. You should see the same correct customer information output as before.

Using Serialization to Save and Restore Objects.

In this practice you use serialization to save and restore first a simple object, and then a more complex one containing nested objects. You then mark one of the nested objects as "transient" to specify that it should not be saved when the owning object is written to file. The files created by this approach are a permanent copy of the object data, which can be used elsewhere in this application, or in another one.
(**Hint:** Refer to the serialization example in the lesson if you need help.)

1. **Use serialization to save and restore a Customer object:** Save the `customer1` object to a stream and then restore and run it.

   a. Ensure that the `Order`, `OrderItem` and `Customer` classes can use object serialization, by specifying that they implement the **`Serializable`** interface.

b. In the **OrderEntry** class, declare a new **ObjectOutputStream** instance based on a new **FileOutputStream** instance, referencing the file you want to save the object to. (Call the file **customers.ser**).
```
ObjectOutputStream cs = new   ObjectOutputStream(new
FileOutputStream("customers.ser"));
```

c. Write the **DataMan customer1** object to the file.
```
cs.writeObject(DataMan.customer1);
//entire object is written
```

d. Close the **ObjectOutputStream** instance.

e. Create an **ObjectInputStream** instance based on a new **FileInputStream** instance, referencing the file you just created, to enable you to read the object back.
```
ObjectInputStream ois =
new ObjectInputStream(new
FileInputStream("customers.ser"));
```

f. Read the saved **Customer** object from **customers.ser** into a different `Customer` variable.
```
Customer restCust1 = (Customer)ois.readObject();
//entire object is read
```

g. Close the **ObjectInputStream** instance.

h. Print out the restored `Customer` object.

i. Run `OrderEntry`.

In the Log window, you should see the same information as displayed from the original customer1 earlier. This is a very simple object that does not contain any nested objects or object references within it.

2. **Use serialization to save and restore an Order object:** Save the `order2` object to a stream and then restore and run it. The `Order` class is more complex, and contains `OrderItem` and `Customer` classes nested within it, enabling you to see the power of object serialization.

   a. In the `OrderEntry` class, declare a new **ObjectOutputStream** instance as before, referencing the file you want to save the object to. (Call the file **orders.ser**.)

   b. Write the **order2** object to the file. This saves the complete `Order` class structure, known as a "graph."

   c. Close the **ObjectOutputStream** instance.

   d. Create an **ObjectInputStream** instance as before to enable you to read the object back from the **orders.ser** file.

   e. Create a new instance of `Order` called **restOrd2** to hold the restored order object, and read the saved **order2** object into it.

f. Close the **`ObjectInputStream`** instance.

g. Print out the **`restOrd2`** object.

h. Run `OrderEntry`. You should see the details for `restOrd2` in the Log window—the same information as displayed from the original `order2` earlier in the Log.

Using the "transient" Modifier to Prevent Fields being Saved and Restored

If a nested object's class is not marked as serializable, or you do not want it to be stored with the "owning" object, you mark it as "transient." This tells the JVM to ignore it when writing the object to an object stream.

It is essential that references to a transient object must include a test for a null value, in order to be safe when processing a restored copy of the owning object.

It has been decided that, in the `OrderEntry` application, customer information will no `longer` be stored in the order graph. To accomplish this, mark the customer variable as "transient."

1. In **`Order.java`**, add the modifier **`transient`** to the customer variable.
   `private transient Customer customer;`
2. Scroll down to the `showOrder()` method, and check that references to customer are conditional on it having a non-null value.
3. Run `OrderEntry`. In the Log window, you should see that the information displayed for `restOrd2` no longer contains the customer information.

Practice 10*: Inheritance and Polymorphism*

Goal

The goal of this practice is to understand how to create subclasses in Java and use polymorphism with inheritance through the `Company` and `Individual` subclasses of the `Customer` class. You refine the subclasses, override some methods, and add some new attributes using the Class Editor in JDeveloper.

Note: If you have successfully completed the previous practice, continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les10` directory, load the `OrderEntryApplicationLes10` application, and continue with this practice.

Viewing the model: To view the course application model up to this practice, examine the `UML Class Diagram`. This diagram displays all of the classes created up to this point in the course.

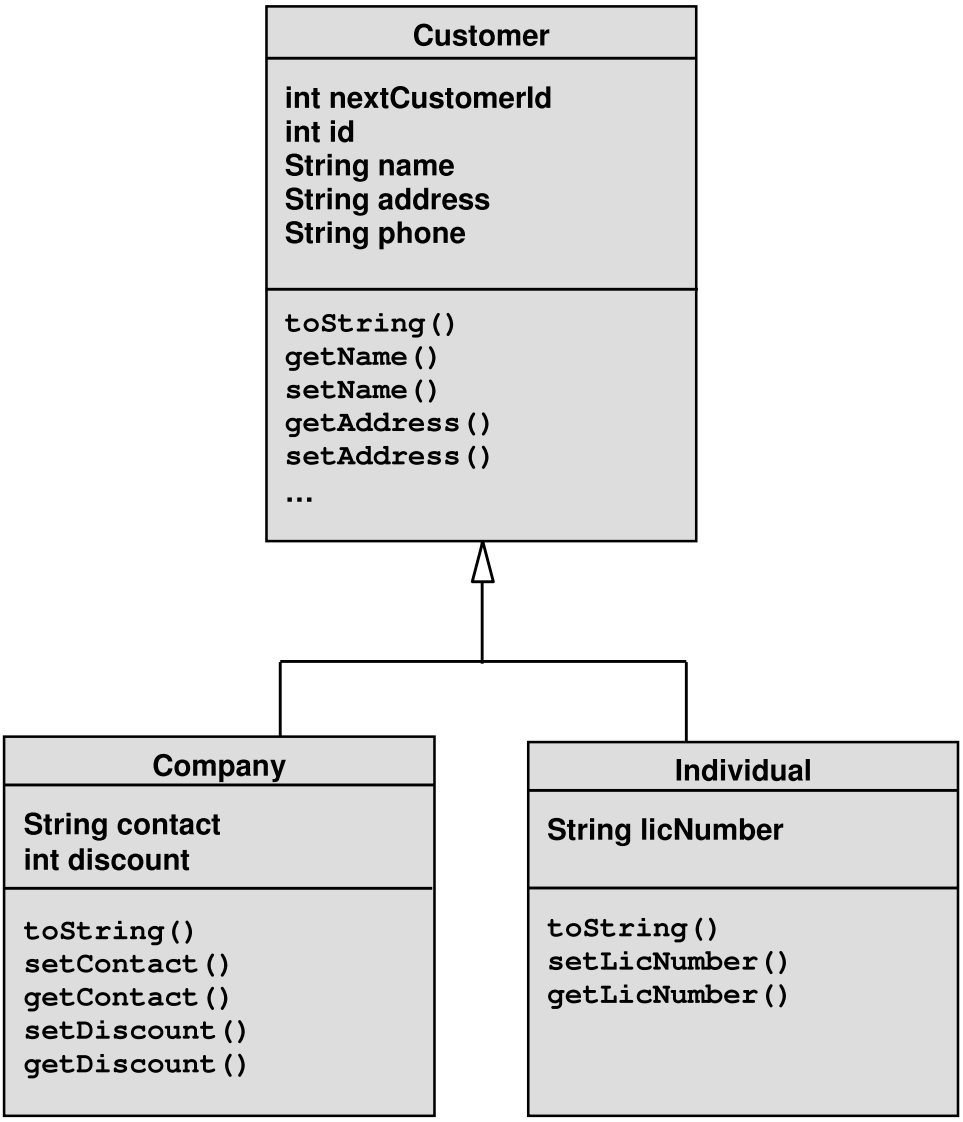
Business Scenario

The owners of the business have decided to expand their business and sell their products to companies as well as individuals. Both are customers, but because companies have slightly different attributes to individuals, there is a need to hold separate company information and individual information. It therefore makes sense to create subclasses for `Company` and `Individual`. Each of the subclasses will have a few of their own methods and will override the `toString()` method of `Customer`.

Your Assignment

Add two new classes as subclasses. The added classes are `Company` and `Individual`, and they both inherit from the `Customer` class. Here is a class diagram to show the relationships between `Customer`, `Company`, and `Individual`. Each box represents a class. The name of the class appears at the top of each box. The middle section specifies the attributes in the class, and underlined attributes represent class variable. The lower section specifies the methods in the class.

Notice the arrow on the line connecting `Company` and `Individual` to `Customer`. This is the UML notation for inheritance.


Customer
int nextCustomerId
int id
String name
String address
String phone
toString()
getName()
setName()
getAddress()
setAddress()
...
Company
String contact
int discount
toString()
setContact()
getContact()
setDiscount()
getDiscount()
Individual
String licNumber
toString()
setLicNumber()
getLicNumber()

Defining a New `Company` Class

1. Define a `Company` class that extends `Customer` and includes the attributes and methods that were defined in the class diagram on the preceding page of this practice.

   a. Right-click the **`OrderEntryProject`** project and select **New** from the context menu. In the New Gallery window, select the General category (if not selected by default) and **Java Class** from the **Items** list. Click **OK**.

   b. In the Create Java Class wizard, enter **`Company`** in the **Name** field, set the package to **`oe`**, and then click the **Browse** button next to the Extends field. In the Class Browser window, click the **Hierarchy** tab and locate and expand the `oe` package. Select the **`Customer`** class and click the **OK** button. The `oe.Customer` class is displayed in the Extends field. Leave the Optional Attributes in their default state and click **OK**. When the source code for the generated class is displayed, save your work.

   c. In the Code Editor for the Company.java file, declare the following attributes:

```
private String contact;
private int discount;
```

   d. To generate the methods for the attributes, right-click in the Code Editor and select **Generate Accessors** from the context menu.

   e. In the **Generate Accessors** dialog, check the check box to the left of the class name. Expand the nodes beside each of the attributes to see the names of the methods that are to be generated for them. Click **OK** to generate the methods.

   f. Save your changes.

2. Modify the `Company` constructor to add arguments.

   a. Add the following arguments to the `no-arg` constructor:

```
public Company(String aName, String aAddress,
               String aPhone, String aContact, int
               aDiscount) { …
}
```

   b. Use the arguments to initialize the object state (including the superclass state). **Hint:** Use the `super(…)` method syntax to pass values to an appropriate superclass constructor to initialize the superclass attributes. Here is an example:

```
super(aName, aAddress, aPhone);
contact = aContact;
discount = aDiscount; ...
```

3. Add a public String `toString()` method in the `Company` class to return the contact name and discount as in the following example: `(Scott Tiger, 20%)`.

   a. Include in the return value the superclass details, and format them as follows: *`<company info>`* `(`*`<contact>`*`, `*`<discount>`*`%)` using the following statement:

```
        return super.toString() + " ("+ contact + ", "+
        discount + "%) ";
```

b. Save and compile the **Company.java** class.

Defining a New Individual Class as a Subclass of Customer

Define an Individual class that extends Customer and includes the attributes and methods that were defined in the class diagram on the preceding page of this practice.

1. Create the **Individual** class as you did for the Company class in step 1.a. Add the **licNumber** attribute as a **String** with a **private** scope, and ensure that the get and set methods are created to retrieve the values.
2. Alter the no-arg constructor to accept four arguments for the name, address, phone, and license number.
3. Complete the constructor body initialization by assigning the arguments to the appropriate instance variables in the Individual class and its superclass.
4. Override the toString() method that is defined in the superclass, and append the license number enclosed in parentheses to the superclass information.
5. Save and compile the **Individual** class.

Modifying the DataMan Class to Include Company and Individual Objects

Add two new class variables to the DataMan class: one for a Company object and the other for an Individual. Open DataMan in the Code Editor and add two new class variables called customer5 and customer6.

1. Create a **Company** variable called customer5, and initialize it by using the Company constructor. Here is an example:

```
static Company customer5 =
newCompany("Oracle","Redw…","80…","Larry…",20);
```

2. Create an **Individual** variable called **customer6** and initialize it using the constructor from the Individual class.
3. Save and compile DataMan.java by right-clicking the file and selecting **Make** from the context menu.

Testing Your New Classes in the OrderEntry Application

1. In these steps you modify the OrderEntry code that assigns a customer object to each of the two order objects in the main() method. You then run the application to see the results of your work.

a. Open **OrderEntry.java** in the Code Editor. Locate the line assigning **`customer3`** with the first **`order`** object.
   For example, find:
   `order.setCustomer(DataMan.customer3);`
   Hint: Press **Ctrl + F** to display a search dialog box.
   Replace `customer3` with `customer5` (the company in `DataMan`).

b. Compile the code and, if successful, explain why the code was successful.

c. Now replace **`customer4`** in the **`order2.setCustomer(DataMan.customer4)`** argument with **`customer6`** (the individual in `DataMan`).

d. Compile and run the `OrderEntry` application. What is displayed in the customer details for each order?
   Explain the results that you see. If you are using the same application you used in the previous practice, remember that if you want the customer information to appear as part of the stored order (2), you need to remove the `transient` modifier on the customer declaration in `Order.java`.

Optional: Refining the `Util` and `Customer` Classes and Testing the Results

It is not obvious to the casual user that the data that is printed for the customer, company, or individual objects represents different objects, unless the user is made aware of the meaning of the subtle differences in the displayed data. Therefore, modify your code to explicitly indicate the object type name in the text that is printed before the rest of the object details, as follows:

```
[Customer] <customer details>
[Company] <company details>
[Individual] <individual details>
```

If you manually add the bracketed text string before the return values of the `toString()` methods in the respective classes, then `[Company]` is concatenated to `[Customer]` and `[Individual]` is concatenated to `[Customer]` for the subclasses of `Customer`. Therefore, the solution is to use inherited code called from the `Customer` class that dynamically determines the run-time object type name.

You can determine the run-time object type name of any Java object by calling its `getClass()` method, which is inherited from the `java.lang.Object` class. The `getClass()` method returns a `java.lang.Class` object reference, through which you can call a `getName()` method returning a `String` containing the fully qualified run-time object name. For example, suppose that you add the following line to the `Customer` class:

```
String myClassName = this.getClass().getName();
```

The variable `myClassName` will contain a fully qualified class name that includes the package name. The value that is stored in `myClassName` will be `oe.Customer`.

Practice 10: Inheritance and Polymorphism (continued)

To extract only the class name, you must strip off the package name and the dot that precedes the class name. This can be done by using a `lastIndexOf()` method in the `String` class to locate the position of the last dot in the package name, and then extract the remaining text thereafter.

1. Add the `getClassName()` method to the `Util` class, and call it from the `toString()` method in the `Customer` class.

   a. Open **`Util.java`** in the Code Editor and add a **`static String getClassName()`** method that determines the run-time object type name and returns only the class name.

   ```
   public static String getClassName(Object o) {
     String className = o.getClass().getName();
     return className.substring(
                    className.lastIndexOf('.')+1,
                    className.length());
   }
   ```

   b. Save and compile **`Util.java`**. Note that JDeveloper automatically recompiles other classes that are dependent on code in `Util.java`. JDeveloper has a class-dependency checking mechanism.

2. Open **`Customer.java`** in the Code Editor.

   a. Prefix a call to the **`Util.getClassName()`** method before the rest of the return value data in the `toString()` method, as follows:

   ```
   return "[" + Util.getClassName(this) + "]
                                          " + id+...;
   ```

   b. Save and compile **`Customer.java`**.

   c. Run the OrderEntry application to view the results.

   d. In the preceding step a, what does this represent? Why do you pass the parameter value **`this`** to the `Util.getClassName()` method?
   Explain why the compiler accepts the syntax.

Practice 11*: Using Arrays and Collections*

Goal

The goal of this practice is to gain experience with Java array objects and work with collection classes such as the `java.util.ArrayList` class. You also work with command-line arguments.

Note: If you have successfully completed the previous practice, continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les11` directory, load the `OrderEntryApplicationLes11` application, and continue with this practice.

Viewing the model: To view the course application model up to this practice, load the `OrderEntryApplicationLes11` application. In the Applications – Navigator node, expand `OrderEntryApplicationLes11 – OrderEntryProjectLes11 – Application Sources – oe` and double-click the `UML Class Diagram1` entry. This diagram displays all of the classes created up to this point in the course.

Your Assignment

Continue to use JDeveloper to build on the application classes from the previous practices. Enhance the `DataMan` class to construct an array of `Customer` objects, and then provide a method to find and return a `Customer` object for a given ID.

The `Order` class needs to be modified to contain an `ArrayList` of order items, requiring a method to add items into the `ArrayList`, and (optionally) another method to remove the items.

Modifying `DataMan` to Hold `Customer` Objects in an Array

1. Modify `DataMan` to build an array of customers.
   a. Define a private static array of `Customer` objects named **`customers`**.
      `private static Customer[] customers;`
   b. Initialize the array to a **`null`** reference.
      `private static Customer[] customers = null;`
2. Create a public static void method called `buildCustomers()` to populate the array of customers. The array must hold six objects using the four `Customer` objects, the `Company` object, and the `Individual` object that you have already created.

a. In the body of the method, first test whether the `customers` variable is not `null`, and if so, return from the method without doing anything because a non-null reference indicates that the customers array has been initialized. If `customers` is `null`, then you must create the array object to hold the six customer objects that are already created.

```
public static void buildCustomers()
{
    if (customers != null) return;
    customers = new Customer[6];
```

b. Now move (cut and paste) the definitions of the four existing **`Customer`** objects, the **`Company`**, and the **`Individual`** into the body of this method, *after* creating the array object. Then, delete the **`static`** keyword and class name or type before each **`customer<n>`** variable name. Modify each variable to be the name of the array variable followed by brackets enclosing an array element number. *Remember, array elements start with a zero base*.
For example, replace:

```
static Customer customer1 = new Customer(…);
```

with:

```
customers[0] = new Customer(…);
```

The example here assigns the customer object to the first element in the array. Repeat this for each **`customer<n>`** object reference in the code.

c. Create a static block that invokes the **`buildCustomers()`** method to create and initialize the array of customer objects, when the `DataMan` class is loaded. Place the block at the end of the `DataMan` class. (Static blocks in the class definition are sometimes called class constructors.)

```
static
    {
buildCustomers();
    }
```

d. Save and compile the **`DataMan`** class. What other classes are compiled? Explain the results. (Only fix errors that are related to the `DataMan` class, if any. Any errors pertaining to the `OrderEntry` class will be fixed after doing the next set of questions).
Hint: Look in the Messages and Compiler tabs in the Log Window.

Modifying `DataMan` to Find a Customer by ID

1. Create a public static method called `findCustomerById(int custId)`, which returns a `Customer` object, where the argument represents the ID of the `Customer` object to be found. If found, return the object reference for the matching `Customer`; otherwise, return a `null` reference value.

a. Why is the customer array guaranteed to be initialized when the `findCustomerById()` method is called? Thus, you can write code assuming that the array is populated.

b. Write a loop to scan through the `customers` array, obtaining each customer object reference to compare the **`custId`** parameter value with the return value from the **`getId()`** method of each customer. If there is a match, return the customer object reference; otherwise, return a null.

c. Save and compile the **`DataMan`** class, only fixing the syntax errors that are reported for the `DataMan` class.

2. Next, fix the syntax errors in the `OrderEntry` class as a result of the changes made to `DataMan`.

   a. In the Code Editor, locate and modify each line that directly refers to the **`DataMan.customer<n>`** variables that previously existed.
   Hint: You can quickly navigate to the error lines by double-clicking the error message line in the Compiler tab of the Log window.
   Replace each occurrence of the **`DataMan.customer<n>`** text with a method call to: **`DataMan.findCustomerById(n)`**. For example, replace:

   ```
   System.out.println(DataMan.customer1.toString());
   ```
   with
   ```
   System.out.println(DataMan.findCustomerById(1).toStrin
   g());
   ```

   b. Save, compile and run the **`OrderEntry.java`** file to test your changes.

Optional: Modifying the Order Class to Hold an `ArrayList` of `OrderItem` Objects

Currently, the `Order` class has hard-coded the creation of two `OrderItem` objects as instance variables, and the details of each `OrderItem` object are set in the `getOrderTotal()` method. This is impractical for the intended behavior of the `Order` class. You must now replace the two `OrderItem` variables with an `ArrayList` that will contain the `OrderItem` objects. Therefore, you must create methods to add and remove `OrderItem` objects to and from the `ArrayList`.

In the `Order` class, define an `ArrayList` of order items, and replace the `OrderItem` instance variables, removing the code dependent on the original `OrderItem` instance variables.

1. Add a statement at the beginning of your class, after the package statement, to

   ```
   import the java.util.ArrayList class
   ```

2. Declare a new instance variable called **`items`** as an **`ArrayList`** object reference. Also remove, or comment out, the declarations of the two instance variables called `item1` and `item2`, and the code that is using these variables.
   Hint: The following methods directly use the `item1` and `item2` variables:

`getOrderTotal()`, `showOrder()`.

3. In the **`Order no-arg` constructor,** add a line to create the item `ArrayList`, as follows:

```
items = new ArrayList(10);
```

4. Save your changes to the `Order` class and compile it.

Optional: Modifying `OrderItem` to Handle Product Information

Before you create the method to add an `OrderItem` object to the `items` ArrayList, you must first modify the `OrderItem` class to hold information about the product being ordered. Each `OrderItem` object represents an order line item. Each order line item contains information about a product that is ordered, its price, and quantity that is ordered.

1. Edit the **`OrderItem`** class and add a new instance variable called **`product`**. Declare the variable as a **`private int`**, and generate or write the **`getProduct()`** method and **`setProduct()`** methods. Modify the **`toString()`** method to add the `product` value between the `lineNbr` and `quantity`.
2. Create an **`OrderItem`** constructor to initialize the object by using values that are supplied from the following two arguments: **`int productId`** and **`double itemPrice`**. Initialize the item **`quantity`** variable to **1**.
   Note: The `OrderItem` class does not provide a `no-arg` constructor.
3. Save and compile the **`OrderItem`** class.

Optional: Modifying `Order` to Add Products into the `OrderItem` ArrayList

In the `Order` class, create a new `public void` method called `addOrderItem()` that accepts one argument, an integer called `product`, representing an ID of the product being ordered. This method must perform the following tasks:

1. Search the **`items`** array list for an **`OrderItem`** containing the supplied product. To do this, create a loop to get each `OrderItem` element from the `items` array list.
   Hint: Use the **`size()`** method of the `ArrayList` object to determine the number of elements in the array list.
2. Use the **`getProduct()`** method of the `OrderItem` class to compare the product value with the existing product value in the order item. If the product with the specified ID is found in an `OrderItem` element from the array list, increment the quantity by using the `setQuantity()` method. If the specified product *does not exist* in any `OrderItem` object in the array list, create a new `OrderItem` object by using the constructor that will accept the product and a price.

Then, add the new `OrderItem` object into the array list.
Note: Because line item numbers are set relative to their order, set the line number for the **`OrderItem`**, by using the **`setLineNbr()`** method, after an item is added to the array list. The line number is set using the **`size()`** of the array list because the elements are added to the end of the array list. For now, assume that all products have a price of $5.00.

```
public void addOrderItem(int product)
{
    OrderItem item = null;
    boolean productFound = false;
    for (int i = 0; i < items.size() &&
!productFound; i++)
   {
        item = (OrderItem) items.get(i);
        productFound = (item.getProduct() ==
product);
   }
   if (productFound)
       {
          item.setQuantity(item.getQuantity() + 1);
       }
       else
           {
              item = new OrderItem(product, 5.0);
              items.add(item);
              item.setLineNbr(items.size());
   }
  }
```

3. The **`orderTotal`** value is now calculated as each product is added to the order. Thus, you must also add the price of each product to **`orderTotal`**.

   Hint: Use the **`getUnitPrice()`** method from the `OrderItem` class. Because the **`orderTotal`** is now updated as each product is added to the order, the **`getOrderTotal()`** method can simply return the **`orderTotal`** value.

   ```
   orderTotal += item.getUnitPrice();
   ```

   Note: This may already be done due to previous changes to the method.

Practice 11: Using Arrays and Collections (continued)

4. Modify the **`showOrder()`** method to use an iteration technique to loop through the **`items`** **array list** to display each **`OrderItem`** object by calling the **`toString()`** method.
   Hint: Import `java.util.Iterator`, and use the array list `elements()` method to create an iteration. See your course notes for an example, or ask your instructor for further clarification.

```
public void showOrder()
{
    System.out.println(toString());
    if (customer != null)
   {
       System.out.println("Customer: " + customer);
    }
    System.out.println("Items:");
    for (Iterator it = items.iterator(); it.hasNext(); )
   {
        System.out.println(it.next().toString());
    }
}
```

5. Save and compile the **`Order`** class and remove any syntax errors.

6. Test your changes to the `OrderItem` and `Order` classes by modifying the **`OrderEntry`** class to add products 101 and 102 to the first order object.
   For example, before the call to **`showOrder()`**, enter the bold lines shown:

```
order.setCustomer(DataMan.findCustomerById(5));
order.addOrderItem(101);
order.addOrderItem(102);
order.addOrderItem(101);
order.showOrder();
```

7. Compile (eliminating syntax errors first), save, and run **`OrderEntry.java`**. Confirm that your results are accurate. For example, verify that the order total for Order 100 is reported as $15.

Practices for Lesson 12

Practice 12*: Using Generic Types*

There is no practice for this lesson.

Practice 13: *Structuring Code Using Abstract Classes and Interfaces*

Goal

The goal of this practice is to learn how to create and use an abstract class and how to create and use an interface.

Note: If you have successfully completed the previous practice, continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les13` directory, load the `OrderEntryApplicationLes13` application, and continue with this practice.

Your Assignment

The `OrderItem` class currently tracks a product only as an integer. This is insufficient for the business, which must know the name, description, and retail price of each product. To meet this requirement, you need to create an abstract class called `Product` and define three concrete subclasses called `Software`, `Hardware`, and `Manual`.

To support the business requirement of computing the sales tax on the hardware products, you create an interface called `Taxable` that is implemented by the `Hardware` subclass.

To test your changes, you modify `DataMan` to build a list of `Product` objects and to provide a method to find a product by its ID.

Creating an Abstract Class and Three Supporting Subclasses

Create a public abstract class called Product in OrderEntryProject.

1. Declare the following attributes and their **`get`*`XXX`*`()`** and **`set`*`XXX`*`()`** methods.
   Note: Remember to add the **`abstract`** keyword before the `class` keyword in the source code after the **`Product.java`** file is created.

   ```
   private static int nextProductId = 2000;
   private int id;
   private String name;
   private String description;
   private double retailPrice;
   ```

2. Define a **`public no-arg`** constructor that assigns the next product ID to the ID of a new product object before incrementing **`nextProductId`**.

   ```
   public Product()
   {
       id = nextProductId++;
   }
   ```

Practice 13: Structuring Code Using Abstract Classes and Interfaces (continued)

3. Add a **`public String toString()`** method to return the **`ID, name`**, and **`retailPrice`**. Prefix with the class name using **`getClassName(this)`** from the **`Util`** class. Then, format **`retailPrice`** with **`Util.toMoney()`.**

```
public String toString()
{
    return "[" + Util.getClassName(this) + "] " + id + "
" + name + " " + Util.toMoney(retailPrice);
}
```

4. Compile and save the **`Product`** class.
5. Create three concrete subclasses of the **`Product`** class, each with attributes and initial values that are listed in the following table. Add the appropriate get and set methods.

Subclass	Attributes
`Software`	`String license = "30 Day Trial";`
`Hardware`	`int warrantyPeriod = 6;`
`Manual`	`String publisher = "Oradev Press";`

6. Modify the **`no-arg`** constructor for the **`Software`, `Hardware`**, and **`Manual`** subclasses to accept three arguments for the product **`name`, `description`,** and **`price`**. Use this code example for the `Software` class as an example:

```
public Software(String name,String desc,double price)
{
  setName(name);
  setDescription(desc);
  setRetailPrice(price);
}
```

7. Compile and save the new subclasses.

Modifying `DataMan` to Provide a List of Products and a Finder Method

Use the new class definitions in the `DataMan` class to build an inventory of products.

1. In `DataMan`, create an object to hold a collection of products.
   a. Create a private static inner class called **`ProductMap`** that extends `HashMap`.
   Note: Remember to import **`java.util.HashMap`**.

Practice 13: Structuring Code Using Abstract Classes and Interfaces (continued)

b. In the **ProductMap** inner class, create the following method to add product objects to the collection. The **Id** is the key and the object reference is the value:

```
public void add(Product p) {
  String key = Integer.toString(p.getId());
  put(key, p);  // use inherited put() method
}
```

c. Declare a **private static ProductMap** variable called **products**, for example:

```
private static ProductMap products = null;
```

d. Compile and save the **DataMan** class.

2. Create a method to populate the `ProductMap` variable with product objects.

a. Create the method called **buildProducts()** in the **DataMan** class as follows:

```
public static void buildProducts() {
  if (products != null) return;
  products = new ProductMap();
  products.add(new Product();
}
```

b. Save and compile your code. Explain the compilation error that is listed for the line adding the `Product` to the `products` map.

c. Fix the compilation error by adding concrete subclasses of the `Product` class. Replace the line of code `products.add(new Product())` with the following text:

```
products.add(
  new Hardware("SDRAM - 128 MB", null, 299.0));
products.add(new Hardware("GP 800x600", null, 48.0));
products.add(
   new Software("Spreadsheet-SSP/V2.0", null, 45.0));
     products.add(
   new Software("Word Processing-
SWP/V4.5",null,65.0));
products.add(
   new Manual("Manual-Vision OS/2x +", null, 125.0));
```

d. Compile the `DataMan` class and save your changes. Your compilation should work this time.

e. At the end of the file, in the static block of **DataMan**, add a call to the **buildProducts()** method.

f. Add the following method called **`findProductById()`** to return a `Product` object matching a supplied ID.

```
public static Product findProductById(int id) {
  String key = Integer.toString(id);
  return (Product) products.get(key);
}
```

Note: Because `products` is a `HashMap`, you simply find the product object by using its key—that is, the Id of the product.

g. Save the changes to the `DataMan` class and compile it.

h. Test the **`DataMan`** code and additional classes by printing the product that is found by its Id. Add the following line to the **`OrderEntry`** class at the end of **`main():`**

```
System.out.println("Product is: " +
                   DataMan.findProductById(2001));
```

i. Compile, save and run the OrderEntry application to test the code.

Optional: Modifying `OrderItem` to Hold `Product` Objects

Replace uses of the `product` variable as an `int` type with the Product class you just created.

1. In **`OrderItem.java`**, change the type declaration for the product instance variable to be **`Product`** instead of **`int.`**
2. Replace the two argument constructors with a single argument called **`newProduct`** whose type is `Product`—that is, remove the **`productId`** and **`itemPrice`** arguments.
3. Change the body of the constructor to store the **`newProduct`** in the **`product`** variable, and set the **`unitPrice`** to be the value that is returned by calling the **`getRetailPrice()`** method of the `product` object.
4. Modify the **`getProduct()`** method to return a **`Product`** instead of an **`int`**, and change the **`setProduct()`** method to accept a **`Product`** instead of an **`int`**.
5. Alter the **`toString()`** method to display the item total instead of the **`unitPrice`**. **Hint:** Use the **`getItemTotal()`** method.
6. Compile and save your code changes. Eliminate syntax errors from the **`OrderItem`** class **only**. Errors that are reported for the `Order` class are corrected in the next step of this lab.

Optional: Modifying `Order` to Add `Product` Objects into `OrderItem`

Alter the `Order.java` class to use the `Product` object instead of an `int` value; to do this, you need to modify the **`addOrderItem()`** method:

Practice 13: Structuring Code Using Abstract Classes and Interfaces (continued)

1. In **addOrderItem**, rename the argument to be `productId`, and in the `for` loop replace:

   ```
   productFound = (item.getProduct() == product);
   ```

   with

   ```
   productFound =
                   (item.getProduct().getId() == productId);
   ```

2. In the **else** section of the `if` statement, call **findProductById()** from `DataMan` by using the **productId** value. If a product object is found, create the `OrderItem` using the `product` object; otherwise, do nothing. Here is an example:

   ```
   item = new OrderItem(product, 5.0);
   items.add(item);
   ```

   becomes:

   ```
   Product p = DataMan.findProductById(productId);
   if (p != null) {
     item = new OrderItem(p);
     items.add(item);
   }
   ```

3. Test the changes that are made to the code supporting the `Product` class and its subclasses by modifying `OrderEntry` to use the new `productId` values.

4. Because the ID of `products` (or its subclasses) starts at 2000, edit the **OrderEntry.java** file, replacing parameter values in all of the calls to the **order.addOrderItem()** method, as shown in the following table:

Replace	With
101	2001
102	2002

5. Save, compile, and run the **OrderEntryProject** project, and check the changes to the printed items. Check whether the price calculations are still correct.

Optional: Creating and Implementing the `Taxable` Interface

1. Create an interface called `Taxable` to compute the sales tax on hardware products. This interface is to be implemented by the `Hardware` subclass. To do this, follow these steps.

   a. Right-click the **OrderEntryProject.jpr** file in the `Navigator` and select **New** from the context menu. In the General category, select **Java Interface** and click **OK**. Enter **Taxable** in the class name and click the **OK** button.

   b. In the Code Editor, add the following variable and method definitions to the interface:

```
double TAX_RATE = 0.10;
     double getTax(double amount);
```

Note: Remember that all variables are implicitly `public static final` and that methods are all implicitly `public`. The implementer of the interface must multiply the `amount`, such as a price, by the `TAX_RATE` and return the result as a `double`.

c. Compile and save the interface.

2. Edit the `Hardware` class to implement the Taxable interface.

   a. Add code to the class definition to implement the interface, as follows:

```
public class Hardware extends Product
 implements Taxable {
```

   b. Compile the **`Hardware`** class and explain the error.

   c. Add the following method to complete the implementation of the interface:

```
public double getTax(double amount) {
return amount * TAX_RATE;
   }
```

   Note: To perform this step, right-click the arrow in the margin to the left of the interface declaration. From the context menu, select **Implement Methods**. JDeveloper generates all of the code except for the return-value calculation (which you can modify appropriately).

   d. Compile and save the **`Hardware`** class.

3. Modify the `OrderItem` class to obtain the tax for each item.

   a. Add a **`public double getTax()`** method to determine whether the **`Product`** in the item is taxable. If the product is taxable, return the tax amount for the item total (use the **`getItemTotal()`** method); otherwise, return 0.0. Here is an example:

```
double itemTax = 0.0;
if (product instanceof Taxable)
{
itemTax =((Taxable) product).getTax(getItemTotal());
}
```

   b. Modify the **`toString()`** method to display the tax amount for the item, if and only if, the product is taxable. Use the **`getTax()`** method that you created, and format the value with **`Util.toMoney()`**.

   c. View the changes by compiling **`OrderItem.java`** and then running OrderEntry.

4. Modify the `Order` class to display the tax; modify the order total to include the tax.

Practice 13: Structuring Code Using Abstract Classes and Interfaces (continued)

a. In the **`showOrder()`** method, add a **`local double`** variable called **`taxTotal`** initialized to **0.0** that accumulates the total tax for the order.

b. Modify the **`for`** loop by using iteration to call the **`getTax()`** method for each item, and add the value to **`taxTotal`**.
Hint: To do this, you must cast the return value of **`it.next()`** to **`OrderItem`**.

c. Add three **`System.out.println()`** statements after the loop: one to print the **`taxTotal`**, the second to print the **`orderTotal`** including **`taxTotal`**, and the last without a parameter to print a blank line. Use the **`Util.toMoney()`** method to format the totals.

d. To view the results, compile and save **`Order.java`**. Then, run **`OrderEntry`**.

Practice 14: *Throwing and Catching Exceptions*

Goal

The goal of this practice is to learn how to create your own exception classes, throw an exception object by using your own class, and handle the exceptions.

Note: If you have successfully completed the previous practice, continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les14` directory, load the `OrderEntryApplicationLes14` application, and continue with this practice.

Your Assignment

The application does not appropriately handle situations when an invalid customer ID is supplied to the `DataMan.findCustomerById()` method, or when an invalid product ID is supplied to the `DataMan.findProductById()` method. In both cases, a `null` value is returned. Your tasks are to:

- Create a user-defined (checked) exception called `oe.NotFoundException`.
- Modify `DataMan.findCustomerById()` to throw the exception when an invalid customer Id is provided.
- Modify `DataMan.findProductById()` in the `DataMan` class to throw the exception if the given product ID is not valid (that is, not found).

Creating the `NotFoundException` Class

In `OrderEntryProject`, create a new class called `NotFoundException`.

1. Right-click the project name in the Navigator, and select **New** from the context menu. From the New Gallery window, ensure that the Category selected is **General** and the Item selected is **Java Class**. Enter the class name **`NotFoundException`**, and make it a subclass of **`java.lang.Exception`**.
2. Modify the default `no-arg` constructor to accept a **message `String`** argument, and pass the string to the superclass constructor, as in the following example:

```
public NotFoundException(String message) {
  super(message);
}
```

3. Compile and save the **`NotFoundException`** class.

Throwing Exceptions in `DataMan`; Finding Methods, and Handling Them in `OrderEntry`

1. Edit `DataMan.java`, and modify the `findCustomerById()` method to throw the `NotFoundException` when the given customer ID is not found in the array.

   a. At the end of the **`for` loop**, if the local customer object reference is `null` (that is, if the customer is not found), throw an **`exception`** object by using the following error message structure in the constructor argument:

   ```
   "Customer with id " + custId + " does not exist"
   ```

   b. Compile the **`DataMan`** class. Explain the error.

   c. Fix the error by modifying the method declaration to propagate the exception.

   d. Compile **`DataMan`** again. What errors do you get this time? Explain the errors.

   e. Fix the compilation errors by handling the exceptions with a **`try-catch` block** in the **`OrderEntry`** class. For simplicity, use one `try-catch` block to handle *all* of the calls to the **`DataMan.findCustomerById()`** methods. Alternatively, if desired, handle each call in its own `try-catch` block.

   ```
   try {  // calls to findCustomerById() here …
   }
   catch (NotFoundException e) {
     // handle the error here …
        }
   ```

   In the **`catch`** block, you can use the exception's inherited methods to display error information. Use the following two methods to display error information:
   - **`e.printStackTrace()`** to display the exception, message, and stack trace
   - **`e.getMessage()`** to return the error message text as a **`String`**.

   f. Compile, save and run **`OrderEntry.java`**. Test your code with the errors.

2. Modify `findProductById()` to throw `NotFoundException` when the supplied product ID is not found in the product map.

   a. The **`findProductById`** method calls **`get(key)`** to find a product from the `HashMap`. If `get(key)` returns **`null`**, throw **`NotFoundException`** by using the following error message; otherwise, return the product object found. You must also add the product declaration line and modify the current return statement.

   ```
   "Product with id " + id + " is not found"
   ```

   b. Modify the **`findProductById`** method to propagate the exception.

   c. Compile and save **`DataMan`** and explain the compile-time error reported.

   d. In the **`Order`** class, modify **`addOrderItem`** to propagate the exception.

   e. Compile the **`Order`** class and explain why it compiles successfully.

Practice 14: Throwing and Catching Exceptions (continued)

f. In **`OrderEntry.java`**, use a value of **9999** as the product ID in the first call to **`order.addOrderItem(2001)`**. Compile and run **`OrderEntry`**. Explain why the application terminates immediately after adding product 9999.

g. In **`Order.java`**, remove **`throws NotFoundException`** from the end of the **`addOrderItem()`** method declaration. Write a try-catch block to handle the exception in this method.
Hint: You must return from the method in the catch block to ensure that the **`itemTotal`** is not affected.

h. Compile **`Order`**, run **`OrderEntry.java`**, and explain the difference in output results.

i. In **`OrderEntry`**, replace the 9999 product ID with 2001. Compile, save, and run.

Practices for Lesson 15

Practice 15*: Using JDBC to Access the Database*

Goal

The goal of this practice is to use the course application to interact with the Oracle database. During this practice, you establish a connection to the database, perform query statements to access the database and retrieve information to integrate into the application.

Note: If you have successfully completed the previous practice, continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les15` directory, load the `OrderEntryApplicationLes15` application, and continue with this practice.

Your Assignment

In the `DataMan` class, connect to the database and retrieve data from the `Customers` table.

Setting Up the Environment to Use JDBC

Update the project to include the necessary JDBC classes.

1. Open the **`OrderEntryApplicationLes15`**, double-click the project name, in this case **`OrderEntryProjectLes15`**, and select the **Libraries and Classpath** node. This opens the libraries pane.
2. Click the **Add Library** button, and then scroll through the list of displayed libraries and select **Oracle JDBC**. Select it and click **OK**. Click **OK** again to close the Project Properties dialog box.
3. Save the project.

Adding JDBC Components to Query the Database

1. Modify the `DataMan` class to provide connection information.
   a. Specify the package(s) to import:

      ```
      import java.sql.*;
      ```

   b. Add a static variable to hold the connection information:

      ```
      private static Connection conn = null;
      ```

   c. After the static block at the end of the **`DataMan`** class, add a new method that queries the Customers table. This method takes the `customer_id` as the input parameter and queries the **`Customers`** table. Start by setting the connection code as follows (your instructor will provide you with the appropriate connection URL):

```
public static Customer selectCustomerById(int id)
throws Exception {
// Register the Oracle JDBC driver
DriverManager.registerDriver(new
oracle.jdbc.OracleDriver());
// Define the connection url
String url = "jdbc:oracle:thin:@myhost:1521:SID";
// Provide db connection information
conn = DriverManager.getConnection (url, "oe", "oe");
     }
```

2. Add statements to execute the select statement based on a customer ID, and return the result to a customer object. In this practice, you populate only two of the items from the database. In the previous practices, the phone number was defined as a `String`. However, in the database it is stored as a complex object type. A utility named JPublisher can be used to convert an object type to a string so that it can be used in your application. This process is much more detailed than can be covered in this course. So in this practice, the phone number item is not populated with any values.

   a. In the **`selectCustomerById`** method, issue the query based on a customer ID:

   ```
   Customer customer = null;
   Statement stmt = conn.createStatement();
   System.out.println
   ("Table Customers query for customer with id: " + id);
   ResultSet rset = stmt.executeQuery
   ("SELECT cust_last_name, nls_territory" +
   " FROM customers WHERE customer_id = " + id);
   ```

   b. If a record is returned, populate the `customer` object:

   ```
   if (rset.next ())       {
   customer = new Customer();
   customer.setId(id);
   customer.setName(rset.getString(1));         //holds
   first column
   customer.setAddress(rset.getString(2));    //holds
   second column
   System.out.println("Customer found: " + customer);
   // prints both columns to the command window
   }
   ```

c. Otherwise throw an exception that the customer is not found, close the statement, and return the customer:

```
else {
throw new NotFoundException("Customer with id " + id +
" not found");
}
rset.close();
stmt.close();
return customer;
}
```

d. Save and compile **DataMan.java.** Correct any errors.

3. Add a new method to close the connection at the end of the **DataMan** class.

a. To do this, add the following code:

```
public static void closeConnection()  {
try {
conn.close();
}
catch (Exception e) {
System.out.println("Error: " + e.getMessage());
 }
}
```

b. Save the **DataMan.java** file.

Testing the JDBC Database Access

1. Up to this point in the course you have been using "hard-coded" customer data. You now modify the code that displays the hard-coded customer information to display "real" customer information from the database. In the **OrderEntry** class, add a call to the `selectCustomerById` method, catch the exception if no customer is found, and close the connection. To do this, follow these steps:

a. Add a call to the **selectCustomerById()** method that you just created. In the **for** loop that displays customer information for the customer IDs specified in the command-line arguments, replace the method **DataMan.findCustomerById(custId)** in the following line:

System.out.println("Arg: " + custId...)

with

DataMan.selectCustomerById(custId);

b. To catch the exception from **DataMan** if no customer is found, scroll down to the **catch** block and replace the line:

catch (NotFoundException e)

with

catch (Exception e)

c. After the catch block, add a call to the **`closeConnection()`** method in the **`DataMan`** class.

d. Save and compile the **`DataMan`** class and correct any errors.

4. Change the customer IDs that you have used until now (these are specified in the project's run/debug configuration), and then test the application.

a. Right-click the project in the Navigator, and select **Project Properties** from the context menu. Select **Run/Debug** in the left pane and click the **Edit** button. In the **Program Arguments** field, replace the existing values with real IDs as follows: **150, 352, 468, 999** (do not use commas to separate the IDs when you type them in the Program Arguments field). Click **OK** and then **OK** again.

b. Run the application. Did all of the IDs return customer information? If not, did you see the exception raised?

Practices for Lesson 16

Practice 16*: Swing Basics for Planning the Application Layout*

Goal

The goal of this practice is to use JDeveloper to start creating the application layout. You create the main application frame as an MDI window and the internal order frames that are contained in the main window. Working with these frames helps you explore Swing classes and the ways to build GUI applications.
Note: For this practice, you use the **`OrderEntryApplicationLes16`** application.

Your Assignment

Start by creating the main window as an extension of the `JFrame` class. This class contains a `JDesktopPane` object to manage the internal frame layout. You also create a class based on the `JInternalFrame` class in which the customer and order details are entered via atomic Swing components. The components layout is managed through the use of panels and associated layout managers. You use the JDeveloper Frame Wizard to create a basic menu for the application, and a status bar in the main application window.

Creating the Main Application Window

1. Start by creating a new subclass of the `JFrame` class in the OrderEntry project.
   a. Right-click the project name in the navigator, and select **New** from the context menu. Select the **Frame** item from the **Client tier > Swing/AWT** category, and click the **OK** button.
   b. In the **Create Frame** dialog, enter the class name **`OrderEntryMDIFrame`**, and extend **`javax.swing.JFrame`**. In the Optional Attributes section, set the **Title** field to: **Order Entry Application**, and select only the **Menu Bar** and **Status Bar** check boxes. Then, click the **OK** button.
   c. Examine the code for the new class that is generated by JDeveloper, by clicking the **Source** tab. Note that JDeveloper creates a **`jbInit()`** method that is called from the default `no-arg` constructor. The `jbInit()` method should contain all code to initialize the user interface structure. You should modify the code if required, to match with the one displayed.
   d. In the Editor window, click the **Design** tab and examine the visual container hierarchy and presentation of the frame. The container hierarchy is visible in the Structure window (located under the Navigator).
   e. Return to the Code Editor, by clicking the **Source** tab and make the following changes:
      - Replace the **`JPanel panelCenter`** variable declaration, with:
        `JDesktopPane desktopPane = new JDesktopPane();`
        Note: You may need to import **`javax.swing.JDesktopPane`**.

- In the **`jbInit()`** method, replace **`panelCenter`** references with **`desktopPane`**.

f. Save and compile the **`OrderEntryMDIFrame`** class.

2. Make the frame visible. To do this:

   a. Modify **`OrderEntry.java`** by renaming the **`main()`** method to **`test1(…)`**.

   b. At the end of the class, create a new **public static void main(String[] args)** method, which creates an instance of the OrderEntryMDIFrame and makes it visible.

3. Compile, save, and run the OrderEntry application.

Creating the `JInternalFrame` Class for Order Data

This frame contains the bulk of the UI code for data entry and user interaction for an order, and for assigning a customer and adding items to the order.

1. Create the JInternal Frame and name it OrderEntryFrame.

   a. Navigate to the **Client Tier > Swing/AWT** node and select the **Frame** item. In the **Create Frame** dialog box, enter the class name **`OrderEntryFrame`**, and extend the **`javax.swing.JInternalFrame`**. Set the **title** to **`Order`**, and then click **OK**.
   Note: The `JInternalFrame` class can be selected by clicking the Browse button.
   Note: Once you select `javax.swing.JInternalFrame` in the Extends field, the Title field becomes 'grayed out' and is no longer enterable (I think this is a small bug). To get around this, you either need to enter the title into the Title field **before** selecting `javax.swing.JInternalFrame` in the Extends field, or enter the title via the Property Inspector after having created the frame.

   b. The OrderEntryFrame that is generated does not have the desired layout manager or content pane. To set the layout manager and to add a panel to the content pane, open **`OrderEntryFrame`** with the UI Editor.
   Note: When the UI Editor is activated (the UI Editor can be invoked by clicking the Design tab), the Property Inspector window is also displayed, showing the properties of the object that is selected in the UI Editor.

   c. Select the **`frame`** object by clicking the frame title bar in the UI Editor or the node labeled **`this`** in the structure window. (You may have to expand the UI node to view objects in the containment hierarchy.) In the Property Inspector window, locate the **layout** property and select **BorderLayout** from the pop-up list options. (**Note:** The layout property is under the Visual node).

Practice 16: Swing Basics for Planning the Application Layout (continued)

d. Examine lines of code that JDeveloper added or changed in your class by clicking the Source tab. When creating a Swing UI using the JDeveloper UI Editor, it is wise to view changes that are made to the source code as an aid to learning what you would need to write yourself when building the UI manually. Remove the **`private`** declaration from **`BorderLayout`**.

2. Add a `JPanel` to the frame.

   a. Return to the Design view. JDeveloper provides a Component Palette in the toolbar (ask the instructor, if needed).

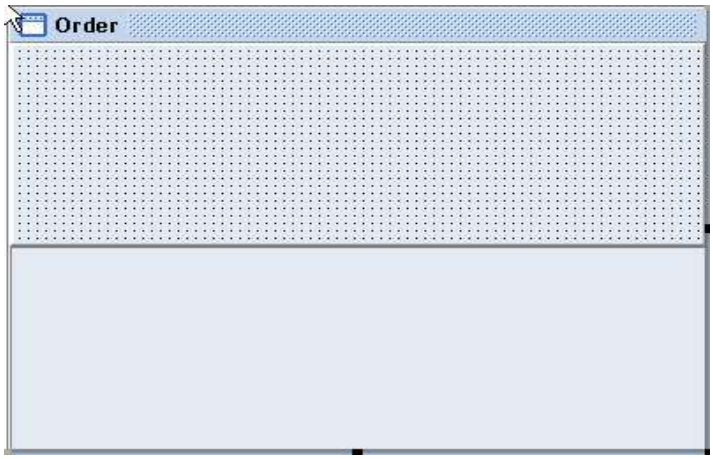
   b. In the Component Palette, select **Swing** from the list, and in the **Containers** sub-group, click the `JPanel` icon, and then click the center of the frame in the UI Editor (or click the node labeled **`this`** in the UI Structure pane). This adds a new panel to the center region of the border layout.
   Note: If the JPanel icon is not visible, expand the Component Palette window by increasing the height.

3. Divide the `JPanel` into two sections by using a GridLayout for the layout, with one column and two rows.

   a. Select **`JPanel1`**, and set its layout property to **GridLayout**.

   b. Expand **JPanel1** in the UI Structure pane, select the **`gridLayout1`** object, and in the **Model** node, set the **columns** property to **1** and set the **rows** property to **2**.

4. Using the following picture as a guide, add another panel to the top and a scroll pane to the bottom of the content panel.



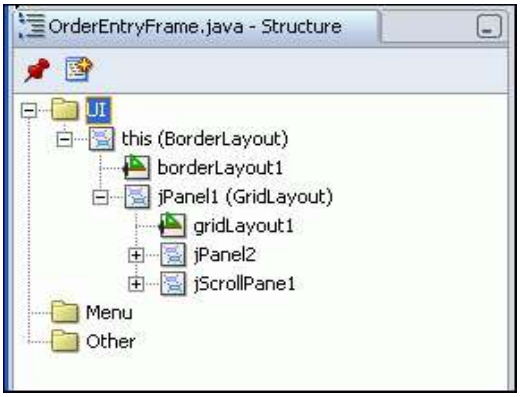
   a. Add a second panel to the top half (or first row) of the first panel by clicking the **`JPanel`** icon in the **Swing Components** palette, and then clicking the **`jPanel1`** object in the UI Editor or in the Structure pane.
   Note: Confirm that the new panel is called **`jPanel2`**, and more importantly, that it is nested inside **`jPanel1`** in the hierarchy.

Practice 16: Swing Basics for Planning the Application Layout (continued)

b. Add a raised-bevel border to the new panel, **`jPanel2`**, by selecting its **border** property in the Property Inspector and selecting **Swing Border** from the pop-up list. In the **Border dialog box**, select **BevelBorder** and select the **RAISED** option button, and then click the **OK** button.
Note: `jPanel2` should visually occupy the top half of the `jPanel1`.

c. Add a scroll pane to the bottom half (second row) of **`jPanel1`** by clicking the **JScrollPane** button in the **Swing Components** palette and clicking the bottom area of the `jPanel1`. (Alternatively, click the `jPanel1` object in the Structure pane to add the JScrollPane.)

d. Use the Structure window to ensure that you have the following containment hierarchy:



e. Save and compile the **`OrderEntryFrame`** class.

Modifying `OrderEntryMDIFrame` Class to Contain an Internal `OrderEntryFrame`

1. To view the visual results of your internal frame at run time, modify the constructor in `OrderEntryMDIFrame` to create an instance of `OrderEntryFrame`, and then make it visible. To do this, follow these steps.

   a. Edit **`OrderEntryMDIFrame.java`** and, at the end of the constructor, add the following lines of code:

   ```
   OrderEntryFrame iFrame = new OrderEntryFrame();
   iFrame.setVisible(true);
   desktopPane.add(iFrame);
   ```

   Note: The bounds (size and location) of the internal frame must now be set; otherwise, it does not become visible. In addition, you must also alter the dimensions of **`OrderEntryMDIFrame`** to be larger than the initial size of the internal **`OrderEntryFrame`**.

   b. In the jbInit() method of the **OrderEntryMDIFrame** class, locate the following statement:
   ```
   this.setSize(new Dimension(400,300));
   ```

   Then, modify the dimension arguments to be **`700,500`**.

c. Switch to the **`OrderEntryFrame`**, and add the following line to the **`jbInit()`** method:

```
this.setBounds(0, 0, 400, 300);
```

d. Compile and save **`OrderEntryMDIFrame`** and **`OrderEntryFrame`**.

e. Run **`OrderEntry.java`** to view the results.

2. Notice that the internal frame cannot be maximized, "iconified" (minimized), or closed. Make changes to `OrderEntryFrame` to enable these features.

a. In the **`jbInit()`** method, add the following lines of code to enable the internal frame to be maximized, "iconified," and closed:

```
this.setMaximizable(true);
this.setIconifiable(true);
this.setClosable(true);
```

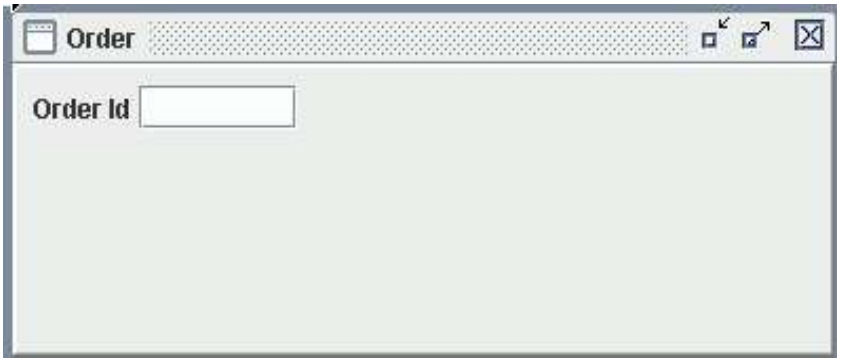
b. Compile and save the changes to **`OrderEntryFrame.java`**.

c. Run the application and notice the changes.

Adding UI Components to `OrderEntryFrame`

1. Before adding UI components to `jPanel2` in OrderEntryFrame, set its layout to `null`.
Note: You could also use the JDeveloper `XYLayout`.
In either case, JDeveloper uses absolute positioning and sizing for components that are added to the panel. It is easier to use absolute positioning when building the initial UI layout. You change the layout again in a subsequent lesson.

Use the following image as a guide to the desired results:



a. In Design mode, select the **Swing** option in the Component Palette pop-up list. Then, add a **`Jlabel`** to **`jPanel2`** and set its **text** property to **Order Id**. Resize the label to see the label value, if needed. What lines of code have been added to your class?

Hint: You should find at least five lines of code (some of them in the `jbInit()` method). Try to identify the three that make the object visible in the

panel.
Note: The `setBounds` value can be modified (if required) in the source to make the label clearly visible.

b. From the Swing page of the Component Palette, select a **`JTextField`** component and add it to **`jPanel2`** (to the right of the label).
Note: `setBounds` values can be changed if required.

c. Compile and save **`OrderEntryFrame`**, and then run **`OrderEntry`** to view the results.

Practices for Lesson 17

Practice 17-1*: Adding User Interface Components*

Goal

In this practice, you create the menu and visual components so that users can enter order details. The application includes a button to find the customer assigned to the order, and buttons to add and remove products as items in the order. You learn how to build a Swing-based UI application by using the JDeveloper UI Editor to construct the user interface. You also learn how to handle events for the Swing components that are added to the application.

Note: Whenever you create a UI component, JDeveloper declares it as private, but you can remove that if required.

Note: If you have successfully completed the previous practice, continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les17` directory, load the `OrderEntryApplicationLes17` application, and continue with this practice.

The UI for the Order Entry Application

The slide in Lesson 17 of your course manual shows a snapshot of the final visual appearance of the application's main window, `OrderEntryMDIFrame`, and a sample `OrderEntryFrame` for an order that is created as an internal frame.

Using the Application

`OrderEntryMDIFrame` provides the main application menu, from which users select the Order > New menu option to create a new order for a customer.

The new order request should create the internal `OrderEntryFrame` and a new `Order` object (whose ID sets the Order Id text field in the frame).

Customer details are entered by providing an ID value in the Customer ID field and clicking the Find button. The Find button event validates whether the customer exits (by using the `DataMan.findCustomerById()` method). When the event validates, it assigns the customer to the order and displays the customer details in the fields provided; otherwise, an error message is displayed.

Products are added to the order by entering a value in the Product ID field and clicking the Add button. Products are found by using the `DataMan.findProductById()` method. They are then added to the order contained in the order item objects that are added to the `JList` in the bottom pane of the `OrderEntryFrame`. Multiple products can be added to the order, but adding a product that already exists in the order increments the item quantity.

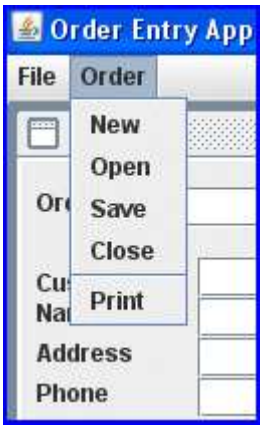
Practice 17-1: Adding User Interface Components (continued)

Your Assignment

Use the following screenshot as a guide to modify the menu of `OrderEntryMDIFrame` and add several Swing components to `OrderEntryFrame` to meet user requirements:

Creating the `OrderEntryMDIFrame` Menu

1. The menu structure that is added to the main window should look like the following screenshot:



Use the JDeveloper UI Editor to modify the menu to include the Order menu and its menu items, as shown in the preceding menu.
Note: File > Exit already exists.

a. Edit the **`OrderEntryMDIFrame`** with the UI Editor. Expand the **Menu** item in the Structure pane, and click the **menuBar** entry to display the initial menu structure in the UI Editor window.

b. Add the **Order** menu to the right of the **File** menu. Right-click the *outlined* box on the right side of the File menu item, and select the **Insert Menu** option from the context menu.

c. The new menu should be selected and shown as **`jMenu1`**. With the menu selected, enter the text **`Order`**, overwriting the default menu label text, and then press the **Enter** key.

d. Save your work and compile the class, ensuring that there are no compilation errors. What lines has JDeveloper added to the `OrderEntryMDIFrame` class?

2. Add menu items and a separator to the Order menu, as per the following diagram.

Menu Item `text`
`New`
`Open`
`Save`
`Close`
`<separator>`
`Print`

a. In the Structure window, click the **Order** option that you just created, select the blank outline box at the bottom of the Order menu, and enter the menu label text **New** in the box. Do the same for other menu items in the table. To add a separator (a line that separates menu items), right-click and select **Insert Separator** from the context menu.

b. Save and compile the class. Then, run **`OrderEntry.java`** to view the menu.

Adding Components to `OrderEntryFrame` to Form Its Visual Structure

1. Add text fields and labels for the customer details. Add a Find button for finding a customer by the ID, and add an area in the bottom part of the frame where order information will be displayed.

a. In the top panel, add **`JLabel`** and **`JTextField`** components for the **Customer** details. (These components can be found in the **Swing** list.) Use the sample window on the previous page as a guide for the layout. Create label and text field items as per the following table:

`JLabel` **text** property
Customer Id
Name
Address
Phone

Hint: Aligning UI components works best with **`XYLayout`** **Manager**, in which alignment is relative to the first component clicked. Select additional components

while holding Shift or Control. Right-click a selected component and select an Align option from the context menu.

If you are using the `null` layout manager, JDeveloper generates calls to each component `setBounds()` method, with a `Rectangle` parameter defining the components x, y location, width, and height. You can alter the parameters (x, y, width, height) in the `Rectangle` constructor to manually align and size components, or you can set the `bounds` property for each component in the Inspector.

b. Add a **`JButton`** to the right of the **customer ID** text field, and then set the **text** property to **Find**. Then, save your work.

c. Add a **`JList`** component to the scroll pane in the bottom panel of the `OrderEntryFrame`. The list component should fill the entire bottom section of the frame (just click in the lower pane, and the JList expands and takes up the entire pane).

If you have time…

The following steps take you through adding more Order information to the frame as well as adding and removing products. The process is obviously very similar to what you have already done when adding Customer information to the frame. Therefore, if you are short of time, skip these steps. To see what the completed OrderEntryFrame looks like and how the Add Product functionality works, open `OrderEntryApplication17-2` and run `OrderEntry.java` You can also use this application to start the next practice.

1. Add components for the order information and for the addition of products to an order.

   a. Create a **`JLabel`** and set the **text** property to **Product ID**. To the right of the label, add a **`JTextField`**. Below the product ID label and text field, create a **`JButton`** component, and set the **text** property to **Add**.

   b. Add two **`JLabel`** components at the upper-right side of the top panel, for the order date. Set the first label **text** property to **Order Date**. Set the second label's **text** property to the empty string. In the **Border** property, select the **Swing Border** option in the border property, and then, from the Border dialog box, select the **LineBorder** value. Set the **border thickness** to **1** if it is not already set.

   c. Add two more **`JLabel`** components under the order date labels, for the order total. Set the first label's **text** property to **Order Total**. Set the second label's **text** property to the empty string. In the **Border** property, select the **Swing Border** option in the border property, and then, from the Border dialog box, again select the **LineBorder** value. Set the border thickness to **1** if it is not already set.

   d. Save and compile the OrderEntryFrame class.

   e. Run the OrderEntry application to view the resulting UI layout in the internal frame. Quickly make minor adjustments to the UI layout to make all items clearly visible.

Practice 17-2*: Adding Event Handling*

Goal

In this practice, you create the order entry details. You add event handling code for the Order > New menu, the Find Customer button, and the Add Product and button.

Note: If you have successfully completed the previous practice, continue using the same directory and files. If you did not complete the practice or if the compilation from the previous practice was unsuccessful and you would like to move on to this practice, change to the `les17-2` directory, load the `OrderEntryApplicationLes17-2` application, and continue with this practice.

Your Assignment

In the previous practice you created the frames to display customer and order information. You now add functionality to the frames, so that a user can display an order, find a customer, and add a product to an order.

Adding Event Handling for the Order > New Menu

1. Modify `OrderEntryFrame.java` in the Source Editor to create an order object and display its initial state in the appropriate components. To do this, do the following:
   a. Create a new instance variable for the order object as follows:
   ```
   Order order = null;
   ```
   b. Add the following method to create a new order object and display its contents in the appropriate components in the frame:
   ```
   private void initOrder() {
    order = new Order();
    jTextField1.setText(
              Integer.toString(order.getId()));
     jLabel8.setText(
      Util.toDateString(order.getOrderDate()));
     jLabel10.setText(
               Util.toMoney(order.getOrderTotal()));
              }
   ```
   Note: The preceding italics represent identifiers for labels and text fields added in the last practice. If you did not add the labels and text fields in the same sequence as the practice steps, you will need to modify the preceding code to refer to the identifiers you have used for the labels and text fields. For example, `jTextField1` refers to the Order Id text field. If you need to check the identifier you have used, select the Order Id field in the Design view and refer to the Property Inspector to see the identifier you have used for that component.
   c. Call the **`initOrder()`** method at end of the **`jbInit()`** method.
   d. To control the x, y location of the upper-left corner of the frame, when it is displayed, declare the following instance and class variables:

```
private static int x = 0;
private static int y = 0;
private static final int OFFSET = 20;
private static final int MAX_OFFSET = 200;
```

and create the following method, to create a cascading effect as new order frames are created:

```
private void setBounds() {
this.setResizable(true);
this.setBounds(x, y,
this.getWidth(), this.getHeight());
x = (x + OFFSET) % MAX_OFFSET;
y = (y + OFFSET) % MAX_OFFSET;
}
```

e. Add a call to your **`setBounds()`** method at the end of the **`jbInit()`** method, after calling **`initOrder()`**.

f. Add one more method to **`OrderEntryFrame`** to make it the active window as follows:

```
public void setActive(boolean active) {
     try {
          this.setSelected(active);
     }
     catch (Exception e) {}
     this.setVisible(active);
     if (active) {
          this.toFront();
     }
}
```

Note: This method will be called from the Order > New menu event handler.

g. Compile and save the **`OrderEntryFrame`** class.

2. Modify `OrderEntryMDIFrame.java` to create the event handler code for the new order menu option.

   a. Open **`OrderEntryMDIFrame.java`** in the UI Editor, expand the menu either in the Visual Editor or the Structure pane, and then select the **New** menu item under the **Order** menu.

b. Click the **Events** node at the bottom of the Properties Inspector window, click in the text area to the right of the first event called **`actionPerformed`**. The text area will show a button with three dots (ellipses). Click this button to display the `actionPerformed` event generation dialog box. Take note of the **name of the action method** and accept the defaults, and then click the **OK** button.

   Note: JDeveloper generates the event listener code as an anonymous inner class (in the `jbInit()` method) that calls the method that is named in the event dialog window. JDeveloper will position the cursor in the Code Editor inside the empty body of the event handler method created.

c. Move the following lines from the **`OrderEntryMDIFrame()`** constructor to the body of the **`jMenuItem1_actionPerformed()`** method, deleting (or commenting out) the line, making the frame visible, as shown:

```
OrderEntryFrame iFrame = new OrderEntryFrame();
        // iFrame.setVisible(true);
        desktopPane.add(iFrame);
```

   Also add the following line, after adding the frame to the desktop pane in the **`jMenuItem1_actionPerformed`** method:

```
iFrame.setActive(true);
```

d. Compile the **`OrderEntryMDIFrame`** class and save the changes. Run and test the `OrderEntry` application by selecting the **Order > New** menu.
   Note: When the application first starts there should not be any order window displayed. Close the internal window by clicking its Close icon (X).

If you are short of time…

The following steps take you through adding event handling for the **Find Customer** button. However if you are short of time, omit the steps and open the OrderEntryApplication17-2Sol application in the Solutions folder, and run `OrderEntry.java` to see how the finished functionality should work.

Adding Event Handling for the Find Customer Button

In this section of the code, you add event handling functionality to the Find button that allows you to display details of a valid customer.

1. Modify `OrderEntryFrame.java` by adding code to do the following:

   – **test** if the customer ID text field has a non-zero length string, and convert it to an integer used in the `DataMan.findCustomerById()` method to return a valid customer. If the customer ID field is empty, or is not a number, the `DataMan.findCustomerById()` method should throw a `NotFoundException`.

   – **display** an error message using the `javax.swing.JOptionPane` class.

– If the customer is a valid customer, **associate** the customer object with the order and display the customer details in the field that is provided in `OrderEntryFrame`.

The following steps guide you through these tasks.

a. Select the **Find** button and click the **Events** tab in the Property Inspector. Click the ellipses to generate the skeleton code for the **`actionPerformed`** event.

b. In the body of the generated **`jButton1_actionPerformed()`** method, add the following code:

```
int custId = 0;
     Customer customer = null;
     if (jTextField5.getText().length() > 0) {
          try {
     custId = Integer.parseInt(jTextField5.getText());
     customer = DataMan.findCustomerById(custId);
               order.setCustomer(customer);
     jTextField3.setText(customer.getName());
       jTextField4.setText(customer.getAddress());
       jTextField2.setText(customer.getPhone());
            }
          catch (NumberFormatException err) {
               JOptionPane.showMessageDialog(this,
              "The Customer Id: " + err.getMessage() +
              " is not a valid number",
              "Error", JOptionPane.ERROR_MESSAGE);
              jTextField2.setText("");
              }
        catch (NotFoundException err) {
               JOptionPane.showMessageDialog(this,
                         err.getMessage(),
               "Error", JOptionPane.ERROR_MESSAGE);
               jTextField2.setText("");
             }
       }
       else {
            JOptionPane.showMessageDialog(this,
              "Please enter a Customer Id", "Error",
              JOptionPane.ERROR_MESSAGE);
       }
```

Note: As before, pay close attention to the preceding variables in italic type to ensure that you have correctly identified the appropriate labels and text fields.

c. Compile and save your changes. Run the `OrderEntry` application to test your code changes (customer IDs range from 1 to 6).

Additional Extra Credit: Adding Event Handling for the Add Product Button

In this section you write code to add products to the order.

1. Modify `OrderEntryFrame.java` by adding code to do the following:
   - Read the product ID that is entered and supply it to the `order.addOrderItem()` method.
   - Update the Order Total field with the latest total after each product is added to the order.
   - Handle errors as appropriate.

   The following steps assist you with these tasks.

   a. Select the **Add** button and create its **`actionPerformed`** event handler using the following code:

```
Product p = null;
            int prodId = 0;
if (jTextField6.getText().length() > 0) {
               try {
      prodId =
     Integer.parseInt(jTextField6.getText();
                           p =
DataMan.findProductById(prodId);
                order.addOrderItem(p.getId());
                jLabel10.setText(

Util.toMoney(order.getOrderTotal()));
                 }
             catch (Exception err) {
                String message = err.getMessage();
                if (err instanceof
                NumberFormatException) {
                  message = "Product id '" +
                message +
              "' is not a valid number";
                }
             JOptionPane.showMessageDialog(this,
     message,"Error", JOptionPane.ERROR_MESSAGE);
             jTextField6.setText("");
                }
             }
             else {
               JOptionPane.showMessageDialog(this,
```

```
                    "Please enter a Product Id",
            "Error",JOptionPane.ERROR_MESSAGE);
              }
                                        }
```

Note: As before, pay close attention to the preceding variables in italics to ensure that you have correctly identified the appropriate labels and text fields.

b. Compile and save the code. Run the **OrderEntry** application to test the code. Add products to the order (product IDs start at 2000). Did you see the products visually added to the list? If not, explain why. Did the order total get updated?

2. Modify the Order class to support the UI by replacing the `Vector` type for `items` to be a `javax.swing.DefaultListModel`. Provide a method in the `Order` class to return the reference to the model.

a. Modify **Order.java** class and replace the `items` declaration as shown:

```
// private ArrayList items = null;
```

replace with ...

```
private DefaultListModel jList1 = null;
```

Note: You will need to import **javax.swing.DefaultListModel**.

b. In the **Order** `no-arg` constructor, create the **DefaultListModel** object to initialize the **items** variable, instead of using a Vector, for example:

```
// items = new ArrayList(10);
jList1 = new DefaultListModel();
```

In the **addOrderItem()** method, comment out the following statement:

```
//items.add(item);
```

replace with…

```
jList1.addElement(item);
```

In the **showOrder()** method, comment out the **for** loop statements:

```
/*  for (Iterator it = items; ...) {
          ...
      System.out.println(item.toString());
}*/
```

c. Add a new method with the signature shown to return the `items` reference to the caller:

```
public DefaultListModel getModel() { … }
```

Note: This method will be used as the model for the `JList` causing it to dynamically display `OrderItem` objects as products are added to the order.

d. Modify `OrderEntryFrame` to add the call to use the method in the button.

```
jList1.setModel(order.getModel());
```

Practice 17-2: Adding Event Handling (continued)

e. Compile and save the changes to the **`Order`** class.

f. Save and compile **`OrderEntryFrame`**, and run the **`OrderEntry`** application to test if items are dynamically displayed in the list as they are added.

Practices for Lesson 18

Practice 18*: Deploying Java Applications*

There is no practice for this lesson.