

Java SE 7 Programming

Activity Guide

D67238GC20

Edition 2.0

June 2012

D74998

ORACLE

Authors

Michael Williams

Tom McGinn

Matt Heimer

**Technical Contributors
and Reviewers**

Peter Hall

Marnie Knue

Lee Klement

Steve Watts

Brian Earl

Vasily Strelnikov

Andy Smith

Nancy K.A.N

Chris Lamb

Todd Lowry

Ionut Radu

Joe Darcy

Brian Goetz

Alan Bateman

David Holmes

Editors

Richard Wallis

Daniel Milne

Vijayalakshmi Narasimhan

Graphic Designer

James Hans

Publishers

Syed Imtiaz Ali

Sumesh Koshy

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.**Disclaimer**

This document contains proprietary information and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except where your use constitutes "fair use" under copyright law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Restricted Rights Notice

If this documentation is delivered to the United States Government or anyone using the documentation on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Table of Contents

Practices for Lesson 1: Introduction	1-1
Practices for Lesson 1: Overview.....	1-2
Practice 1-1: Verifying Software Installation	1-3
Practice 1-2: Software Installation.....	1-5
Practice 1-3: Configuring NetBeans 7.0.1 to Utilize JDK 7	1-7
Practices for Lesson 2: Java Syntax and Class Review	2-1
Practices for Lesson 2: Overview.....	2-2
Practice 2-1: Summary Level: Creating Java Classes.....	2-3
Practice 2-1: Detailed Level: Creating Java Classes	2-5
Practices for Lesson 3: Encapsulation and Subclassing	3-1
Practices for Lesson 3: Overview.....	3-2
Practice 3-1: Summary Level: Creating Subclasses.....	3-3
Practice 3-1: Detailed Level: Creating Subclasses.....	3-6
(Optional) Practice 3-2: Adding a Staff to a Manager	3-11
Practices for Lesson 4: Java Class Design.....	4-1
Practices for Lesson 4.....	4-2
Practice 4-1: Summary Level: Overriding Methods and Applying Polymorphism	4-3
Practice 4-1: Detailed Level: Overriding Methods and Applying Polymorphism	4-6
Practices for Lesson 5: Advanced Class Design	5-1
Practices for Lesson 5: Overview.....	5-2
Practice 5-1: Summary Level: Applying the Abstract Keyword	5-3
Practice 5-1: Detailed Level: Applying the Abstract Keyword	5-7
Practice 5-2: Summary Level: Applying the Singleton Design Pattern	5-12
Practice 5-2: Detailed Level: Applying the Singleton Design Pattern	5-14
(Optional) Practice 5-3: Using Java Enumerations.....	5-16
(Optional) Practice 5-4: Recognizing Nested Classes	5-18
(Optional) Solution 5-4: Recognizing Nested Classes	5-19
Practices for Lesson 6: Inheritance with Java Interfaces	6-1
Practices for Lesson 6: Overview.....	6-2
Practice 6-1: Summary Level: Implementing an Interface	6-3
Practice 6-1: Detailed Level: Implementing an Interface	6-6
Practice 6-2: Summary Level: Applying the DAO Pattern.....	6-11
Practice 6-2: Detailed Level: Applying the DAO Pattern.....	6-13
(Optional) Practice 6-3: Implementing Composition	6-18
Practices for Lesson 7: Generics and Collections.....	7-1
Practices for Lesson 7: Overview.....	7-2
Practice 7-1: Summary Level: Counting Part Numbers by Using HashMaps.....	7-3
Practice 7-1: Detailed Level: Counting Part Numbers by Using HashMaps.....	7-5
Practice 7-2: Summary Level: Matching Parentheses by Using a Deque.....	7-7
Practice 7-2: Detailed Level: Matching Parentheses by Using a Deque	7-8
Practice 7-3: Summary Level: Counting Inventory and Sorting by Using Comparators	7-10
Practice 7-3: Detailed Level: Counting Inventory and Sorting by Using Comparators	7-13
Practices for Lesson 8: String Processing.....	8-1
Practices for Lesson 8: Overview.....	8-2
Practice 8-1: Summary Level: Parsing Text with split()	8-3

Practice 8-1: Detailed Level: Parsing Text with split()	8-5
Practice 8-2: Summary Level: Creating a Regular Expression Search Program	8-8
Practice 8-2: Detailed Level: Creating a Regular Expression Search Program	8-9
Practice 8-3: Summary Level: Transforming HTML by Using Regular Expressions	8-11
Practice 8-3: Detailed Level: Transforming HTML by Using Regular Expressions	8-13
Practices for Lesson 9: Exceptions and Assertions	9-1
Practices for Lesson 9: Overview	9-2
Practice 9-1: Summary Level: Catching Exceptions	9-3
Practice 9-1: Detailed Level: Catching Exceptions	9-6
Practice 9-2: Summary Level: Extending Exception	9-9
Practice 9-2: Detailed Level: Extending Exception	9-13
Practices for Lesson 10: Java I/O Fundamentals	10-1
Practices for Lesson 10: Overview	10-2
Practice 10-1: Summary Level: Writing a Simple Console I/O Application	10-3
Practice 10-1: Detailed Level: Writing a Simple Console I/O Application	10-5
Practice 10-2: Summary Level: Serializing and Deserializing a ShoppingCart	10-8
Practice 10-2: Detailed Level: Serializing and Deserializing a ShoppingCart	10-11
Practices for Lesson 11: Java File I/O (NIO.2)	11-1
Practices for Lesson 11: Overview	11-2
Practice 11-1: Summary Level: Writing a File Merge Application	11-3
Practice 11-1: Detail Level: Writing a File Merge Application	11-6
Practice 11-2: Summary Level: Recursive Copy	11-10
Practice 11-2: Detailed Level: Recursive Copy	11-12
(Optional) Practice 11-3: Summary Level: Using PathMatcher to Recursively Delete	11-15
(Optional) Practice 11-3: Detailed Level: Using PathMatcher to Recursively Delete	11-17
Practices for Lesson 12: Threading	12-1
Practices for Lesson 12: Overview	12-2
Practice 12-1: Summary Level: Synchronizing Access to Shared Data	12-3
Practice 12-1: Detailed Level: Synchronizing Access to Shared Data	12-6
Practice 12-2: Summary Level: Implementing a Multithreaded Program	12-10
Practice 12-2: Detailed Level: Implementing a Multithreaded Program	12-12
Practices for Lesson 13: Concurrency	13-1
Practices for Lesson 13: Overview	13-2
(Optional) Practice 13-1: Using the java.util.concurrent Package	13-3
(Optional) Practice 13-2: Using the Fork-Join Framework	13-5
Practices for Lesson 14: Building Database Applications with JDBC	14-1
Practices for Lesson 14: Overview	14-2
Practice 14-1: Summary Level: Working with the Derby Database and JDBC	14-3
Practice 14-1: Detailed Level: Working with the Derby Database and JDBC	14-5
Practice 14-2: Summary Level: Using the Data Access Object Pattern	14-7
Practice 14-2: Detailed Level: Using the Data Access Object Pattern	14-10
Practices for Lesson 15: Localization	15-1
Practices for Lesson 15: Overview	15-2
Practice 15-1: Summary Level: Creating a Localized Date Application	15-3
Practice 15-1: Detailed Level: Creating a Localized Date Application	15-5
Practice 15-2: Summary Level: Localizing a JDBC Application (Optional)	15-8

NAVEEN UNNIKRIISHNAN (naveenunnikrishnan3.14@gmail.com)
has a non-transferable license to use this Student Guide.

NAVEEN UNNIKRIISHNAN (naveenunnikrishnan3.14@gmail.com)
has a non-transferable license to use this Student Guide.

Practices for Lesson 1: Introduction

Chapter 1

NAVEEN UNNIKRIISHNAN (naveenunnikrishnan14@gmail.com)
has a non-transferable license to use this Student Guide.

Practices for Lesson 1: Overview

Practices Overview

These practices cover configuring a development environment for Java SE 7. These practices should not be performed unless directed to do so by your instructor.

Practice 1-1: Verifying Software Installation

Overview

In this practice, you will verify the installation of the necessary software to perform Java 7 software development. If verification fails, you will proceed with the remaining practices.

Assumptions

Your instructor has instructed you to perform these steps.

Summary

You have been given a computer system that will be used for Java SE 7 software development. You must validate that the Java 7 SE Development Kit is installed, the NetBeans 7.0.1 IDE is installed, and the NetBeans IDE is correctly configured to use JDK 7.

Tasks

1. Open a command or terminal window.

Note: If you are using Windows, you can open a command window by performing the following steps:

- a. Click the Start button.
- b. Click Run.
- c. Type `cmd` in the Run dialog box and click the OK button.

2. Execute the `java -version` command. This verifies that a *JRE* is installed; this does not verify that the *JDK* is installed.

- a. Verify that the output of the `java -version` command matches the following example output. For 64-bit machines, the output should be:

```
java version "1.7.0"
Java(TM) SE Runtime Environment (build 1.7.0-b147)
Java HotSpot(TM) 64-Bit Server VM (build 21.0-b17, mixed mode)
```

For 32-bit machines, the output should be:

```
java version "1.7.0"
Java(TM) SE Runtime Environment (build 1.7.0-b147)
Java HotSpot(TM) Client VM (build 21.0-b17, mixed mode, sharing)
```

- b. If a different version number or an error message is displayed, you may have one of the following possible problems:
 - The JRE/JDK is not installed.
 - The `java` command is not in your path.
 - The wrong JRE/JDK version is installed.
 - Multiple JREs/JDKs are installed.

Note: To exclude an incorrect PATH environment variable as the reason for an incorrect or unrecognized Java version, you may attempt to locate the path to your JDK and execute `java -version` by using a fully qualified path. For example:

```
"C:\Program Files\Java\jdk1.7.0\bin\java.exe" -version
```

3. Execute the `javac -version` command. This verifies that a JDK is installed.
 - a. Verify that the output of the `javac -version` command matches the following example output:

```
javac 1.7.0
```

- b. If a different version number or an error message is displayed, you may have one of the following possible problems:
 - The JDK is not installed.
 - The `javac` command is not in your path.
 - The wrong JDK version is installed.
 - Multiple JDKs are installed.

Note: It is very common that the directory containing `javac` is not listed in your PATH environment variable. You do not need to modify the PATH for most IDEs to function; instead, locate the path to your JDK and execute `javac -version` by using a fully qualified path to verify the presence and version of the JDK. For example:

```
"C:\Program Files\Java\jdk1.7.0\bin\javac.exe" -version
```

4. Start the NetBeans IDE and verify the version number of the JDK used by the IDE.
 - a. The installation of NetBeans places a menu in your Start menu. Locate the NetBeans IDE 7.0.1 shortcut within the Start menu and click it.
 - b. Within NetBeans, click the Help menu, and then click About.
 - c. The About dialog box displays both the NetBeans and JDK version numbers that are being used. For 64-bit machines, you should see:

```
Product Version: NetBeans IDE 7.0.1 (Build 201107282000)
Java: 1.7.0; Java HotSpot(TM) 64-Bit Server VM 21.0-b17
```

For 32-bit machines, you should see:

```
Product Version: NetBeans IDE 7.0.1 (Build 201107282000)
Java: 1.7.0; Java HotSpot(TM) Client VM 21.0-b17
```

Note: Even if JDK7 was discovered in a previous step, you must verify that NetBeans is using Java 1.7.0.

- d. Click the Close button to close the About dialog box.

Note: NetBeans 7.0.1 was the first version of NetBeans to fully support the final release of JDK 7.

Practice 1-2: Software Installation

Overview

In this practice, you will install the software needed to perform Java 7 SE development.

Assumptions

Your instructor has instructed you to perform these steps.

Summary

You have been given a computer system that will be used for Java SE 7 software development. You will use the information obtained in the previous practice to determine the software to be installed, and then you will install the software.

A JDK and NetBeans cobundle is available that will reduce the number of files to download. Downloading the JDK and NetBeans separately provides greater flexibility for selecting the JDK to be used (32-bit or 64-bit) and reduces the required amount of data to download if either the JDK or NetBeans is already installed.

For more information about Oracle JDK 7 supported operating systems, go to <http://www.oracle.com/technetwork/java/javase/config-417990.html>. Java 7 support for additional operating systems may be available from other channels.

Tasks

1. Obtain the required software.
 - a. If you require both the JDK and NetBeans, the easiest method is to download the “JDK 7 with NetBeans 7.0.1” cobundle.
 - 1) Using a web browser, go to <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.
 - 2) Locate the Java SE Development Kit (JDK) Cobundles table.
 - 3) Click the Download button for “JDK 7 with NetBeans 7.0.1.”
 - 4) You must accept the JDK 7 and NetBeans 7.0.1 Cobundle License Agreement to download the software.
 - 5) Download the file for your operating system. As of this writing, Linux, Solaris, and Windows downloads are available.
 - b. If you require just the JDK, download the Java SE 7 JDK.
 - 1) Using a web browser, go to <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.
 - 2) Locate the Java Platform, Standard Edition table.
 - 3) Click the Download button for Java SE 7 JDK.

Note: Be sure to download the *JDK* and not the *JRE*.

 - 4) You must accept the JDK 7 and NetBeans 7.0.1 Cobundle License Agreement to download the software.
 - 5) Download the file for your operating system. As of this writing, Linux, Solaris, and Windows downloads are available.

- c. If you require just the NetBeans IDE, download “NetBeans IDE 7.0.1 for Java SE.”
 - 1) Using a web browser, go to <http://download.netbeans.org/netbeans/7.0.1/final/>.
 - 2) Click the Download button for Java SE.
 - 3) If the download does not start automatically, click the “download it here” link.
2. Install the required software.
 - Install the software downloaded in the previous step.
 - The JDK includes optional demos and a database known as the JavaDB. Install all optional JDK components if you are installing the JDK.
 - If you downloaded the JDK and NetBeans separately (not the cobundle), complete the installation of the JDK before installing NetBeans.
 - If you have multiple JDK versions installed, be sure to select the Java SE 7 JDK if prompted while installing NetBeans.
 - If NetBeans 7.0.1 was installed before JDK 7 or an older version of the JDK was selected during NetBeans installation, perform practice 1-3 to reconfigure NetBeans to use JDK 7.
3. Verify the software installation.
 - Repeat practice 1-1 to verify software installation.

Practice 1-3: Configuring NetBeans 7.0.1 to Utilize JDK 7

Overview

In this practice, you will configure NetBeans 7.0.1 to use a locally installed instance of JDK 7.

Assumptions

Your instructor has instructed you to perform these steps.

Summary

These steps are required only if both NetBeans 7.0.1 and JDK 7 are installed but NetBeans is running with the wrong JDK. You can verify the JDK being used by NetBeans with the steps outlined in practice 1-1, step 4. Failure to correct the NetBeans configuration to use JDK 7 will result in an inability to create Java projects that use Java 7 language features.

Tasks

1. Configure NetBeans to be aware of the Java 7 Platform.
 - a. In the NetBeans IDE, select Tools, and then Java Platforms from the Main menu.
 - b. Click the Add Platform button and specify the directory that contains your JDK 7 installation.
 - c. In the Platform Name step, verify that the default locations of the Platform Sources zip file and API documentation are valid.
 - d. Click the Finish button to close the Add Java Platform dialog box.
 - e. Ensure that JDK 1.7 is selected in the Platforms list and click Close.
2. Configure NetBeans to start with Java SE 7 JDK.
 - a. Open the directory containing the NetBeans configuration files, typically: C:\Program Files\NetBeans 7.0.1\etc\.
 - b. Use a text editor to edit the `netbeans.conf` file.
 - c. Modify the `netbeans_jdkhome` property to have a value of the JDK 7 installation location, for example: `netbeans_jdkhome="C:\Program Files\Java\jdk1.7.0"`.
3. Restart NetBeans and verify the JDK being used by NetBeans with the steps outlined in practice 1-1, step 4.

NAVEEN UNNIKRIISHNAN (naveenunnikrishnan3.14@gmail.com)
has a non-transferable license to use this Student Guide.

Practices for Lesson 2: Java Syntax and Class Review

Chapter 2

Practices for Lesson 2: Overview

Practices Overview

In these practices, you will use the NetBeans IDE and create a project, create packages, and a Java main class, and then add classes and subclasses. You will also run your project from within the IDE and learn how to pass command-line arguments to your main class.

Note: There are two levels of practice for most of the practices in this course. Practices that are marked “Detailed Level” provide more instructions and, as the name implies, at a more detailed level. Practices that are marked “Summary Level” provide less detail, and likely will require additional review of the student guide materials to complete. The end state of the “Detailed” and “Summary” level practices is the same, so you can also use the solution end state as a tool to guide your experience.

Practice 2-1: Summary Level: Creating Java Classes

Overview

In this practice, using the NetBeans IDE, you will create an `Employee` class, create a class with a `main` method to test the `Employee` class, compile and run your application, and print the results to the command line output.

Tasks

1. Start the NetBeans IDE by using the icon from Desktop.
2. Create a new project `EmployeePractice` in the `D:\labs\02-Review\practices` directory with an `EmployeeTest` main class in the `com.example` package.
3. Set the Source/Binary format to JDK 7.
 - a. Right-click the project and select Properties.
 - b. Select JDK 7 from the drop-down list for Source/Binary Format.
 - c. Click OK.
4. Create another package called `com.example.domain`.
5. Add a Java Class called `Employee` in the `com.example.domain` package.
6. Code the `Employee` class.
 - a. Add the following data fields to the `Employee` class—use your judgment as to what you want to call these fields in the class. Refer to the lesson materials for ideas on the field names and the syntax if you are not sure. Use `public` as the access modifier.

Field use	Recommended field type
Employee id	<code>int</code>
Employee name	<code>String</code>
Employee Social Security Number	<code>String</code>
Employee salary	<code>double</code>

7. Create a no-arg constructor for the `Employee` class.
 NetBeans can format your code at any time for you. Right-click anywhere in the class and select Format, or press the Alt-Shift-F key combination.
8. Add accessor/mutator methods for each of the fields.
 Note that NetBeans has a feature to create the getter methods for you. Click in your class where you want the methods to go, then right-click and choose Insert Code (or press the Alt-Insert keys). Choose getters from the Generate menu, and click the boxes next to the fields for which you want getter and setter methods generated.

9. Write code in the `EmployeeTest` class to test your `Employee` class.
 - a. Construct an instance of `Employee`.
 - b. Use the setter methods to assign the following values to the instance:

Field	Value
Employee id	101
Employee name	Jane Smith
Employee Social Security Number	012-34-4567
Employee salary	120_345.27

- c. In the body of the main method, use the `System.out.println` method to write the value of the employee fields to the console output.
 - d. Resolve any missing import statements.
 - e. Save the `EmployeeTest` class.
10. Run the `EmployeePractice` project.
11. (Optional) Add some additional employee instances to your test class.

Practice 2-1: Detailed Level: Creating Java Classes

Overview

In this practice, using the NetBeans IDE, you will create an `Employee` class, create a class with a `main` method to test the `Employee` class, compile and run your application, and print the results to the command-line output.

Tasks

1. Start the NetBeans IDE by using the icon from Desktop.
2. Create a new Project called `EmployeePractice` in NetBeans with an `EmployeeTest` class and a `main` method.
 - a. Click File > New Project.
 - b. Select Java from Categories, and Java Application from Projects.
 - c. Click Next.
 - d. On the New Application screen, enter the following values:

Window/Page Description	Choices or Values
Project Name:	EmployeePractice
Project Location	D:\labs\02-Review\practices
Use Dedicated Folder for Storing Libraries	Deselected
Create Main Class	Selected Change the name to <code>com.example.EmployeeTest</code> – <code>com.example</code> is the package name.
Set as Main Project	Selected

- e. Click Finish.

You will notice that NetBeans has saved you a great deal of typing by creating a class called `EmployeeTest`, including the package name of `com.example`, and writing the skeleton of the `main` method for you.
3. Set the Source/Binary format to JDK 7.
 - a. Right-click the project and select Properties.
 - b. Select JDK 7 from the drop-down list for Source/Binary Format.
 - c. Click OK.

4. Create another package called `com.example.domain`.
 - a. Right-click the current package `com.example` under Source Packages.
 - b. Select **New > Java Package**. The New Java Package dialog box highlights the new subpackage name.
 - c. Enter `com.example.domain` in the Package Name field, and click Finish.

You will notice that the icon beside the package name is gray in the Project—this is because the package has no classes in it yet.

5. Create a new Java Class called `Employee` in the `com.example.domain` package.
 - a. Right-click the `com.example.domain` package and select **New > Java Class**.
 - b. In the Class Name field, enter `Employee`.
 - c. Click Finish to create the class.

Notice that NetBeans has generated a class with the name `Employee` in the package `com.example.domain`.

6. Code the `Employee` class.
 - a. Add the following data fields to the `Employee` class.

Field use	Access	Recommended field type	Field name
Employee id	public	int	empId
Employee name	public	String	name
Employee Social Security Number	public	String	ssn
Employee salary	public	double	salary

- b. Add a constructor to the `Employee` class:

```
public Employee() { }
```

NetBeans can format your code at any time for you. Right-click anywhere in the class and select **Format**, or press the **Alt-Shift-F** key combination.

- c. Create accessor/mutator (getter/setter) methods for each of the fields.

Note that NetBeans has a feature to create the getter methods for you. Click in your class where you want the methods to go, then right-click and choose **Insert Code** (or press the **Alt-Insert** keys). Select **"Getter and Setter"** from the **Generate** menu, and click the boxes next to the fields for which you want getter and setter methods generated. You can also click the class name (`Employee`) to select all fields. Click **Generate** to insert the code.

The built-in automated syntax checker for NetBeans should have provided you with hints when you have syntax errors or code errors. Save your class.

7. Write code in the `EmployeeTest` main class to test your `Employee` class.

- a. Add an import statement to your class for the `Employee` object:

```
import com.example.domain.Employee;
```

- b. In the main method of `EmployeeTest`, create an instance of your `Employee` class, like this:

```
Employee emp = new Employee();
```

- c. Using the employee object instance, add data to the object using the setter methods. For example:

```
emp.setEmpId(101);
emp.setName("Jane Smith");
emp.setSsn ("012-34-5678");
emp.setSalary(120_345.27);
```

Note that after you type the `emp.`, Netbeans provides you with suggested field names (in green) and method names (in black) that can be accessed via the `emp` reference you typed. You can use this feature to cut down on typing. After typing the dot following `emp`, use the arrow keys or the mouse to select the appropriate method from the list. To narrow the list down, continue typing some of the first letters of the method name. For example, typing `set` will limit the list to the method names that begin with `set`. Double-click the method to choose it.

- d. In the body of the main method, use the `System.out.println` method to write messages to the console output.

```
System.out.println ("Employee id:      " + emp.getEmpId());
System.out.println ("Employee name:      " + emp.getName());
System.out.println ("Employee Soc Sec #: " + emp.getSsn());
System.out.println ("Employee salary:    " + emp.getSalary());
```

The `System` class is in the `java.lang` package, which is why you do not have to import it (by default, you always get `java.lang`). You will learn more about how this multiple dot notation works, but for now understand that this method takes a string argument and writes that string to the console output.

- e. Save the `EmployeeTest` class.

8. Examine the Project Properties.

- Right-click the project and select Properties.
- Expand Build, if necessary, and select Compiling. The option at the top, "Compile on Save," is selected by default. This means that as soon as you saved the `Employee` and `EmployeeTest` classes, they were compiled.
- Select Run. You will see that the Main Class is `com.example.EmployeeTest`. This is the class the Java interpreter will execute. The next field is Arguments, which is used for passing arguments to the main method. You will use arguments in a future lesson.
- Click Cancel to close the Project Properties.

9. Run the `EmployeePractice` project.

- a. To run your `EmployeePractice` project, right-click the project and select Run, or click the Run Main Project icon (the green triangle), or press F6.
- b. If your classes have no errors, you should see the following output in the Output window:

```
run:
Employee id:      101
Employee name:    Jane Smith
Employee Soc Sec #: 012-34-5678
Employee salary:  120345.27
BUILD SUCCESSFUL (total time: 1 second)
```

10. (Optional) Add some additional employee instances to your test class.

Practices for Lesson 3: Encapsulation and Subclassing

Chapter 3

Practices for Lesson 3: Overview

Practices Overview

In these practices, you will extend your existing Employee class to create new classes for Engineers, Admins, Managers, and Directors.

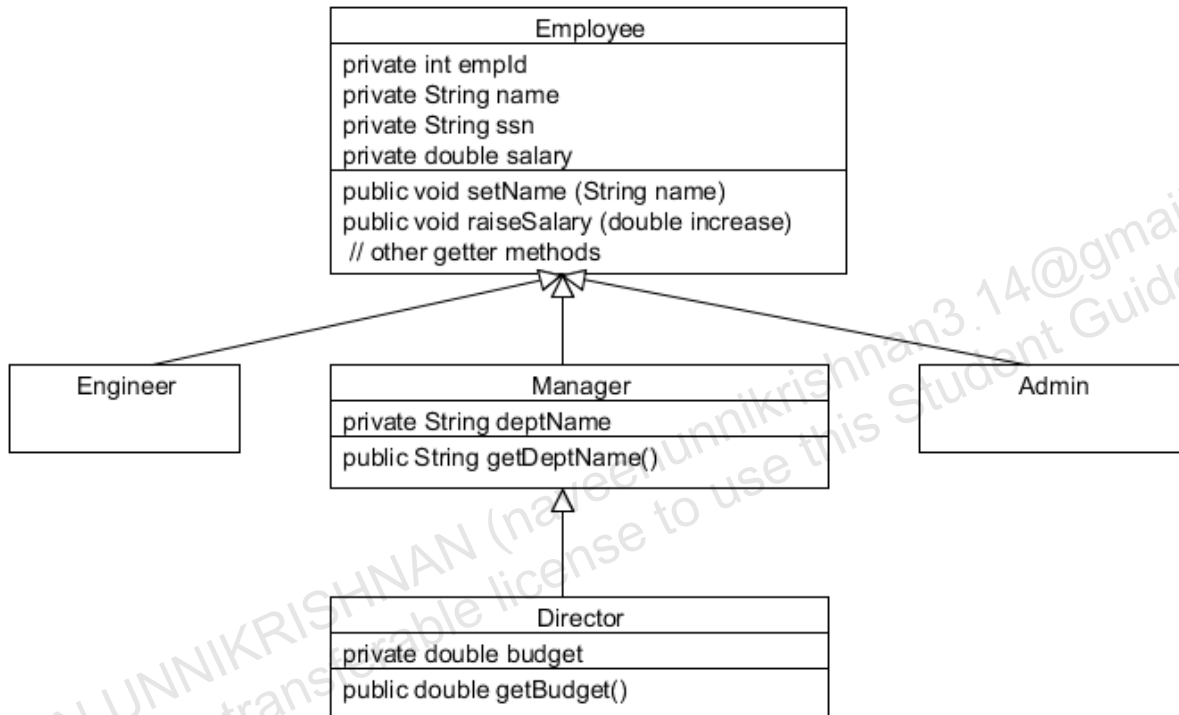
Practice 3-1: Summary Level: Creating Subclasses

Overview

In this practice, you will create subclasses of `Employee`, including `Manager`, `Engineer`, and `Administrative assistant (Admin)`. You will create a subclass of `Manager` called `Director`, and create a test class with a `main` method to test your new classes.

Assumptions

Use this Java class diagram to help guide this practice.



Tasks

1. Open the project `EmployeePractice` in the `practices` directory.
2. Apply encapsulation to the `Employee` class.
 - a. Make the fields of the `Employee` class private.
 - b. Replace the no-arg constructor in `Employee` with a constructor that takes `empId`, `name`, `ssn`, and `salary`.
 - c. Remove all the setter methods except `setName`.
 - d. Add a method named `raiseSalary` with a parameter of type `double` called `increase` to increment the salary.
 - e. Save `Employee.java`.

3. Create a subclass of `Employee` called `Manager` in the same package.
 - a. Add a private `String` field to store the department name in a field called `deptName`.
 - b. Create a constructor that includes all the parameters needed for `Employee` and `deptName`.
 - c. Add a getter method for `deptName`.
4. Create subclasses of `Employee`: `Engineer` and `Admin` in the `com.example.domain` package. These do not need fields or methods at this time.
5. Create a subclass of `Manager` called `Director` in the `com.example.domain` package.
 - a. Add a private field to store a double value `budget`.
 - b. Create a constructor for `Director` that includes the parameters needed for `Manager` and the `budget` parameter.
 - c. Create a getter method for this field.
6. Save all the classes.
7. Test your subclasses by modifying the `EmployeeTest` class. Have your code do the following:
 - a. Remove the code that creates an instance of the “Jane Smith” `Employee`.
 - b. Create an instance of an `Engineer` with the following information:

Field	Choices or Values
ID	101
Name	Jane Smith
SSN	012-34-5678
Salary	120_345.27

You will likely see an error beside the line that you added to create an `Engineer`. This is because NetBeans cannot resolve `Engineer` using the existing import statements in the class. The quick way to fix import statements is to allow NetBeans to fill them in for you. Right-click in the class and select `Fix Imports`, or press the `Ctrl + Shift + I` key combination. NetBeans will automatically add the import statement for `Engineer` in the appropriate place in the class and the error will disappear.

- c. Create an instance of a `Manager` with the following information:

Field	Choices or Values
ID	207
Name	Barbara Johnson
SSN	054-12-2367
Salary	109_501.36
Department	US Marketing

- d.

Create an instance of an `Admin` with the following information:

Field	Choices or Values
ID	304
Name	Bill Monroe
SSN	108-23-6509
Salary	75_002.34

e. Create an instance of a `Director`:

Field	Choices or Values
ID	12
Name	Susan Wheeler
SSN	099-45-2340
Salary	120_567.36
Department	Global Marketing
Budget	1_000_000.00

f. Save `EmployeeTest` and correct any syntax errors.

8. Add a `printEmployee` method to `EmployeeTest` to print out a formatted `Employee` object (its data fields). The `printEmployee` method should take an `Employee` object as a parameter.
9. Use the `printEmployee` method to print out information about each of your `Employee` objects.
10. (Optional) Use the `raiseSalary` and `setName` methods on some of your objects to make sure that those methods work.
11. Save the `EmployeeTest` class and test your work.
12. (Optional) Improve the look of the salary print output using the `NumberFormat` class.
 - a. Use the following code to get an instance of a static `java.text.NumberFormat` class that you can use to format the salary to look like a standard US dollar currency:


```
NumberFormat.getCurrencyInstance().format(emp.getSalary())
```

In the lesson on abstract classes, you will see how to use an abstract factory, such as `NumberFormat.getCurrencyInstance()`.
13. (Optional) Add additional business logic (data validation) to your `Employee` class.
 - a. Prevent a negative value for the `raiseSalary` method.
 - b. Prevent a null or empty value for the `setName` method.

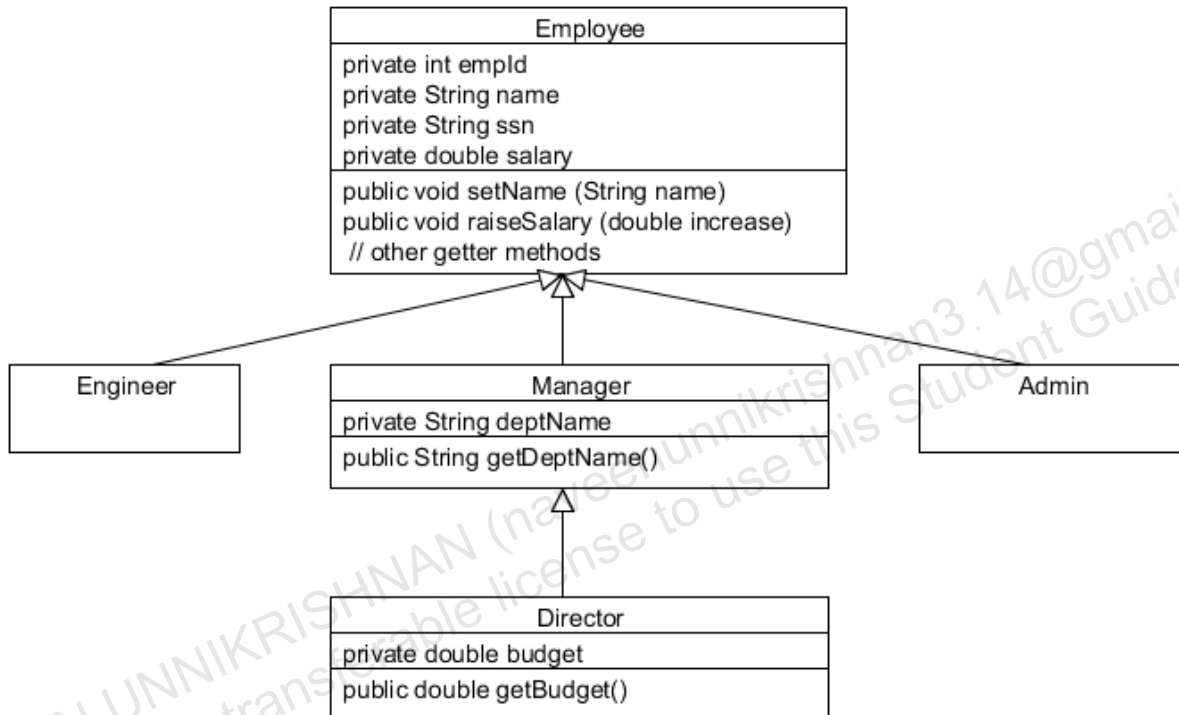
Practice 3-1: Detailed Level: Creating Subclasses

Overview

In this practice, you will create subclasses of `Employee`, including `Manager`, `Engineer`, and `Administrative assistant (Admin)`. You will create a subclass of `Manager` called `Director`, and create a test class with a `main` method to test your new classes

Assumptions

Use this Java class diagram to help guide this practice.



Tasks

1. Open the project `EmployeePractice` in the practices directory.
 - a. Select `File > Open Project`
 - b. Browse to `D:\labs\03-Encapsulation\practices`.
 - c. Select `EmployeePractice`.
 - d. Click `Open Project`.
2. Apply encapsulation to the `Employee` class.
 - a. Open the `Employee` class in the editor.
 - b. Make the fields of the `Employee` class private.

- c. Replace the no-arg constructor in `Employee` with a constructor that takes `empId`, `name`, `ssn`, and `salary`.

```
public Employee(int empId, String name, String ssn, double salary) {
    this.empId = empId;
    this.name = name;
    this.ssn = ssn;
    this.salary = salary;
}
```

- d. Remove all the setter methods except `setName`.
- e. Add a method named `raiseSalary` with a parameter of type `double` named `increase` to increment the salary

```
public void raiseSalary(double increase) {
    salary += increase;
}
```

- f. Save `Employee.java`

3. Create a subclass of `Employee` called `Manager`.

- Right-click the package `com.example.domain` and select `New > Java Class`.
- Enter the class name `Manager` and click `Finish`.
- Modify the class to subclass `Employee`.

Note that the class declaration now has an error mark on it from Netbeans. Recall that constructors are not inherited from the parent class, so you will need to add a constructor that sets the value of the fields inherited from the parent class. The easiest way to do this is to write a constructor that calls the parent constructor using the `super` keyword.

- Add a private `String` field to store the department name in a field called `deptName`.
- Add a constructor that takes `empId`, `name`, `ssn`, `salary`, and a `deptName` of type `String`. The `Manager` constructor should call the `Employee` constructor with the `super` keyword, and then set the value of `deptName`.

```
public Manager(int empId, String name, String ssn, double salary, String deptName) {
    super (empId, name, ssn, salary);
    this.deptName = deptName;
}
```

- Add a getter method for `deptName`.
- Save the `Manager` class.

4. Create two subclasses of `Employee`: `Engineer` and `Admin` in the `com.example.domain` package. These do not need fields or methods at this time.

- Because `Engineers` and `Admins` are `Employees`, add a constructor for each of these classes that will construct the class as an instance of an `Employee`.
Hint: Use the `super` keyword as you did in the `Manager` class.
- Save the classes.

5. Create a subclass of `Manager` called `Director` in the `com.example.domain` package.
 - a. Add a private field to store a double value `budget`.
 - b. Add the appropriate constructors for `Director`. Use the `super` keyword to construct a `Manager` instance and set the value of `budget`.
 - c. Create a getter method for `budget`.
6. Save the class.
7. Test your subclasses by modifying the `EmployeeTest` class. Have your code do the following:
 - a. Remove the code that creates an instance of the “Jane Smith” `Employee`.
 - b. Create an instance of an `Engineer` with the following information:

Field	Choices or Values
ID	101
Name	Jane Smith
SSN	012-34-5678
Salary	120_345.27

You will likely see an error beside the line that you added to create an `Engineer`. This is because NetBeans cannot resolve `Engineer` using the existing import statements in the class. The quick way to fix import statements is to allow NetBeans to fill them in for you. Right-click in the class and select `Fix Imports`, or press the `Ctrl + Shift + I` key combination. NetBeans will automatically add the import statement for `Engineer` in the appropriate place in the class and the error will disappear.

- c. Create an instance of a `Manager` with the following information:

Field	Choices or Values
ID	207
Name	Barbara Johnson
SSN	054-12-2367
Salary	109_501.36
Department	US Marketing

- d. Create an instance of an `Admin` with the following information:

Field	Choices or Values
ID	304
Name	Bill Monroe
SSN	108-23-6509
Salary	75_002.34

- e.

Create an instance of a Director:

Field	Choices or Values
ID	12
Name	Susan Wheeler
SSN	099-45-2340
Salary	120_567.36
Department	Global Marketing
Budget	1_000_000.00

- f. Save `EmployeeTest` and correct any syntax errors.
8. Add a `printEmployee` method to `EmployeeTest`.
 - a. Adding `System.out.println` methods after each of the instances you created is going to create a lot of redundant code. Instead, you will use a method that takes an `Employee` object as the parameter:

```
public static void printEmployee (Employee emp) {
    System.out.println(); // Print a blank line as a separator
    // Print out the data in this Employee object
    System.out.println ("Employee id:      " + emp.getEmpId());
    System.out.println ("Employee name:   " + emp.getName());
    System.out.println ("Employee Soc Sec #: " + emp.getSsn());
    System.out.println ("Employee salary:  " + emp.getSalary());
}
```

Note that all the object instances that you are creating are `Employee` objects, so regardless of which subclass you create, the `printEmployee` method will work. However, the `Employee` class cannot know about the specialization of its subclasses. You will see how to work around this in the next lesson.

9. Use the `printEmployee` method to print out information about your classes. For example:

```
printEmployee(eng);
printEmployee(man);
printEmployee(adm);
printEmployee(dir);
```

10. (Optional) Use the `raiseSalary` and `setName` methods on some of your objects to make sure those methods work. For example:

```
mgr.setName ("Barbara Johnson-Smythe");
mgr.raiseSalary(10_000.00);
printEmployee(mgr);
```

11. Save the `EmployeeTest` class and test your work.

12. (Optional) Improve the look of the salary print output using the `NumberFormat` class.
- Use the following code to get an instance of a static `java.text.NumberFormat` class that you can use to format the salary to look like a standard U.S. dollar currency. Replace the `emp.getSalary()` with the following:

```
NumberFormat.getCurrencyInstance().format(emp.getSalary())
```

In the lesson on abstract classes, you will see how to use an abstract factory, such as `NumberFormat.getCurrencyInstance()`.
13. (Optional) Add additional business logic (data validation) to your `Employee` class.
- Prevent a negative value for the `raiseSalary` method.
 - Prevent a null or empty value for the `setName` method.

(Optional) Practice 3-2: Adding a Staff to a Manager

Overview

In this practice you modify the `Manager` class to add an array of `Employee` objects (a staff) to the manager class, and create methods to add and remove employees from the `Manager`. Finally, add a method to `Manager` to print the staff names and IDs.

Assumptions

Start with the completed project from Practice 3-1 (Summary or Detailed) or the solution from the `solutions\practice1` directory.

Tasks

1. Add fields to the `Manager` class to keep the employee objects.
 - a. Declare a private field called `staff` that is declared as an array of `Employee` objects .
 - b. You will need to keep track of the number of employees in `staff`, so create a private integer field `employeeCount` to keep a count of the number of employees. Initialize the employee count with 0.
 - c. In the constructor, initialize the `staff` array with a maximum of 20 employees.
2. Add a method called `findEmployee`. This method scans the current staff `Employee` array to see whether there is a match between the any member of staff and the `Employee` passed in.
 - a. Return -1 if there is no match, and the index number of the `Employee` if there is a match.
3. Add a method called `addEmployee`. This method adds the `Employee` passed in as a parameter to the end of the array.
 - a. This method should return a boolean value and take an `Employee` object as a parameter. The method should return true if the employee was successfully added and false if the employee already exists as a member of staff.
 - b. Call the `findEmployee` method to determine whether the `Employee` is a member of staff already. Return false if there is match.
 - c. Add the employee object to the staff array. (Hint: Use the `employeeCount` as the index of the array element to assign the employee parameter to.)
 - d. Increment the `employeeCount` and return true.
4. Add a method called `removeEmployee`. This method is a bit more complicated. When you remove an element from the array, you must shift the other elements of the array so there are no empty elements. The easiest way to do this is to create a new array and assign a copy of each of the staff elements to it except for the match. This effectively removes the match from the array.
 - a. Declare a local boolean variable initialized to false to return as the status for the method.
 - b. Declare a temporary array of `Employee` objects to copy the revised staff array to.
 - c. Declare an integer counter of the number of employees copied to the temporary array.

- d. Use a for loop to go through the staff array and attempt to match the employee ID of each element of the staff array with the employee ID of the Employee passed into the method as a parameter.
 - e. If the employee ID's do not match, copy the employee reference from the staff array to the temporary array from step b and increment the count of employees in the temporary array.
 - f. If there is a match, "skip" this employee by continuing to the next element in the staff array, and set the local `boolean` variable from step a to true.
 - g. If there was a match (the local `boolean` is true), replace the current staff array with the temporary array, and the count of employees with the temporary counter from step c.
 - h. Return the local `boolean` variable.
5. Add a method called `printStaffDetails`. This method prints the name of the manager and then each of the elements of staff in turn.

Practices for Lesson 4: Java Class Design

Chapter 4

NAVEEN UNNIKRIISHNAN (naveenunnikrishnan14@gmail.com)
has a non-transferable license to use this Student Guide.

Practices for Lesson 4

Practices Overview

In these practices, you will override methods, including the `toString` method in the `Object` class. You will also create a method in a class that uses the `instanceof` operator to determine which object was passed to the method.

Practice 4-1: Summary Level: Overriding Methods and Applying Polymorphism

Overview

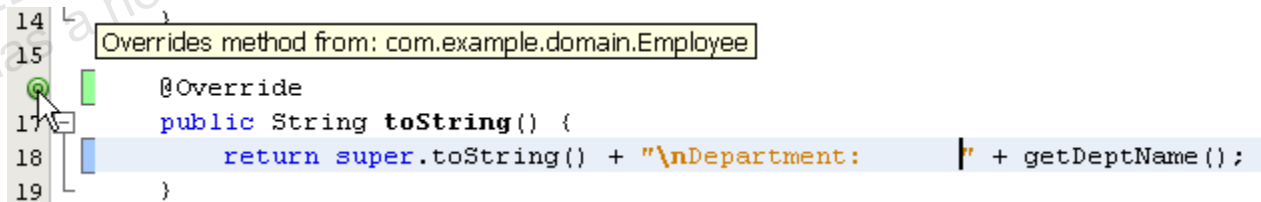
In this practice, you will override the `toString` method of the `Object` class in the `Employee` class and in the `Manager` class. You will create an `EmployeeStockPlan` class with a `grantStock` method that uses the `instanceof` operator to determine how much stock to grant based on the employee type.

Assumptions

Tasks

1. Open the `EmployeePractice` project in the `practices` directory.
2. Edit the `Employee` class to override the `toString()` method from the `Object` class. `Object`'s `toString` method returns a `String`.
 - a. Add a `return` statement that returns a string that includes the employee ID, name, Social Security number, and a salary as a formatted string, with each line separated with a newline character (`"\n"`).
 - b. To format the double salary, use the following:
`NumberFormat.getCurrencyInstance().format(getSalary())`
 - c. Fix any missing import statements.
 - d. Save the class.
3. Override the `toString()` method in the `Manager` class to include the `deptName` field value. Separate this string from the `Employee` string with a newline character.

Note the Green circle icon with the "o" in the center beside the method signature in the `Manager` class. This indicates that NetBeans is aware that this method overrides the method from the parent class, `Employee`. Hold the cursor over the icon to read what this icon represents:



Click the icon, and NetBeans will open the `Employee` class and position the view to the `toString()` method.

4. (Optional) Override the `toString()` method in the `Director` class as well, to display all the fields of a `Director` and the available budget.

5. Create a new class called `EmployeeStockPlan` in the package `com.example.business`. This class will include a single method, `grantStock`, which takes an `Employee` object as a parameter and returns an integer number of stock options based on the employee type:

Employee Type	Number of Stock Options
Director	1000
Manager	100
All other Employees	10

- Add a `grantStock` method that takes an `Employee` object reference as a parameter and returns an integer
 - In the method body, determine what employee type was passed in using the `instanceof` keyword and return the appropriate number of stock options based on that type.
 - Resolve any missing import statements.
 - Save the `EmployeeStockPlan` class.
6. Modify the `EmployeeTest` class. Replace the four print statements in the `printEmployee` method with a single print statement that uses the `toString` method that you created.
7. Overload the `printEmployee` method to take a second parameter, `EmployeeStockPlan`, and print out the number of stock options that this employee will receive.
- Above the `printEmployee` method calls in the main method, create an instance of the `EmployeeStockPlan` and pass that instance to each of the `printEmployee` methods.
 - The new `printEmployee` method should call the first `printEmployee` method and the number of stocks granted to this employee:

```
printEmployee (emp);
System.out.println("Stock Options:  " + esp.grantStock(emp));
```

8. Save the `EmployeeTest` class and run the application. You should see output for each employee that includes the number of Stock Options, such as:

```
Employee id:      101
Employee name:    Jane Smith
Employee Soc Sec #: 012-34-5678
Employee salary:  $120,345.27
Stock Options:    10
```

9. It would be nice to know what type of employee each employee is. Add the following to your original `printEmployee` method above the print statement that prints the employee data fields:

```
System.out.println("Employee type:      " +
    emp.getClass().getSimpleName());
```

This will print out the simple name of the class (`Manager`, `Engineer`, etc). The output of the first employee record should now look like this:

Employee type:	Engineer
Employee id:	101
Employee name:	Jane Smith
Employee Soc Sec #:	012-34-5678
Employee salary:	\$120,345.27
Stock Options:	10

Practice 4-1: Detailed Level: Overriding Methods and Applying Polymorphism

Overview

In this practice, you will override the `toString` method of the `Object` class in the `Employee` class and in the `Manager` class. You will create an `EmployeeStockPlan` class with a `grantStock` method that uses the `instanceof` operator to determine how much stock to grant based on the employee type.

Assumptions

Tasks

1. Open the `EmployeePractice` project in the `practices` directory.
 - a. Select `File > Open Project`.
 - b. Browse to `D:\labs\04-Class_Design\practices`.
 - c. Select `EmployeePractice` and click `Open Project`.
2. Edit the `Employee` class to override the `toString()` method from the `Object` class. `Object`'s `toString` method returns a `String`.
 - a. Add the `toString` method to the `Employee` class with the following signature:


```
public String toString() {
```
 - b. Add a `return` statement that returns a string that includes the employee information: ID, name, Social Security number, and a formatted salary like this:


```
return "Employee ID:      " + getEmpId() + "\n" +
        "Employee Name:    " + getName() + "\n" +
        "Employee SSN:      " + getSsn() + "\n" +
        "Employee Salary: " +
        NumberFormat.getCurrencyInstance().format(getSalary());
```
 - c. Save the `Employee` class.
3. Override the `toString` method in the `Manager` class to include the `deptName` field value.
 - a. Open the `Manager` class.
 - b. Add a `toString` method with the same signature as the `Employee toString` method:

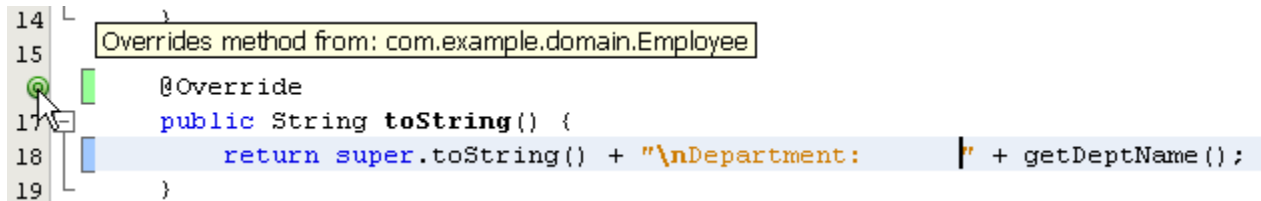

```
public String toString() {
```

The `toString` method in the `Manager` class overrides the `toString` method inherited from the `Employee` class.

- c. Call the parent class method by using the `super` keyword and add the department name:

```
return super.toString() + "\nDepartment: " + getDeptName();
```

Note the Green circle icon with the “o” in the center beside the method signature in the `Manager` class. This indicates that NetBeans is aware that this method overrides the method from the parent class, `Employee`. Hold the cursor over the icon to read what this icon represents:



Click the icon, and NetBeans will open the `Employee` class and position the view to the `toString()` method.

- d. Save the `Manager` class.

4. (Optional) Override the `toString` method in the `Director` class as well, to display all the fields of a director and the available budget.
5. Create a new class called `EmployeeStockPlan` in the package `com.example.business`. This class will include a single method, `grantStock`, which takes an `Employee` object as a parameter and returns an integer number of stock options based on the employee type:

Employee Type	Number of Stock Options
Director	1000
Manager	100
All other Employees	10

- a. Create the new package and class in one step by right-clicking Source Package, and then selecting `New > Java Class`.
- b. Enter `EmployeeStockPlan` as the Class Name and `com.example.business` as the Package and click `Finish`.
- c. In the new class, add fields to the class to define the stock levels, like this:

```
private final int employeeShares = 10;
private final int managerShares = 100;
private final int directorShares = 1000;
```

- d. Add a `grantStock` method that takes an `Employee` object reference as a parameter and returns an integer:

```
public int grantStock(Employee emp) {
```

- e. In the method body, determine what employee type was passed in using the `instanceof` keyword and return the appropriate number of stock options based on that type. Your code might look like this:

```
// Stock is granted based on the employee type
if (emp instanceof Director) {
    return directorShares;
} else {
    if (emp instanceof Manager) {
        return managerShares;
    } else {
        return employeeShares;
    }
}
```

- f. Resolve any missing import statements.
- g. Save the `EmployeeStockPlan` class.
6. Modify the `EmployeeTest` class. Replace the four print statements in the `printEmployee` method with a single print statement that uses the `toString` method that you created.
- a. Replace these lines:

```
System.out.println("Employee id:      " + emp.getEmpId());
System.out.println("Employee name:    " + emp.getName());
System.out.println("Employee Soc Sec #: " + emp.getSsn());
System.out.println("Employee salary:   " +
    NumberFormat.getCurrencyInstance().format((double)
    emp.getSalary()));
```

- b. With one line that uses the `toString()` method:
- ```
System.out.println(emp);
```
7. Overload the `printEmployee` method to take a second parameter, `EmployeeStockPlan`, and print out the number of stock options that this employee will receive.
- a. Create another `printEmployee` method that takes an instance of the `EmployeeStockPlan` class:
- ```
public static void printEmployee(Employee emp, EmployeeStockPlan
    esp) {
```
- b. This method first calls the original `printEmployee` method:
- ```
printEmployee(emp);
```
- c. Add a print statement to print out the number of stock options that the employee is entitled to:
- ```
System.out.println("Stock Options:      " +
    esp.grantStock(emp));
```

- d. Above the `printEmployee` method calls in the main method, create an instance of the `EmployeeStockPlan` and pass that instance to each of the `printEmployee` methods:

```
EmployeeStockPlan esp = new EmployeeStockPlan();
printEmployee(eng, esp);
... modify the remaining printEmployee invocations
```

- e. Resolve any missing import statements.

8. Save the `EmployeeTest` class and run the application. You should see output for each employee that includes the number of Stock Options, such as:

```
Employee id:      101
Employee name:    Jane Smith
Employee Soc Sec #: 012-34-5678
Employee salary:  $120,345.27
Stock Options:    10
```

9. It would be nice to know what type of employee each employee is. Add the following to your original `printEmployee` method above the print statement that prints the employee data fields:

```
System.out.println("Employee type:      " +
    emp.getClass().getSimpleName());
```

This will print out the simple name of the class (Manager, Engineer, etc). The output of the first employee record should now look like this:

```
Employee type:    Engineer
Employee id:      101
Employee name:    Jane Smith
Employee Soc Sec #: 012-34-5678
Employee salary:  $120,345.27
Stock Options:    10
```

NAVEEN UNNIKRIISHNAN (naveenunnikrishnan3.14@gmail.com)
has a non-transferable license to use this Student Guide.

Practices for Lesson 5: Advanced Class Design

Chapter 5

NAVEEN UNNIKRIISHNAN (naveenunnikrishnan14@gmail.com)
has a non-transferable license to use this Student Guide.

Practices for Lesson 5: Overview

Practices Overview

In these practices, you will use the abstract, final, and static Java keywords. You will also learn to recognize nested classes.

Practice 5-1: Summary Level: Applying the Abstract Keyword

Overview

In this practice, you will take an existing application and refactor the code to use the `abstract` keyword.

Assumptions

You have reviewed the abstract class section of this lesson.

Summary

You have been given a project that implements the logic for a bank. The banking software supports only the creation of time deposit accounts. Time deposit accounts allow withdraw only after a maturity date. Time deposit accounts are also known as term deposit, certificate of deposit (CD), or fixed deposit accounts. You will enhance the software to support checking accounts.

A checking account and a time deposit account have some similarities and some differences. Your class design should reflect this. Additional types of accounts might be added in the future.

Tasks

1. Open the `AbstractBanking` project as the main project.
 - a. Select `File > Open Project`.
 - b. Browse to `D:\labs\05-Advanced_Class_Design\practices`.
 - c. Select `AbstractBanking` and select the “Open as Main Project” check box.
 - d. Click the `Open Project` button.
2. Expand the project directories.
3. Run the project. You should see a report of all customers and their accounts.
4. Review the `TimeDepositAccount` class.
 - a. Open the `TimeDepositAccount.java` file (under the `com.example` package).
 - b. Identify the fields and method implementations of `TimeDepositAccount` that are related to time or are in some other way specific to `TimeDepositAccount`. Add a code comment if desired.
 - c. Identify the fields and method implementations of `TimeDepositAccount` that could be used by any type of account. Add a code comment if desired.
5. Create a new Java class, `Account`, in the `com.example` package.
6. Code the `Account` class.
 - a. This class should be declared as `abstract`.
 - b. Move any fields and method implementations from `TimeDepositAccount` that could be used by any type of account to the `Account` class.

Note: The fields and methods should be removed from `TimeDepositAccount`.

- c. Add abstract methods to the `Account` class for any methods in `TimeDepositAccount` that are time related but have a method signature that would make sense in any type of account.
- Hint:** Would all accounts have a description?
- d. Add an `Account` class constructor that has a `double balance` parameter.
- e. The `Account` class should have a protected access level `balance` field that is initialized by the `Account` constructor.
7. Modify the `TimeDepositAccount` class.
- a. `TimeDepositAccount` should be a subclass of `Account`.
- b. Modify the `TimeDepositAccount` constructor to call the parent class constructor with the `balance` value.
- c. Make sure that you are overriding the abstract `withdraw` and `getDescription` methods inherited from the `Account` class.
- Note:** It is a good practice to add `@Override` to any method that should be overriding a parent class method.
8. Modify the `Customer` and `CustomerReport` classes to use `Account` references.
- a. Open the `Customer.java` file (under the `com.example` package).
- b. Change all `TimeDepositAccount` references to `Account` type references.
- c. Open the `CustomerReport.java` file (under the `com.example` package).
- d. Change all `TimeDepositAccount` references to `Account` type references.
9. Run the project. You should see a report of all customers and their accounts.
10. Create a new Java class, `CheckingAccount`, in the `com.example` package.
- a. `CheckingAccount` should be a subclass of `Account`.
- b. Add an `overDraftLimit` field to the `CheckingAccount` class.
- ```
private final double overDraftLimit;
```
- c. Add a `CheckingAccount` constructor that has two parameters.
- `double balance`: Pass this value to the parent class constructor.
  - `double overDraftLimit`: Store this value in the `overDraftLimit` field.
- d. Add a `CheckingAccount` constructor that has one parameter. This constructor should set the `overDraftLimit` field to zero.
- `double balance`: Pass this value to the parent class constructor.
- e.



Override the abstract `getDescription` method inherited from the `Account` class.

```
@Override
public String getDescription() {
 return "Checking Account";
}
```

**Note:** It is a good practice to add `@Override` to any method that should be overriding a parent class method.

- f. Override the abstract `withdraw` method inherited from the `Account` class.
  - The `withdraw` method should allow an account balance to go negative up to the amount specified in the `overDraftLimit` field.
  - The `withdraw` method should return `false` if the withdraw cannot be performed, and `true` if it can.

11. Modify the `AbstractBankingMain` class to create checking accounts for the customers.

```
// Create several customers and their accounts
bank.addCustomer("Jane", "Simms");
customer = bank.getCustomer(0);
customer.addAccount(new TimeDepositAccount(500.00,
cal.getTime()));
customer.addAccount(new CheckingAccount(200.00, 400.00));

bank.addCustomer("Owen", "Bryant");
customer = bank.getCustomer(1);
customer.addAccount(new CheckingAccount(200.00));

bank.addCustomer("Tim", "Soley");
customer = bank.getCustomer(2);
customer.addAccount(new TimeDepositAccount(1500.00,
cal.getTime()));
customer.addAccount(new CheckingAccount(200.00));

bank.addCustomer("Maria", "Soley");
customer = bank.getCustomer(3);
// Maria and Tim have a shared checking account
customer.addAccount(bank.getCustomer(2).getAccount(1));
customer.addAccount(new TimeDepositAccount(150.00,
cal.getTime()));
```

**Note:** Both `Customer` and `CustomerReport` can utilize `CheckingAccount` instances, because you previously modified them to use `Account` type references.

12. Run the project. You should see a report of all customers and their accounts. Note that the date displayed should be one hundred and eighty days in the future.

```
CUSTOMERS REPORT
=====

Customer: Simms, Jane
 Time Deposit Account Sat Feb 04 11:14:54 CST 2012: current
balance is 500.0
 Checking Account: current balance is 200.0

Customer: Bryant, Owen
 Checking Account: current balance is 200.0

Customer: Soley, Tim
 Time Deposit Account Sat Feb 04 11:14:54 CST 2012: current
balance is 1500.0
 Checking Account: current balance is 200.0

Customer: Soley, Maria
 Checking Account: current balance is 200.0
 Time Deposit Account Sat Feb 04 11:14:54 CST 2012: current
balance is 150.0
```

## Practice 5-1: Detailed Level: Applying the Abstract Keyword

### Overview

In this practice, you will take an existing application and refactor the code to use the `abstract` keyword.

### Assumptions

You have reviewed the abstract class section of this lesson.

### Summary

You have been given a project that implements the logic for a bank. The banking software supports only the creation of time deposit accounts. Time deposit accounts allow withdraw only after a maturity date. Time deposit accounts are also known as term deposit, certificate of deposit (CD), or fixed deposit accounts. You will enhance the software to support checking accounts.

A checking account and a time deposit account have some similarities and some differences. Your class design should reflect this. Additional types of accounts might be added in the future.

### Tasks

1. Open the `AbstractBanking` project as the main project.
  - a. Select `File > Open Project`.
  - b. Browse to `D:\labs\05-Advanced_Class_Design\practices`.
  - c. Select `AbstractBanking` and select the “Open as Main Project” check box.
  - d. Click the `Open Project` button.
2. Expand the project directories.
3. Run the project. You should see a report of all customers and their accounts.

```

CUSTOMERS REPORT
=====

Customer: Simms, Jane
 Time Deposit Account Fri Mar 09 12:04:28 CST 2012: current
balance is 500.0

Customer: Bryant, Owen

Customer: Soley, Tim
 Time Deposit Account Fri Mar 09 12:04:28 CST 2012: current
balance is 1500.0

Customer: Soley, Maria
 Time Deposit Account Fri Mar 09 12:04:28 CST 2012: current
balance is 150.0

```

4. Review the `TimeDepositAccount` class.

- a. Open the `TimeDepositAccount.java` file (under the `com.example` package).
- b. Identify the fields and method implementations of `TimeDepositAccount` that are related to time or are in some other way specific to `TimeDepositAccount`. Add a code comment to the `maturityDate` field and the `withdraw` and `getDescription` methods. For example:

```
// time deposit account specific code
private final Date maturityDate;
```

- c. Identify the fields and method implementations of `TimeDepositAccount` that could be used by any type of account. Add a code comment to the `balance` field and the `getBalance`, `deposit`, and `toString` methods. For example:

```
// generic account code
private double balance;
```

5. Create a new Java class, `Account`, in the `com.example` package.6. Code the `Account` class.

- a. This class should be declared as abstract.

```
public abstract class Account
```

- b. Move the `balance` field and the `getBalance`, `deposit`, and `toString` methods from `TimeDepositAccount` to the `Account` class.

**Note:** The fields and methods should be removed from `TimeDepositAccount`.

- c. Add an abstract `getDescription` method to the `Account` class.

```
public abstract String getDescription();
```

- d. Add an abstract `withdraw` method to the `Account` class.

```
public abstract boolean withdraw(double amount);
```

- e. The `Account` class should have a protected access level `balance` field. If you have already moved this field from the `TimeDepositAccount`, just change the access level.

```
protected double balance;
```

- f. Add an `Account` class constructor that has a double `balance` parameter.

```
public Account(double balance) {
 this.balance = balance;
}
```

7. Modify the `TimeDepositAccount` class.

- a. `TimeDepositAccount` should be a subclass of `Account`.

```
public class TimeDepositAccount extends Account
```

- b. Modify the `TimeDepositAccount` constructor to call the parent class constructor with the balance value.

```
super(balance);
```

- c. Make sure that you are overriding the abstract `withdraw` and `getDescription` methods inherited from the `Account` class, by using the `@Override` annotation.

```
@Override
public String getDescription() {
 return "Time Deposit Account " + maturityDate;
}
```

**Note:** It is a good practice to add `@Override` to any method that should be overriding a parent class method.

8. Modify the `Customer` and `CustomerReport` classes to use `Account` references.
  - a. Open the `Customer.java` file (under the `com.example` package).
  - b. Change all `TimeDepositAccount` references to `Account` type references.
  - c. Open the `CustomerReport.java` file (under the `com.example` package).
  - d. Change all `TimeDepositAccount` references to `Account` type references.
9. Run the project. You should see a report of all customers and their accounts.
10. Create a new Java class, `CheckingAccount`, in the `com.example` package.

- a. `CheckingAccount` should be a subclass of `Account`.

```
public class CheckingAccount extends Account
```

- b. Add an `overDraftLimit` field to the `CheckingAccount` class.

```
private final double overDraftLimit;
```

- c. Add a `CheckingAccount` constructor.

```
public CheckingAccount(double balance, double overDraftLimit) {
 super(balance);
 this.overDraftLimit = overDraftLimit;
}
```

- d. Add a `CheckingAccount` constructor that has one parameter.

```
public CheckingAccount(double balance) {
 this(balance, 0);
}
```

- e. Override the abstract `getDescription` method inherited from the `Account` class.

```
@Override
public String getDescription() {
 return "Checking Account";
}
```

**Note:** It is a good practice to add `@Override` to any method that should be overriding a parent class method.

- f. Override the abstract `withdraw` method inherited from the `Account` class. The `withdraw` method should allow an account balance to go negative up to the amount specified in the `overDraftLimit` field.

```
@Override
public boolean withdraw(double amount) {
 if (amount <= balance + overDraftLimit) {
 balance -= amount;
 return true;
 } else {
 return false;
 }
}
```

11. Modify the `AbstractBankingMain` class to create checking accounts for the customers.

```
// Create several customers and their accounts
bank.addCustomer("Jane", "Simms");
customer = bank.getCustomer(0);
customer.addAccount(new TimeDepositAccount(500.00,
cal.getTime()));
customer.addAccount(new CheckingAccount(200.00, 400.00));

bank.addCustomer("Owen", "Bryant");
customer = bank.getCustomer(1);
customer.addAccount(new CheckingAccount(200.00));

bank.addCustomer("Tim", "Soley");
customer = bank.getCustomer(2);
customer.addAccount(new TimeDepositAccount(1500.00,
cal.getTime()));
customer.addAccount(new CheckingAccount(200.00));

bank.addCustomer("Maria", "Soley");
customer = bank.getCustomer(3);
// Maria and Tim have a shared checking account
customer.addAccount(bank.getCustomer(2).getAccount(1));
customer.addAccount(new TimeDepositAccount(150.00,
cal.getTime()));
```

**Note:** Both `Customer` and `CustomerReport` can utilize `CheckingAccount` instances, because you previously modified them to use `Account` type references.

12. Run the project. You should see a report of all customers and their accounts. Note that the date displayed should be one hundred and eighty days in the future.

```
CUSTOMERS REPORT
=====

Customer: Simms, Jane
 Time Deposit Account Sat Feb 04 11:14:54 CST 2012: current
balance is 500.0
 Checking Account: current balance is 200.0

Customer: Bryant, Owen
 Checking Account: current balance is 200.0

Customer: Soley, Tim
 Time Deposit Account Sat Feb 04 11:14:54 CST 2012: current
balance is 1500.0
 Checking Account: current balance is 200.0

Customer: Soley, Maria
 Checking Account: current balance is 200.0
 Time Deposit Account Sat Feb 04 11:14:54 CST 2012: current
balance is 150.0
```

## Practice 5-2: Summary Level: Applying the Singleton Design Pattern

### Overview

In this practice, you will take an existing application and refactor the code to implement the Singleton design pattern.

### Assumptions

You have reviewed the static and final keyword sections of this lesson.

### Summary

You have been given a project that implements the logic for a bank. The application currently allows the creation of an unlimited number of Bank instances.

```
Bank bank = new Bank();
Bank bank2 = new Bank();
Bank bank3 = new Bank();
```

Using the static and final keywords you will limit the number of Bank instances to one per Java virtual machine (JVM).

### Tasks

1. Open the SingletonBanking project as the main project.
  - a. Select File > Open Project.
  - b. Browse to D:\labs\05-Advanced\_Class\_Design\practices.
  - c. Select SingletonBanking and select the "Open as Main Project" check box.
  - d. Click the Open Project button.
2. Expand the project directories.
3. Run the project. You should see a report of all customers and their accounts.

```
CUSTOMERS REPORT
=====

Customer: Simms, Jane
 Time Deposit Account Fri Mar 09 12:04:28 CST 2012: current
balance is 500.0

Customer: Bryant, Owen

Customer: Soley, Tim
 Time Deposit Account Fri Mar 09 12:04:28 CST 2012: current
balance is 1500.0

Customer: Soley, Maria
 Time Deposit Account Fri Mar 09 12:04:28 CST 2012: current
balance is 150.0
```



4. Modify the `Bank` class to implement the Singleton design pattern.
  - a. Open the `Bank.java` file (under the `com.example` package).
  - b. Change the constructor's access level to `private`.
  - c. Add a new field named `instance`. The field should be:
    - `private`
    - Marked `static`
    - Marked `final`
    - Type of `Bank`
    - Initialized to a new `Bank` instance
  - d. Create a static method named `getInstance` that returns the value stored in the `instance` field.
5. Modify the `SingletonBankingMain` class to use the `Bank` singleton.
  - a. Open the `SingletonBankingMain.java` file (under the `com.example` package).
  - b. Replace any calls to the `Bank` constructor with calls to the previously created `getInstance` method.
  - c. In the `main` method, create a second local `Bank` reference named `bank2` and initialize it using the `getInstance` method.
  - d. Use reference equality checking to determine whether `bank` and `bank2` reference the same object.

```
if(bank == bank2) {
 System.out.println("bank and bank2 are the same object");
}
```

- e. Try initializing only the second `Bank` but running the report on the first `Bank`.

```
initializeCustomers(bank2);

// run the customer report
CustomerReport report = new CustomerReport();
report.setBank(bank);
report.generateReport();
```

6. Run the project. You should see a report of all customers and their accounts.

## Practice 5-2: Detailed Level: Applying the Singleton Design Pattern

---

### Overview

In this practice, you will take an existing application and refactor the code to implement the Singleton design pattern.

### Assumptions

You have reviewed the static and final keyword sections of this lesson.

### Summary

You have been given a project that implements the logic for a bank. The application currently allows the creation of an unlimited number of `Bank` instances.

```
Bank bank = new Bank();
Bank bank2 = new Bank();
Bank bank3 = new Bank();
```

Using the static and final keywords you will limit the number of `Bank` instances to one per Java Virtual Machine (JVM).

### Tasks

1. Open the `SingletonBanking` project as the main project.
  - a. Select `File > Open Project`.
  - b. Browse to `D:\labs\05-Advanced_Class_Design\practices`.
  - c. Select `SingletonBanking` and select the “Open as Main Project” check box.
  - d. Click the `Open Project` button.
2. Expand the project directories.
3. Run the project. You should see a report of all customers and their accounts.
4. Modify the `Bank` class to implement the Singleton design pattern.
  - a. Open the `Bank.java` file (under the `com.example` package).
  - b. Change the constructor's access level to `private`.

```
private Bank() {
 customers = new Customer[10];
 numberOfCustomers = 0;
}
```

c.

Add a new field named `instance`. The field should be:

- `private`
- Marked `static`
- Marked `final`
- Type of `Bank`
- Initialized to a new `Bank` instance

```
private static final Bank instance = new Bank();
```

- d. Create a static method named `getInstance` that returns the value stored in the `instance` field.

```
public static Bank getInstance() {
 return instance;
}
```

5. Modify the `SingletonBankingMain` class to use the `Bank` singleton.

- a. Open the `SingletonBankingMain.java` file (under the `com.example` package).
- b. Replace any calls to the `Bank` constructor with calls to the previously created `getInstance` method.

```
Bank bank = Bank.getInstance();
```

- c. In the `main` method, create a second local `Bank` reference named `bank2` and initialize it using the `getInstance` method.

```
Bank bank2 = Bank.getInstance();
```

- d. Use reference equality checking to determine whether `bank` and `bank2` reference the same object.

```
if (bank == bank2) {
 System.out.println("bank and bank2 are the same object");
}
```

- e. Initialize only the second `Bank`, but run the report on the first `Bank`.

```
initializeCustomers(bank2);

// run the customer report
CustomerReport report = new CustomerReport();
report.setBank(bank);
report.generateReport();
```

6. Run the project. You should see a report of all customers and their accounts.

## (Optional) Practice 5-3: Using Java Enumerations

### Overview

In this practice, you will take an existing application and refactor the code to use an enum.

### Assumptions

You have reviewed the enum section of this lesson.

### Summary

You have been given a project that implements the logic for a bank. The application currently allows the creation of `TimeDepositAccount` instances with any maturity date.

```
//180 day term
Calendar cal = Calendar.getInstance();
cal.add(Calendar.DAY_OF_YEAR, 180);
new TimeDepositAccount(500.00, cal.getTime())
```

By creating a new Java enum you will modify the application to only allow for the creation of `TimeDepositAccount` instances with a maturity date that is 90 or 180 in the future.

### Tasks

1. Open the `EnumBanking` project as the main project.
  - a. Select `File > Open Project`.
  - b. Browse to `D:\labs\05-Advanced_Class_Design\practices`.
  - c. Select `EnumBanking` and select the “Open as Main Project” check box.
  - d. Click the `Open Project` button.
2. Expand the project directories.
3. Run the project. You should see a report of all customers and their accounts.
4. Create a new Java enum, `DepositLength`, in the `com.example` package.
5. Code the `DepositLength` enum.
  - a. Declare a `days` field along with a corresponding constructor and getter method.

```
private int days;

private DepositLength(int days) {
 this.days = days;
}

public int getDays() {
 return days;
}
```

- b. Create two `DepositLength` instances, `THREE_MONTHS` and `SIX_MONTHS` that call the `DepositLength` constructor with values of 90 and 180 respectively.

6. Modify the `TimeDepositAccount` class to only accept `DepositLength` instances for the constructor's maturity date parameter.
  - a. Open the `TimeDepositAccount.java` file (under the `com.example` package).
  - b. Modify the existing constructor to receive an enum as the second parameter.

```
public TimeDepositAccount(double balance, DepositLength
duration) {
 super(balance);
 Calendar cal = Calendar.getInstance();
 cal.add(Calendar.DAY_OF_YEAR, duration.getDays());
 this.maturityDate = cal.getTime();
}
```

7. Modify the `EnumBankingMain` class to create `TimeDepositAccount` instances using the two `DepositLength` instances available.
  - a. Open the `EnumBankingMain.java` file (under the `com.example` package).
  - b. Within the `initializeCustomers` method, remove the code to create calendars.
  - c. Within the `initializeCustomers` method, modify the creation of all `TimeDepositAccount` instances to use the `DepositLength` enum.

```
customer.addAccount(new TimeDepositAccount(500.00,
DepositLength.SIX_MONTHS));
```

**Note:** Try using both the `SIX_MONTHS` and `THREE_MONTHS` values. You can also use a static import to reduce the length of the statement.

8. Run the project. You should see a report of all customers and their accounts. It is now impossible to compile a line of code that creates a `TimeDepositAccount` with an invalid maturity date.

## (Optional) Practice 5-4: Recognizing Nested Classes

---

### Overview

In this practice, you will take an existing application and attempt to recognize the declaration and use of various types of nested classes.

### Assumptions

You have reviewed the nested class section of this lesson.

### Summary

You have been given a small project that contains only two `.java` files. Although there are only two `.java` files, there may be multiple Java classes being created.

Attempt to determine the number of classes being created.

### Tasks

1. Open the `NestedClasses` project as the main project.
  - a. Select `File > Open Project`.
  - b. Browse to `D:\labs\05-Advanced_Class_Design\practices\`.
  - c. Select `NestedClasses` and select the “Open as Main Project” check box.
  - d. Click the Open Project button.
2. Expand the project directories.
3. Run the project. You should see the output in the output window.
4. Count the number of classes created in the `OuterClass.java` file.
  - a. Open the `OuterClass.java` file (under the `com.example` package).
  - b. Determine the total number of classes created in this file.
  - c. Determine the total number of top-level classes created in this file.
  - d. Determine the total number of nested classes created in this file.
  - e. Determine the total number of inner classes.
  - f. Determine the total number of member classes.
  - g. Determine the total number of local classes.
  - h. Determine the total number of anonymous classes.
  - i. Determine the total number of static nested classes.

**Hint:** Using the Files tab in NetBeans, you can see how many `.class` files are created by looking in the `build\classes` folder for a project.

## (Optional) Solution 5-4: Recognizing Nested Classes

---

### Overview

In this solution, you will take an existing application and review the number and types of nested classes created within a single `.java` file.

### Assumptions

You have reviewed the nested class section of this lesson.

### Summary

You have been given a small project that contains only two `.java` files. Although there are only two `.java` files, there may be multiple Java classes being created.

Review the number of classes being created.

### Tasks

1. Open the `NestedClasses` project as the main project.
  - a. Select `File > Open Project`.
  - b. Browse to `D:\labs\05-Advanced_Class_Design\practices`.
  - c. Select `NestedClasses` and select the “Open as Main Project” check box.
  - d. Click the Open Project button.
2. Expand the project directories.
3. Run the project. You should see the output in the output window.
4. Open the `OuterClass.java` file (under the `com.example` package).
  - Within the `OuterClass.java` file there are:
    - 10 classes
      - 1 top-level class
      - 9 nested classes
        - 8 inner classes
          - 3 member classes
          - 2 local classes
          - 3 anonymous classes
        - 1 static nested class

- Classes are declared on the following lines within the `OuterClasss.java` file:
  - line 3: top-level class
  - line 10: local inner class
  - line 22: anonymous local inner class
  - line 32: anonymous inner class
  - line 40: anonymous inner class
  - line 48: member inner class
  - line 62: static nested class
  - line 72: member inner class
  - line 74: member inner class
  - line 77: local inner class

**Hint:** You can show line number in NetBeans by going to the View menu and enabling Show Line Numbers.



## **Practices for Lesson 6: Inheritance with Java Interfaces**

### **Chapter 6**

## Practices for Lesson 6: Overview

---

### Practices Overview

In these practices, you will use Java interfaces and apply design patterns.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

NAVEEN UNNIKRIISHNAN (naveenunnikrishnan3.14@gmail.com)  
has a non-transferable license to use this Student Guide.

## Practice 6-1: Summary Level: Implementing an Interface

### Overview

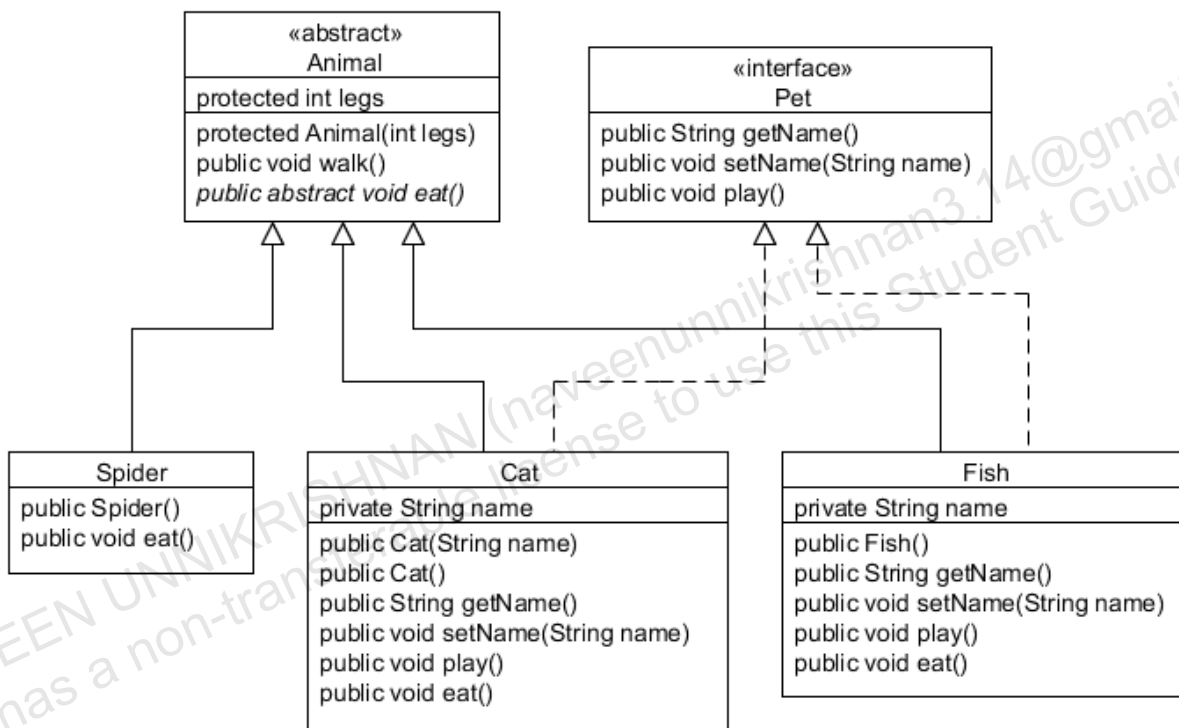
In this practice, you will create an interface and implement that interface.

### Assumptions

You have reviewed the interface section of this lesson.

### Summary

You have been given a project that contains an abstract class named `Animal`. You create a hierarchy of animals that is rooted in the `Animal` class. Several of the animal classes implement an interface named `Pet`, which you will create.



### Tasks

- Open the `Pet` project as the main project.
  - Select File > Open Project.
  - Browse to `D:\labs\06-Interfaces\practices`.
  - Select `Pet` and select the "Open as Main Project" check box.
  - Click the Open Project button.
- Expand the project directories.
- Run the project. You should see text displayed in the output window.

4. Review the `Animal` and `Spider` classes.
  - a. Open the `Animal.java` file (under the `com.example` package).
  - b. Review the abstract `Animal` class. You will extend this class.
  - c. Open the `Spider.java` file (under the `com.example` package).
  - d. The `Spider` class is an example of extending the `Animal` class.
5. Create a new Java interface: `Pet` in the `com.example` package.
6. Code the `Pet` interface. This interface should include three method signatures:
  - `public String getName();`
  - `public void setName(String name);`
  - `public void play();`
7. Create a new Java class: `Fish` in the `com.example` package.
8. Code the `Fish` class.
  - a. This class should:
    - Extend the `Animal` class
    - Implement the `Pet` interface
  - b. Complete this class by creating:
    - A `String` field called `name`
    - Getter and setter methods for the `name` field
    - A no-argument constructor that passes a value of 0 to the parent constructor
    - A `play()` method that prints out "Just keep swimming."
    - An `eat()` method that prints out "Fish eat pond scum."
    - A `walk()` method that overrides the `Animal` class `walk` method. It should first call the super class `walk` method, and then print "Fish, of course, can't walk; they swim."
9. Create a new Java class: `Cat` in the `com.example` package.
10. Code the `Cat` class.
  - a. This class should:
    - Extend the `Animal` class
    - Implement the `Pet` interface
  - b. Complete this class by creating:
    - A `String` field called `name`
    - Getter and setter methods for the `name` field
    - A constructor that receives a `name String` and passes a value of 4 to the parent constructor
    - A no-argument constructor that passes a value of "Fluffy" to the other constructor in this class
    - A `play()` method that prints out `name + " likes to play with string."`
    - An `eat()` method that prints out "Cats like to eat spiders and fish."

11. Modify the `PetMain` class.

- a. Open the `PetMain.java` file (under the `com.example` package).
- b. Review the main method. You should see the following lines of code:

```
Animal a;
//test a spider with a spider reference
Spider s = new Spider();
s.eat();
s.walk();
//test a spider with an animal reference
a = new Spider();
a.eat();
a.walk();
```

- c. Add additional lines of code to test the `Fish` and `Cat` classes that you created.
  - Try using every constructor
  - Experiment with using every reference type possible and determine which methods can be called with each type of reference. Use a `Pet` reference while testing the `Fish` and `Cat` classes.
- d. Implement and test the `playWithAnimal(Animal a)` method.
  - Determine whether the argument implements the `Pet` interface. If so, cast the reference to a `Pet` and invoke the `play` method. If not, print a message of "Danger! Wild Animal".
  - Call the `playWithAnimal(Animal a)` method from within `main`, passing in each type of animal.

## 12. Run the project. You should see text displayed in the output window.

## Practice 6-1: Detailed Level: Implementing an Interface

### Overview

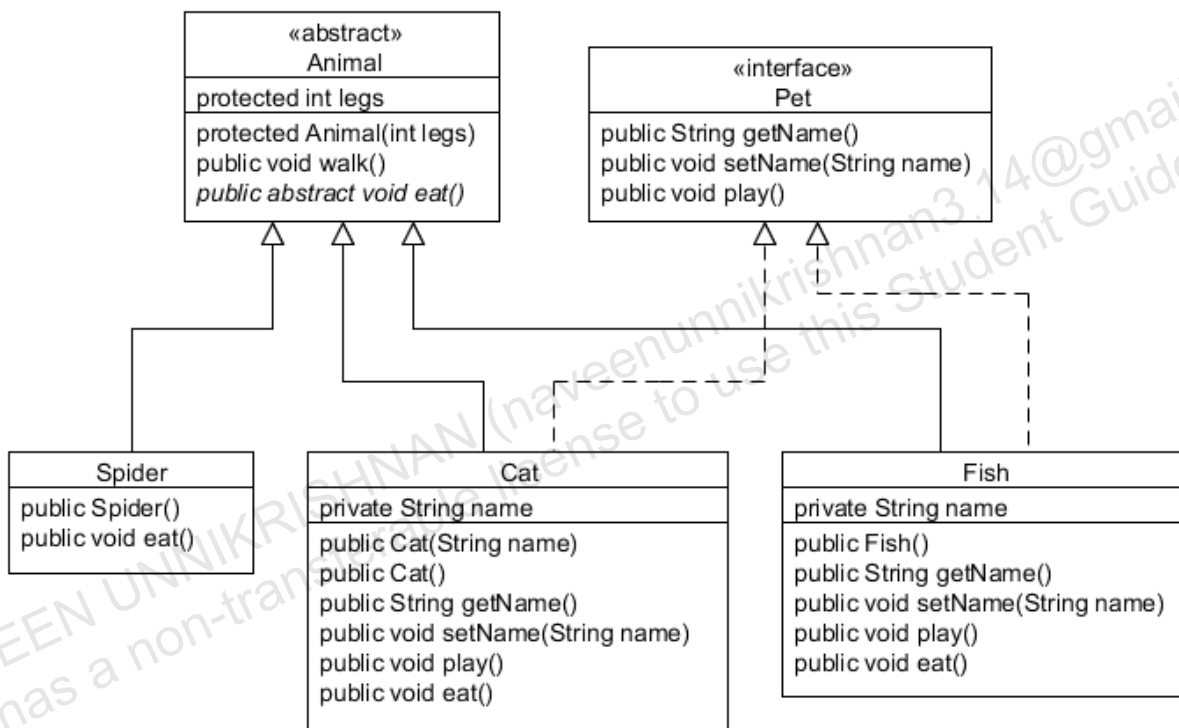
In this practice, you will create an interface and implement that interface.

### Assumptions

You have reviewed the interface section of this lesson.

### Summary

You have been given a project that contains an abstract class named `Animal`. You create a hierarchy of animals that is rooted in the `Animal` class. Several of the animal classes implement an interface named `Pet`, which you will create.



### Tasks

- Open the `Pet` project as the main project.
  - Select `File > Open Project`.
  - Browse to `D:\labs\06-Interfaces\practices`.
  - Select `Pet` and select the "Open as Main Project" check box.
  - Click the `Open Project` button.
- Expand the project directories.
- Run the project. You should see text displayed in the output window.

4. Review the `Animal` and `Spider` classes.
  - a. Open the `Animal.java` file (under the `com.example` package).
  - b. Review the abstract `Animal` class. You will extend this class.
  - c. Open the `Spider.java` file (under the `com.example` package).
  - d. The `Spider` class is an example of extending the `Animal` class.
5. Create a new Java interface: `Pet` in the `com.example` package.
6. Code the `Pet` interface. This interface should include three method signatures:

```
public String getName();
public void setName(String name);
public void play();
```

7. Create a new Java class: `Fish` in the `com.example` package.
8. Code the `Fish` class.
  - a. This class should extend the `Animal` class and implement the `Pet` interface.

```
public class Fish extends Animal implements Pet
```

- b. Complete this class by creating:

- A `String` field called `name`.

```
private String name;
```

- Getter and setter methods for the `name` field.

```
@Override
public String getName() {
 return name;
}

@Override
public void setName(String name) {
 this.name = name;
}
```

- A no-argument constructor that passes a value of 0 to the parent constructor.

```
public Fish() {
 super(0);
}
```

- A `play()` method that prints out "Just keep swimming."

```
@Override
public void play() {
 System.out.println("Just keep swimming.");
}
```

- An `eat()` method that prints out "Fish eat pond scum."

```
@Override
public void eat() {
 System.out.println("Fish eat pond scum.");
}
```

- A `walk()` method that overrides the `Animal` class `walk` method. It should first call the super class `walk` method, and then print "Fish, of course, can't walk; they swim."

```
@Override
public void walk() {
 super.walk();
 System.out.println("Fish, of course, can't walk; they swim.");
}
```

9. Create a new Java class: `Cat` in the `com.example` package.

10. Code the `Cat` class.

- a. This class should extend the `Animal` class and implement the `Pet` interface.

```
public class Cat extends Animal implements Pet
```

- b. Complete this class by creating:

- A `String` field called `name`.
- Getter and setter methods for the `name` field.
- A constructor that receives a `name` `String` and passes a value of 4 to the parent constructor.

```
public Cat(String name) {
 super(4);
 this.name = name;
}
```

- A no-argument constructor that passes a value of "Fluffy" to the other constructor in this class.

```
public Cat() {
 this("Fluffy");
}
```

- A `play()` method that prints out `name + " likes to play with string."`

```
@Override
public void play() {
 System.out.println(name + " likes to play with string.");
}
```

- An `eat()` method that prints out "Cats like to eat spiders and fish."



## 11. Modify the PetMain class.

- a. Open the PetMain.java file (under the com.example package).
- b. Review the main method. You should see the following lines of code:

```
Animal a;
//test a spider with a spider reference
Spider s = new Spider();
s.eat();
s.walk();
//test a spider with an animal reference
a = new Spider();
a.eat();
a.walk();
```

- c. Add additional lines of code to test the Fish and Cat classes that you created.

- Try using every constructor
- Experiment with using every reference type possible and determine which methods can be called with each type of reference. Use a Pet reference while testing the Fish and Cat classes.

```
Pet p;

Cat c = new Cat("Tom");
c.eat();
c.walk();
c.play();
a = new Cat();
a.eat();
a.walk();
p = new Cat();
p.setName("Mr. Whiskers");
p.play();

Fish f = new Fish();
f.setName("Guppy");
f.eat();
f.walk();
f.play();
a = new Fish();
a.eat();
a.walk();
```

- d.

Implement and test the `playWithAnimal(Animal a)` method.

- Determine whether the argument implements the `Pet` interface. If so, cast the reference to a `Pet` and invoke the `play` method. If not, print a message of "Danger! Wild Animal".

```
public static void playWithAnimal(Animal a) {
 if(a instanceof Pet) {
 Pet p = (Pet)a;
 p.play();
 } else {
 System.out.println("Danger! Wild Animal");
 }
}
```

- Call the `playWithAnimal(Animal a)` method at the end of the `main` method, passing in each type of animal.

```
playWithAnimal(s);
playWithAnimal(c);
playWithAnimal(f);
```

12. Run the project. You should see text displayed in the output window.

## Practice 6-2: Summary Level: Applying the DAO Pattern

### Overview

In this practice, you will take an existing application and refactor the code to implement the data access object (DAO) design pattern.

### Assumptions

You have reviewed the DAO sections of this lesson.

### Summary

You have been given a project that implements the logic for a human resources application. The application allows for creating, retrieving, updating, deleting, and listing `Employee` objects.

`Employee` objects are currently stored in-memory using an array. You must move any code related to the persistence of `Employee` objects out of the `Employee` class. In later practices, you will supply alternative persistence implementations. In the future, this application should require no modification when substituting the persistence implementation.

### Tasks

1. Open the `EmployeeMemoryDAO` project as the main project.
  - a. Select `File > Open Project`.
  - b. Browse to `D:\labs\06-Interfaces\practices`.
  - c. Select `EmployeeMemoryDAO` and select the "Open as Main Project" check box.
  - d. Click the Open Project button.
2. Expand the project directories.
3. Run the project. You should see a menu. Test all the menu choices.

```
[C]reate | [R]ead | [U]pdate | [D]elete | [L]ist | [Q]uit:
```

**Note:** When entering dates, they should be in the form of: Nov 26, 1976

Employee IDs should be in the range of 0 through 9.

4. Review the `Employee` class.
  - a. Open the `Employee.java` file (under the `com.example.model` package).
  - b. Find the array used to store employees.

```
private static Employee[] employeeArray = new Employee[10];
```

**Note:** The employee's `id` is used as the array index.

- c. Locate any methods that utilize the `employeeArray` field. These methods are used to persist employee objects.
5. Create a new `com.example.dao` package.
6. Create an `EmployeeDAO` interface in the `com.example.dao` package.

7. Complete the `EmployeeDAO` interface with the following method signatures.

```
public void add(Employee emp);
public void update(Employee emp);
public void delete(int id);
public Employee findById(int id);
public Employee[] getAllEmployees();
```

8. Create an `EmployeeDAOMemoryImpl` class in the `com.example.dao` package.
9. Complete the `EmployeeDAOMemoryImpl` class.
- Move the `employeeArray` and any related methods from the `Employee` class to the `EmployeeDAOMemoryImpl` class.
  - Implement the `EmployeeDAO` interface. Modify the methods that you moved in the previous step to become the methods required by the `EmployeeDAO` interface.

**Hint:** In this DAO, the add and update methods will function the same.

10. Update the `EmployeeTestInteractive` class.
- The `EmployeeTestInteractive` class no longer compiles, review the errors.
  - Create an `EmployeeDAO` instance in the main method. Use the `EmployeeDAO` interface as the reference type.
  - Modify any lines containing errors to use the `EmployeeDAO` instance.
11. Run the project. You should see a menu. Test all the menu choices.

**Note:** While functional, the `EmployeeTestInteractive` class is still tied to a specific type of DAO because it references the `EmployeeDAO` implementing class by name.

```
EmployeeDAO dao = new EmployeeDAOMemoryImpl();
```

In the following steps, you remove this tight coupling from the `EmployeeTestInteractive` class by creating a DAO factory.

12. Modify the `EmployeeDAOMemoryImpl` interface.
- Add a protected, no-argument constructor.
13. Create an `EmployeeDAOFactory` class in the `com.example.dao` package.
14. Complete the `EmployeeDAOFactory` class.
- Add a method that returns an `EmployeeDAO`.

```
public EmployeeDAO createEmployeeDAO() {
 return new EmployeeDAOMemoryImpl();
}
```

15. Update the `EmployeeTestInteractive` class to use the `EmployeeDAOFactory` class.
- Obtain an `EmployeeDAOFactory` instance in the main method.
  - Obtain an `EmployeeDAO` instance using the factory created in the previous step.
16. Run the project. You should see a menu. Test all the menu choices.

In the future, you will be able to change the persistence mechanism to use a database without changing the reference types or method calls used in the `EmployeeTestInteractive` class.

## Practice 6-2: Detailed Level: Applying the DAO Pattern

### Overview

In this practice, you will take an existing application and refactor the code to implement the data access object (DAO) design pattern.

### Assumptions

You have reviewed the DAO sections of this lesson.

### Summary

You have been given a project that implements the logic for a human resources application. The application allows for creating, retrieving, updating, deleting, and listing `Employee` objects.

`Employee` objects are currently stored in-memory using an array. You must move any code related to the persistence of `Employee` objects out of the `Employee` class. In later practices, you will supply alternative persistence implementations. In the future, this application should require no modification when substituting the persistence implementation.

### Tasks

1. Open the `EmployeeMemoryDAO` project as the main project.
  - a. Select `File > Open Project`.
  - b. Browse to `D:\labs\06-Interfaces\practices`.
  - c. Select `EmployeeMemoryDAO` and select the "Open as Main Project" check box.
  - d. Click the Open Project button.
2. Expand the project directories.
3. Run the project. You should see a menu. Test all the menu choices.

```
[C]reate | [R]ead | [U]pdate | [D]elete | [L]ist | [Q]uit:
```

**Note:** When entering dates, they should be in the form of: Nov 26, 1976

Employee IDs should be in the range of 0 through 9.

4. Review the `Employee` class.
  - a. Open the `Employee.java` file (under the `com.example.model` package).
  - b. Find the array used to store employees. You will relocate this field in a subsequent step.

```
private static Employee[] employeeArray = new Employee[10];
```

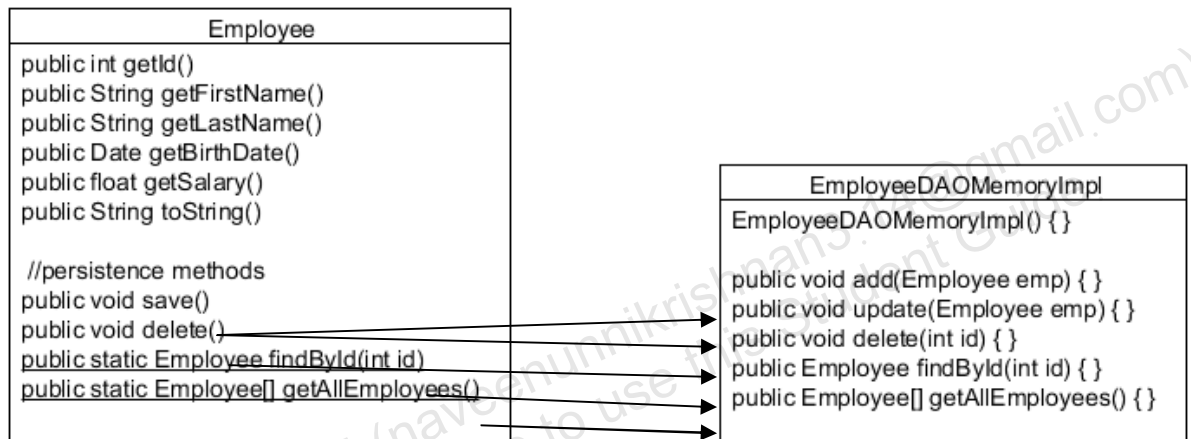
**Note:** The employee's `id` is used as the array index.

- c. Locate the `save`, `delete`, `findById`, and `getAllEmployees` methods that utilize the `employeeArray` field. These methods are used to persist employee objects. You will relocate these methods in a subsequent step.
5. Create a new `com.example.dao` package.
6. Create an `EmployeeDAO` interface in the `com.example.dao` package.

7. Complete the `EmployeeDAO` interface with the following method signatures. Add import statements as needed.

```
public void add(Employee emp);
public void update(Employee emp);
public void delete(int id);
public Employee findById(int id);
public Employee[] getAllEmployees();
```

8. Create an `EmployeeDAOMemoryImpl` class in the `com.example.dao` package.
9. Complete the `EmployeeDAOMemoryImpl` class.
- a. Move the `employeeArray` and any related methods from the `Employee` class to the `EmployeeDAOMemoryImpl` class:



- b. Implement the `EmployeeDAO` interface. Modify the methods that you moved in the previous step to become the methods required by the `EmployeeDAO` interface.
- The `save` method becomes the `add` method and is modified to have an `Employee` parameter.
  - The `save` method is duplicated to become the `update` method and is modified to have an `Employee` parameter.
  - The `delete` method is modified to have an `id` parameter.
  - The `findById` method is no longer `static`.
  - The `getAllEmployees` method is no longer `static`.

```

public class EmployeeDAOMemoryImpl implements EmployeeDAO {
 private static Employee[] employeeArray = new Employee[10];

 public void add(Employee emp) {
 employeeArray[emp.getId()] = emp;
 }

 public void update(Employee emp) {
 employeeArray[emp.getId()] = emp;
 }

 public void delete(int id) {
 employeeArray[id] = null;
 }

 public Employee findById(int id) {
 return employeeArray[id];
 }

 public Employee[] getAllEmployees() {
 List<Employee> emps = new ArrayList<>();
 for (Employee e : employeeArray) {
 if (e != null) {
 emps.add(e);
 }
 }
 return emps.toArray(new Employee[0]);
 }
}

```

10. Update the EmployeeTestInteractive class.

- a. The EmployeeTestInteractive class no longer compiles, review the errors.
- b. Create an EmployeeDAO instance in the main method. Use the EmployeeDAO interface as the reference type. Replace the line:

```
//TODO create dao
```

With:

```
EmployeeDAO dao = new EmployeeDAOMemoryImpl();
```

- c.

Modify any lines containing errors to use the `EmployeeDAO` instance. For example:

```
case 'C':
 emp = inputEmployee(in);
 dao.add(emp);
 System.out.println("Successfully added Employee Record: " +
emp.getId());
 System.out.println("\n\nCreated " + emp);
 break;
```

**Note:** You can also remove the now unnecessary finding of employee object that is present when deleting an employee.

```
// Find this Employee record
emp = null;
emp = Employee.findById(id);
if (emp == null) {
 System.out.println("\n\nEmployee " + id + " not found");
 break;
}
```

11. Run the project. You should see a menu. Test all the menu choices.

**Note:** While functional, the `EmployeeTestInteractive` class is still tied to a specific type of DAO because it references the `EmployeeDAO` implementing class by name.

```
EmployeeDAO dao = new EmployeeDAOMemoryImpl();
```

In the following steps, you remove this tight coupling from the `EmployeeTestInteractive` class by creating a DAO factory.

12. Modify the `EmployeeDAOMemoryImpl` interface.

- a. Add a protected, no-argument constructor.

```
EmployeeDAOMemoryImpl() {
}
```

13. Create an `EmployeeDAOFactory` class in the `com.example.dao` package.

14. Complete the `EmployeeDAOFactory` class.

- Add a method that returns an `EmployeeDAO`.

```
public class EmployeeDAOFactory {

 public EmployeeDAO createEmployeeDAO() {
 return new EmployeeDAOMemoryImpl();
 }
}
```



15. Update the `EmployeeTestInteractive` class to use the `EmployeeDAOFactory` class.

- a. Obtain an `EmployeeDAOFactory` instance in the `main` method. Replace the line:

```
//TODO create factory
```

With:

```
EmployeeDAOFactory factory = new EmployeeDAOFactory();
```

- b. Obtain an `EmployeeDAO` instance using the factory created in the previous step. Replace the line:

```
EmployeeDAO dao = new EmployeeDAOMemoryImpl();
```

With:

```
EmployeeDAO dao = factory.createEmployeeDAO();
```

- c. Fix any imports.

16. Run the project. You should see a menu. Test all the menu choices.

In the future, you will be able to change the persistence mechanism to use a database without changing the reference types or method calls used in the `EmployeeTestInteractive` class. Notice that none of the `*MemoryImpl` classes are used by name from within the `EmployeeTestInteractive` class.

## (Optional) Practice 6-3: Implementing Composition

---

### Overview

In this practice, you will take an existing application and refactor it to make use of composition.

### Assumptions

You have reviewed the interface and composition sections of this lesson.

### Summary

You have been given a small project that represents a hierarchy of animals that is rooted in the `Animal` class. Several of the animal classes implement an interface named `Pet`. This project is a completed implementation of the “Implementing an Interface” practice.

There are some potential problems with the design of the existing project. If you wanted to restrict a pet’s name to less than 20 characters how many classes would you have to modify? Would this problem become worse with the addition of new animals?

If some types of animals, such as `Fish`, cannot walk, should they have a walk method?

### Tasks

1. Open the `PetComposition` project as the main project.
  - a. Select `File > Open Project`.
  - b. Browse to `D:\labs\06-Interfaces\practices`.
  - c. Select `PetComposition` and select the “Open as Main Project” check box.
  - d. Click the `Open Project` button.
2. Expand the project directories.
3. Run the project. You should see output in the output window.
4. Centralize all “name” functionality.

All pets can be named, but you may want to give names to objects that cannot play. For instance, you could name a volleyball “Wilson.” Your design should reflect this.

- a. Create a `Nameable` interface (under the `com.example` package).
- b. Complete the `Nameable` interface with `setName` and `getName` method signatures.

```
public interface Nameable {

 public void setName(String name);

 public String getName();

}
```

- c. Create a `NameableImpl` class (under the `com.example` package).
- d.

Complete the `NameableImpl` class. It should:

- Implement the `Nameable` interface
  - Contain a private `String` field called `name`
  - Only accept names less than 20 characters in length
  - Print "Name too long" if a name is too long
- e. Modify the `Pet` interface.
- Extend the `Nameable` interface.
  - Remove the `getName` and `setName` method signatures (they are inherited now).
- f. Modify the `Fish` and `Cat` classes to use composition.
- Delete the `name` field.
  - Delete the existing `getName` and `setName` methods.
  - Add a new `Nameable` field.

```
private Nameable nameable = new NameableImpl();
```

- Add `getName` and `setName` methods that delegate to the `Nameable` field.

**Hint:** Position the cursor within the curly braces for the class. Open the Source menu, select Insert Code, select Delegate Method, select the `Nameable` check box, and click the Generate button.

- Replace any use of the old `name` field with calls to the `getName` and `setName` methods.

5. Centralize all walking functionality.

Only some animal can walk. Remove the `walk` method from the `Animal` class and use interfaces and composition to facilitate walking.

- a. Create an `Ambulatory` interface (under the `com.example` package).
- b. Complete the `Ambulatory` interface with the `walk` method signature.

```
public interface Ambulatory {
 public void walk();
}
```

- c. Create an `AmbulatoryImpl` class (under the `com.example` package).
- d. Complete the `AmbulatoryImpl` class. It should:
  - Implement the `Ambulatory` interface
  - Contain a private `int` field called `legs`
  - Contain a single argument constructor that receives an `int` value to be stored in the `legs` field
  - Contain a `walk` method

```
public void walk() {
 System.out.println("This animal walks on " + legs + "
legs.");
}
```

- e. Delete the `walk` method from the `Fish` class.
- f. Modify the `Spider` and `Cat` classes to use composition.
  - Add a new `Ambulatory` field.

```
private Ambulatory ambulatory;
```

- Add a `walk` method that delegates to the `Ambulatory` field.

**Hint:** Position the cursor within the curly braces for the class. Open the Source menu, select Insert Code, select Delegate Method, select the `Ambulatory` check box, and click the Generate button.

- g. Initialize the `ambulatory` field within the `Spider` and `Cat` constructors. For example:

```
public Spider() {
 ambulatory = new AmbulatoryImpl(8);
}
```

- 6. Modify the `PetMain` class to test the `walk` method. The `walk` method can only be called on `Spider`, `Cat`, or `Ambulatory` references.

## **Practices for Lesson 7: Generics and Collections**

### **Chapter 7**

NAVEEN UNNIKRIISHNAN (naveenunnikrishnan14@gmail.com)  
has a non-transferable license to use this Student Guide.

## Practices for Lesson 7: Overview

---

### Practices Overview

In these practices, use generics and collections to practice the concepts covered in the lecture. For each practice, a NetBeans project is provided for you. Complete the project as indicated in the instructions.

## Practice 7-1: Summary Level: Counting Part Numbers by Using HashMaps

### Overview

In this practice, use the HashMap collection to count a list of part numbers.

### Assumptions

You have reviewed the collections section of this lesson.

### Summary

You have been asked to create a simple program to count a list of part numbers that are of an arbitrary length. Given the following mapping of part numbers to descriptions, count the number of each part. Produce a report that shows the count of each part sorted by the part's product description. The part-number-to-description mapping is as follows:

| Part Number | Description      |
|-------------|------------------|
| 1S01        | Blue Polo Shirt  |
| 1S02        | Black Polo Shirt |
| 1H01        | Red Ball Cap     |
| 1M02        | Duke Mug         |

Once complete, your report should look like this:

```

=== Product Report ===
Name: Black Polo Shirt Count: 6
Name: Blue Polo Shirt Count: 7
Name: Duke Mug Count: 3
Name: Red Ball Cap Count: 5

```

### Tasks

Open the Generics-Practice01 project and make the following changes.

1. For the `ProductCounter` class, add two private map fields. The first map counts part numbers. The order of the keys does not matter. The second map stores the mapping of product description to part number. The keys should be sorted alphabetically by description for the second map.
2. Create a one argument constructor that accepts a `Map` as a parameter. The map that stores the description-to-part-number mapping should be passed in here.
3. Create a `processList()` method to process a list of `String` part numbers. Use a `HashMap` to store the current count based on the part number.

```
public void processList(String[] list) { }
```

4. Create a `printReport()` method to print out the results.

```
public void printReport() { }
```

5. Add code to the `main` method to print out the results.
6. Run the `ProductCounter.java` class to ensure that your program produces the desired output.



## Practice 7-1: Detailed Level: Counting Part Numbers by Using HashMaps

### Overview

In this practice, use the HashMap collection to count a list of part numbers.

### Assumptions

You have reviewed the collections section of this lesson.

### Summary

You have been asked to create a simple program to count a list of part numbers that are of an arbitrary length. Given the following mapping of part numbers to descriptions, count the number of each part. Produce a report that shows the count of each part sorted by the part's product description. The part number to description mapping is as follows:

| Part Number | Description      |
|-------------|------------------|
| 1S01        | Blue Polo Shirt  |
| 1S02        | Black Polo Shirt |
| 1H01        | Red Ball Cap     |
| 1M02        | Duke Mug         |

Once complete, your report should look like this:

```

=== Product Report ===
Name: Black Polo Shirt Count: 6
Name: Blue Polo Shirt Count: 7
Name: Duke Mug Count: 3
Name: Red Ball Cap Count: 5

```

### Tasks

Open the Generics-Practice01 project and make the following changes.

- For the ProductCounter class, add two private map fields. The first map counts part numbers. The order of the keys does not matter. The second map stores the mapping of product description to part number. The keys should be sorted alphabetically by description for the second map.

```

private Map<String, Long> productCountMap = new HashMap<>();
private Map<String, String> productNames = new TreeMap<>();

```

2. Create a one argument constructor that accepts a `Map` as a parameter.

```
public ProductCounter(Map productNames){
 this.productNames = productNames;
}
```

3. Create a `processList()` method to process a list of `String` part numbers. Use a `HashMap` to store the current count based on the part number.

```
public void processList(String[] list){
 long curVal = 0;
 for(String itemNumber:list){
 if (productCountMap.containsKey(itemNumber)){
 curVal = productCountMap.get(itemNumber);
 curVal++;
 productCountMap.put(itemNumber, new
Long(curVal));
 } else {
 productCountMap.put(itemNumber, new Long(1));
 }
 }
}
```

4. Create a `printReport()` method to print out the results.

```
public void printReport(){
 System.out.println("=== Product Report ===");
 for (String key:productNames.keySet()){
 System.out.print("Name: " + key);
 System.out.println("\t\tCount: " +
productCountMap.get(productNames.get(key)));
 }
}
```

5. Add the following code to the `main` method to print out the results.

```
ProductCounter pc1 = new ProductCounter (productNames);
pc1.processList(parts);
pc1.printReport();
```

6. Run the `ProductCounter.java` class to ensure that your program produces the desired output.

## Practice 7-2: Summary Level: Matching Parentheses by Using a Deque

---

### Overview

In this practice, you use the `Deque` object to match parentheses in a programming statement.

### Assumptions

You have reviewed the collections section of this lesson.

### Summary

Use the `Deque` data structure as a stack to match parentheses in a programming statement. You will be given several sample lines containing logical statements. Test the lines to ensure that the parentheses match, return `true` if they do, `false` if they do not.

For example, the output from the program might look like the following.

```
Line 0 is valid
Line 1 is invalid
Line 2 is invalid
Line 3 is valid
```

### Tasks

Open the `Generics-Practice02` project and make the following changes.

1. Modify the `processLine()` method in `ParenMatcher.java` to read a line in and convert the string into a character array.
2. Loop through the array. Push "(" onto the stack. When a ")" is encountered, pop a "(" from the stack. Two conditions should return `false`.
  - a. If you need to call a pop operation and the stack is empty, the number of parentheses do not match, return `false`.
  - b. If after completing the loop a "(" is left on the stack return `false`. The number of parentheses does not match.
3. Run the `ParanMatcher.java` class to ensure that your program produces the desired output.

## Practice 7-2: Detailed Level: Matching Parentheses by Using a Deque

---

### Overview

In this practice, you use the `Deque` object to match parentheses in a programming statement.

### Assumptions

You have reviewed the collections section of this lesson.

### Summary

Use the `Deque` data structure as a stack to match parentheses in a programming statement. You will be given several sample lines containing logical statements. Test the lines to ensure that the parentheses match, return `true` if they do, `false` if they do not.

For example, the output from the program might look like the following.

```
Line 0 is valid
Line 1 is invalid
Line 2 is invalid
Line 3 is valid
```

### Tasks

Open the `Generics-Practice02` project and make the following changes.

1. Modify the `processLine()` method in `ParenMatcher.java` to read a line in and convert the string into an array of characters. Clear the stack and convert the line to a character array.

```
stack.clear();
curLine = line.toCharArray();
```

2. In the same method, loop through the array. Push "(" onto the stack. When a ")" is encountered, pop a "(" from the stack. If "(" is left on the stack, or you attempt to perform a pop operation on an empty stack, the parentheses do not match, return false. Otherwise, return true. To do this, add the following code to the processLine method (replace the return true; statement).

```

for (char c:curLine){
 switch (c){
 case '(':stack.push(c);break;
 case ')':{
 if (stack.size() > 0){
 stack.pop();
 } else {
 return false;
 }
 break;
 }
 }
}
if (stack.size()> 0){
 return false; // Missing match invalid expression
} else {
 return true; //
}

```

3. Run the ParanMatcher.java class to ensure that your program produces the desired output.

## Practice 7-3: Summary Level: Counting Inventory and Sorting by Using Comparators

---

### Overview

In this practice, you process shirt-related transactions for a Duke's Choice store. Compute the inventory level for a number of shirts. Then print out the shirt data sorted by description and by inventory count.

### Assumptions

You have reviewed all the content in this lesson.

### Summary

Any Duke's Choice stores carry a number of products including shirts. In this practice, process the shirt-related transactions and calculate the inventory levels. After the levels have been calculated, print a report sorted by description and a report sorted by inventory count. You will create two classes that implement the Comparator interface to allow sorting shirts by count and by description.

For example, the output from the program might look like the following.

```
=== Inventory Report - Description ===
```

```
Shirt ID: P002
```

```
Description: Black Polo Shirt
```

```
Color: Black
```

```
Size: M
```

```
Inventory: 15
```

```
Shirt ID: P001
```

```
Description: Blue Polo Shirt
```

```
Color: Blue
```

```
Size: L
```

```
Inventory: 24
```

```
Shirt ID: P003
```

```
Description: Maroon Polo Shirt
```

```
Color: Maroon
```

```
Size: XL
```

```
Inventory: 20
```

Shirt ID: P004  
 Description: Tan Polo Shirt  
 Color: Tan  
 Size: S  
 Inventory: 19

=== Inventory Report - Count ===  
 Shirt ID: P002  
 Description: Black Polo Shirt  
 Color: Black  
 Size: M  
 Inventory: 15

Shirt ID: P004  
 Description: Tan Polo Shirt  
 Color: Tan  
 Size: S  
 Inventory: 19

Shirt ID: P003  
 Description: Maroon Polo Shirt  
 Color: Maroon  
 Size: XL  
 Inventory: 20

Shirt ID: P001  
 Description: Blue Polo Shirt  
 Color: Blue  
 Size: L  
 Inventory: 24

## Tasks

Open the Generics-Practice03 project and make the following changes.

1. Review the `Shirt` class and `InventoryCount` interface to see how the `Shirt` class has changed to support inventory features.
2. Review the `DukeTransaction` class to see how transactions are defined for this program.
3. Update the `SortShirtByCount` Comparator class to sort shirts by count.
4. Update the `SortShirtByDesc` Comparator class to sort shirts by description.

5. Update the `TestItemCounter` class to process the shirts and transactions and produce the desired report.
  - Loop through the transactions and update the appropriate shirt object contained in the `polos` map.
  - Each `Shirt` class implements the `InventoryCount` interface. Use those methods and the `count` field to increment and decrement the inventory levels.
  - Print the list of shirts by description.
  - Print the list of shirts by count
6. Run the `TestItemCounter.java` class to ensure that your program produces the desired output.



## Practice 7-3: Detailed Level: Counting Inventory and Sorting by Using Comparators

---

### Overview

In this practice, you process shirt-related transactions for a Duke's Choice store. Compute the inventory level for a number of shirts. Then print out the shirt data sorted by description and by inventory count.

### Assumptions

You have reviewed all the content in this lesson.

### Summary

Any Duke's Choice stores carry a number of products including shirts. In this practice, process the shirt-related transactions and calculate the inventory levels. Once the levels have been calculated print a report sorted by description and a report sorted by inventory count. You will create two classes that implement the Comparator interface to allow sorting shirts by count and by description.

For example, the output from the program might look like the following.

```
=== Inventory Report - Description ===
```

```
Shirt ID: P002
```

```
Description: Black Polo Shirt
```

```
Color: Black
```

```
Size: M
```

```
Inventory: 15
```

```
Shirt ID: P001
```

```
Description: Blue Polo Shirt
```

```
Color: Blue
```

```
Size: L
```

```
Inventory: 24
```

```
Shirt ID: P003
```

```
Description: Maroon Polo Shirt
```

```
Color: Maroon
```

```
Size: XL
```

```
Inventory: 20
```

Shirt ID: P004  
Description: Tan Polo Shirt  
Color: Tan  
Size: S  
Inventory: 19

=== Inventory Report - Count ===

Shirt ID: P002  
Description: Black Polo Shirt  
Color: Black  
Size: M  
Inventory: 15

Shirt ID: P004  
Description: Tan Polo Shirt  
Color: Tan  
Size: S  
Inventory: 19

Shirt ID: P003  
Description: Maroon Polo Shirt  
Color: Maroon  
Size: XL  
Inventory: 20

Shirt ID: P001  
Description: Blue Polo Shirt  
Color: Blue  
Size: L  
Inventory: 24

## Tasks

Open the Generics-Practice03 project and make the following changes.

1. Review the `Shirt` class and `InventoryCount` interface to see how the `Shirt` class has changed to support inventory features.
2. Review the `DukeTransaction` class to see how transactions are defined for this program.

3. Update the `SortShirtByCount` Comparator class to sort shirts by count.

```
public class SortShirtByCount implements Comparator<Shirt>{
 public int compare(Shirt s1, Shirt s2){
 Long c1 = new Long(s1.getCount());
 Long c2 = new Long(s2.getCount());

 return c1.compareTo(c2);
 }
}
```

4. Update the `SortShirtByDesc` Comparator class to sort shirts by description.

```
public class SortShirtByDesc implements Comparator<Shirt>{
 public int compare(Shirt s1, Shirt s2){
 return
s1.getDescription().compareTo(s2.getDescription());
 }
}
```

5. Update the `TestItemCounter` class to process the shirts and transactions and produce the desired report.

- Loop through the transactions and update the appropriate shirt object contained in the `polos` Map. This will produce an inventory count for each product.

```
// Count the shirts
for (DukeTransaction transaction:transactions){
 if (polos.containsKey(transaction.getProductID())){
 currentShirt = polos.get(transaction.getProductID());
 } else {
 System.out.println("Error: Invalid part number");
 }

 switch (transaction.getTransactionType()) {
 case "Purchase":currentShirt.
addItem(transaction.getCount()); break;

 case "Sale":currentShirt.
removeItems(transaction.getCount()); break;

 default: System.out.println("Error: Invalid Transaction
Type"); continue;
 }
}
```

- Print the list of shirts by description.

```
// Convert to List
List<Shirt> poloList = new ArrayList<>(polos.values());

// Init Comparators
Comparator sortDescription = new SortShirtByDesc();
Comparator sortCount = new SortShirtByCount();

// Print Results - Sort by Description
Collections.sort(poloList, sortDescription);
System.out.println("=== Inventory Report - Description ===");

for(Shirt shirt:poloList){
 System.out.println(shirt.toString());
}
```

- Print the list of shirts by count.

```
// Print Results - Sort by Count
Collections.sort(poloList, sortCount);
System.out.println("=== Inventory Report - Count ===");

for(Shirt shirt:poloList){
 System.out.println(shirt.toString());
}
```

6. Run the `TestItemCounter.java` class to ensure that your program produces the desired output.

## **Practices for Lesson 8: String Processing**

### **Chapter 8**

NAVEEN UNNIKRIISHNAN (naveenunnikrishnan14@gmail.com)  
has a non-transferable license to use this Student Guide.

## Practices for Lesson 8: Overview

---

### Practices Overview

In these practices, you use regular expressions and `String.split()` to manipulate strings in Java. For each practice, a NetBeans project is provided for you. Complete the project as indicated in the instructions.

## Practice 8-1: Summary Level: Parsing Text with `split()`

---

### Overview

In this practice, parse comma-delimited text and convert the data into Shirt objects.

### Assumptions

You have participated in the lecture for this lesson.

### Summary

You have been given some comma-delimited shirt data. Parse the data, store it in shirt objects, and print the results. The output from the program should look like the following.

```

=== Shirt List ===
Shirt ID: S001
Description: Black Polo Shirt
Color: Black
Size: XL

Shirt ID: S002
Description: Black Polo Shirt
Color: Black
Size: L

Shirt ID: S003
Description: Blue Polo Shirt
Color: Blue
Size: XL

Shirt ID: S004
Description: Blue Polo Shirt
Color: Blue
Size: M

Shirt ID: S005
Description: Tan Polo Shirt
Color: Tan
Size: XL

Shirt ID: S006
Description: Black T-Shirt
Color: Black
Size: XL

```

Shirt ID: S007  
Description: White T-Shirt  
Color: White  
Size: XL

Shirt ID: S008  
Description: White T-Shirt  
Color: White  
Size: L

Shirt ID: S009  
Description: Green T-Shirt  
Color: Green  
Size: S

Shirt ID: S010  
Description: Orange T-Shirt  
Color: Orange  
Size: S

Shirt ID: S011  
Description: Maroon Polo Shirt  
Color: Maroon  
Size: S

## Tasks

Open the `StringsPractice01` project and make the following changes.

1. Edit the main method of the `StringSplitTest.java` file.
2. Parse each line of the `shirts` array.
3. Convert the shirt data into a `List` of `Shirt` objects.
4. Print out the list of shirts.
5. Run the `StringSplitTest.java` file and verify that your output is similar to that shown in the “Summary” section of this practice.



## Practice 8-1: Detailed Level: Parsing Text with `split()`

---

### Overview

In this practice, parse comma-delimited text and convert the data into Shirt objects.

### Assumptions

You have participated in the lecture for this lesson.

### Summary

You have been given some comma-delimited shirt data. Parse the data, store it in shirt objects, and print the results. The output from the program should look like the following.

```
=== Shirt List ===
Shirt ID: S001
Description: Black Polo Shirt
Color: Black
Size: XL

Shirt ID: S002
Description: Black Polo Shirt
Color: Black
Size: L

Shirt ID: S003
Description: Blue Polo Shirt
Color: Blue
Size: XL

Shirt ID: S004
Description: Blue Polo Shirt
Color: Blue
Size: M

Shirt ID: S005
Description: Tan Polo Shirt
Color: Tan
Size: XL

Shirt ID: S006
Description: Black T-Shirt
Color: Black
Size: XL
```

Shirt ID: S007  
Description: White T-Shirt  
Color: White  
Size: XL

Shirt ID: S008  
Description: White T-Shirt  
Color: White  
Size: L

Shirt ID: S009  
Description: Green T-Shirt  
Color: Green  
Size: S

Shirt ID: S010  
Description: Orange T-Shirt  
Color: Orange  
Size: S

Shirt ID: S011  
Description: Maroon Polo Shirt  
Color: Maroon  
Size: S

## Tasks

Open the StringsPractice01 project and make the following changes.

1. Edit the main method of the StringSplitTest.java file.
2. Parse each line of the shirts array. Create a Shirt object for each line and add the Shirt to a List. A for loop to perform these steps could be as follows:

```
for(String curLine:shirts){
 String[] e = curLine.split(",");
 shirtList.add(new Shirt(e[0], e[1], e[2], e[3]));
}
```

3. Print out the list of shirts. A loop to do this could be like the following:

```
System.out.println("=== Shirt List ===");
for (Shirt shirt:shirtList){
 System.out.println(shirt.toString());
}
```

4. Run the `StringSplitTest.java` file and verify that your output is similar to that shown in the “Summary” section of this practice.

## Practice 8-2: Summary Level: Creating a Regular Expression Search Program

---

### Overview

In this practice, create a program that searches a text file by using regular expressions.

### Assumptions

You have participated in the lecture for this lesson.

### Summary

Create a simple application that will loop through a text file (`gettys.html`) and search for text by using regular expressions. If the desired text is found on a line, print out the line number and the line text. For example, if you performed a search for “<h4>” the output would be:

```
9 <h4>Abraham Lincoln</h4>
10 <h4>Thursday, November 19, 1863</h4>
```

### Tasks

Open the `StringsPractice02` project and make the following changes. Please note that the code to read a file has been supplied for you.

**Note:** The `gettys.html` file is located in the root of the project folder. To examine the file, with the project open, click the Files tab. Double-click the file to open it and examine its contents.

1. Edit the `FindText.java` file.
2. Create a `Pattern` and a `Matcher` field.
3. Generate a `Matcher` based on the supplied `Pattern` object.
4. Search each line for the pattern supplied.
5. Print the line number and the line that has matching text.
6. Run the `FindText.java` file and search for these patterns.
  - All lines that contain: `<h4>`
  - All the lines that contain the word “to” (For example, line 17 should not be selected.)
  - All the lines that start with 4 spaces'
  - Lines that begin with “<p” or “<d”
  - Lines that only contain HTML closing tags (for example, “</div>”)
7. (Optional) Modify the program to accept the file name and regular expression pattern on the command line.

## Practice 8-2: Detailed Level: Creating a Regular Expression Search Program

### Overview

In this practice, create a program that searches a text file by using regular expressions.

### Assumptions

You have participated in the lecture for this lesson.

### Summary

Create a simple application that will loop through a text file and search for text by using regular expressions. If the desired text is found on a line, print out the line number and the line text. For example, if you performed a search for "<h4>" the output would be:

```
9 <h4>Abraham Lincoln</h4>
10 <h4>Thursday, November 19, 1863</h4>
```

### Tasks

Open the `StringsPractice02` project and make the following changes. Please note that the code to read a file has been supplied for you.

**Note:** The `gettys.html` file is located in the root of the project folder. To examine the file, with the project open, click the Files tab. Double-click the file to open it and examine its contents.

1. Edit the `FindText.java` file.
2. Create fields for a `Pattern` and a `Matcher` object.

```
private Pattern pattern;
private Matcher m;
```

3. Outside the search loop, create and initialize your pattern object.

```
pattern = Pattern.compile("<h4>");
```

4. Inside the search loop, generate a `Matcher` based on the supplied `Pattern` object.

```
m = pattern.matcher(line);
```

5. Inside the search loop, search each line for the pattern supplied. Print the line number and the line that has matching text.

```
if (m.find()) {
 System.out.println(" " + c + " " + line);
}
```

6. Run the `FindText.java` file and search for these patterns.

- All the lines that contain: `<h4>`

```
pattern = Pattern.compile("<h4>");
```

- All the lines that contain the word "to" (For example, line 17 should not be selected.)

```
pattern = Pattern.compile("\\bto\\b");
```

- All the lines that start with 4 spaces

```
pattern = Pattern.compile("^\\s{4}");
```

- Lines that begin with "<p" or "<d"

```
pattern = Pattern.compile("<[p|d]");
```

- Lines that only contain HTML closing tags (for example, "</div>")

```
pattern = Pattern.compile("</.?*>$");
```

## 7. (Optional) Modify the program to accept the file name and regular expression pattern on the command line.

## Practice 8-3: Summary Level: Transforming HTML by Using Regular Expressions

### Overview

In this practice, use regular expressions to transform `<p>` tags into `<span>` tags.

### Assumptions

You have participated in the lecture for this lesson.

### Summary

You have decided that you want to change the formatting of the `gettys.html` file. Instead of using `<p>` tags, `<span>` tags should be used. In addition, you think that the value for `class` should be "sentence" instead of "line." Use regular expressions to find the lines that you want to change. Then use regular expressions to transform the tags and the attributes as described. The transformed lines should be output to the console. The output should look like the following:

```
13 Four score and seven years ago our
 fathers brought forth on this continent a new nation, conceived
 in liberty, and dedicated to the proposition that all men are
 created equal.
14 Now we are engaged in a great civil
 war, testing whether that nation, or any nation, so conceived
 and so dedicated, can long endure.
15 We are met on a great battle-field of
 that war.
16 We have come to dedicate a portion of
 that field, as a final resting place for those who here gave
 their lives that that nation might live.
17 It is altogether fitting and proper
 that we should do this.
21 But, in a larger sense, we can not
 dedicate, we can not consecrate, we can not hallow this
 ground.
...
```

One approach to the problem could be to break the algorithm into three steps.

1. Break the line into three parts: the start tag, the content, and the end tag.
2. Replace the current tags with a new tag.
3. Replace the attribute value with a new attribute value.

Then return the newly formatted line.

The method signatures to replace the tag and attributes might look like this:

```
public String replaceTag(String tag, String targetTag,
String replaceTag){ }
public String replaceAttribute(String tag, String attribute,
String value){ }
```

## Tasks

Open the `StringsPractice03` project and make the following changes. Please note that the code to read a file has been supplied for you.

1. Edit the `SearchReplace.java` file.
2. Create a `Pattern` object to match the entire line.
3. As you loop through the file, do the following:
  - Create a `Matcher` to match the current line.
  - Execute the `find()` method to find a match.
  - If there is a match, replace the start and end tags.
  - Replace the attribute
4. Create a method that will replace the contents of any tag.
5. Create a method that will replace a tag's attribute.
6. Run the `SearchReplace.java` file and produce the output shown in the "Summary" section of this practice.



## Practice 8-3: Detailed Level: Transforming HTML by Using Regular Expressions

### Overview

In this practice, use regular expressions to transform `<p>` tags into `<span>` tags.

### Assumptions

You have participated in the lecture for this lesson.

### Summary

You have decided that you want to change the formatting of the `gettys.html` file. Instead of using `<p>` tags, `<span>` tags should be used. In addition, you think that the value for `class` should be "sentence" instead of "line." Use regular expressions to find the lines that you want to change. Then use regular expressions to transform the tags and the attributes as described. The transformed lines should be output to the console. The output should look like the following:

```
13 Four score and seven years ago our
 fathers brought forth on this continent a new nation, conceived
 in liberty, and dedicated to the proposition that all men are
 created equal.
14 Now we are engaged in a great civil
 war, testing whether that nation, or any nation, so conceived
 and so dedicated, can long endure.
15 We are met on a great battle-field of
 that war.
16 We have come to dedicate a portion of
 that field, as a final resting place for those who here gave
 their lives that that nation might live.
17 It is altogether fitting and proper
 that we should do this.
21 But, in a larger sense, we can not
 dedicate, we can not consecrate, we can not hallow this
 ground.
...
```

One approach to the problem could be to break the algorithm into three steps.

1. Break the line into three parts: the start tag, the content, and the end tag.
2. Replace the current tags with a new tag.
3. Replace the attribute value with a new attribute value.

Then return the newly formatted line.

The method signatures to replace the tag and attributes might look like this:

```
public String replaceTag(String tag, String targetTag,
String replaceTag){ }
public String replaceAttribute(String tag, String attribute,
String value){ }
```

## Tasks

Open the `StringsPractice03` project and make the following changes. Please note that the code to read a file has been supplied for you.

1. Edit the `SearchReplace.java` file.
2. Create a `Pattern` object to match the entire line.

```
Pattern pattern1 = Pattern.compile("<" + targetTag +
".*?>")(.*)(</" + targetTag + ".*?>");
```

3. As you loop through the file, do the following:

- Create a `Matcher` to match the current line.

```
Matcher m = pattern1.matcher(line);
```

- Execute the `find()` method to find a match. If there is a match, replace the start and end tags. Replace the attribute

```
if (m.find()) {
 String newStart = replaceTag(m.group(1), targetTag,
replaceTag);
 newStart = replaceAttribute(newStart, attribute, value);
 String newEnd = replaceTag(m.group(3), targetTag, replaceTag);

 String newLine = newStart + m.group(2) + newEnd;
 System.out.printf("%3d %s\n", c, newLine);
}
```

4. Create a method that will replace the contents of any tag.

```
public String replaceTag(String tag, String targetTag, String
replaceTag){
 Pattern p = Pattern.compile(targetTag); // targetTag is
regex
 Matcher m = p.matcher(tag); // tag is text to replace
 if (m.find()){
 return m.replaceFirst(replaceTag); // swap target with
replace
 }
 return targetTag;
}
```

5. Create a method that will replace a tag's attribute.

```
public String replaceAttribute(String tag, String attribute,
String value){
 Pattern p = Pattern.compile(attribute + "=" + "\\.*?\\");
 Matcher m = p.matcher(tag); // tag is text to replace
 if (m.find()){
 return m.replaceFirst(attribute + "=" + "\"" + value +
 "\"");
 }
 return tag;
}
```

6. Run the `SearchReplace.java` file and produce the output shown in the "Summary" section of this practice.

NAVEEN UNNIKRIISHNAN (naveenunnikrishnan3.14@gmail.com)  
has a non-transferable license to use this Student Guide.

## **Practices for Lesson 9: Exceptions and Assertions**

### **Chapter 9**

NAVEEN UNNIKRISSHANNAN (naveenunnikrishnan14@gmail.com)  
has a non-transferable license to use this Student Guide.

## Practices for Lesson 9: Overview

---

### Practices Overview

In these practices, you will use `try-catch` statements, extend the `Exception` class, and use the `throw` and `throws` keywords.

## Practice 9-1: Summary Level: Catching Exceptions

### Overview

In this practice, you will create a new project and catch checked and unchecked exceptions.

### Assumptions

You have reviewed the exception handling section of this lesson.

### Summary

You will create a project that reads from a file. The file-reading code will be provided to you. Your task is to add the appropriate exception-handling code.

### Tasks

1. Create a new `ExceptionPractice` project as the main project.
  - a. Select **File > New Project**.
  - b. Select **Java** under **Categories** and **Java Application** under **Projects**. Click the **Next** button.
  - c. Enter the following information in the “Name and Location” dialog box:
    - **Project Name:** `ExceptionPractice`
    - **Project Location:** `D:\labs\09-Exceptions\practices`.
    - **(checked) Create Main Class:** `com.example.ExceptionMain`
    - **(checked) Set as Main Project**
  - d. Click the *Finish* button.
2. Add the following line to the main method.

```
System.out.println("Reading from file:" + args[0]);
```

**Note:** A command-line argument will be used to specify the file that will be read. Currently no arguments will be supplied, do not correct this oversight yet.

3. Run the project. You should see an error message similar to:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
 at com.example.ExceptionMain.main(ExceptionMain.java:7)
Java Result: 1
```

4. Surround the `println` line of code you added with a **try-catch** statement.
  - The catch clause should:
    - Accept a parameter of type `ArrayIndexOutOfBoundsException`
    - Print the message: `"No file specified, quitting!"`
    - Exit the application with an exit status of 1 by using the appropriate static method within the `System` class

**Note:** Because the compiler did not force you to handle or declare the `ArrayIndexOutOfBoundsException`, it is an unchecked exception. Typically, you should not need to use a `try-catch` block to deal with an unchecked exception. Checking the length of the `args` array is an alternate way to ensure that a command-line argument was supplied.

5. Run the project. You should see an error message similar to:

```
No file specified, quitting!
Java Result: 1
```

6. Add a command-line argument to the project.
  - a. Right-click the `ExceptionPractice` project and select Properties.
  - b. In the Project Properties dialog box, select the Run category.
  - c. In the Arguments field, enter a value of:  
`D:\labs\resources\DeclarationOfIndependence.txt`
  - d. Click the OK button.

7. Run the project. You should see a message similar to:

```
Reading from
file:D:\labs\resources\DeclarationOfIndependence.txt
```

**Warning:** Running the project is not the same as running the file. The command-line argument will only be passed to the `main` method if you run the project.

8. Add the following lines of code to the `main` method below your previously added lines:

```
BufferedReader b =
 new BufferedReader(new FileReader(args[0]));
String s = null;
while((s = b.readLine()) != null) {
 System.out.println(s);
}
```

9. Run the Fix Imports wizard by right-clicking in the source-code window.
10. You should now see compiler errors in some of the lines that you just added. These lines potentially generate checked exceptions. By manually building the project or holding your cursor above the line with errors, you should see a message similar to:

```
unreported exception FileNotFoundException; must be caught or
declared to be thrown
```

11. Modify the project properties to support the `try-with-resources` statement.
  - a. Right-click the `ExceptionPractice` project and select Properties.
  - b. In the Project Properties dialog box, select the Sources category.
  - c. In the Source/Binary Format drop-down list, select JDK 7.
  - d. Click the OK button.



12. Surround the file IO code provided in step 8 with a `try-with-resources` statement.
  - The line that creates and initializes the `BufferedReader` should be an automatically closed resource.
  - Add a catch clause for a `FileNotFoundException`. Within the catch clause:
    - Print "File not found:" + `args[0]`
    - Exit the application.
  - Add a catch clause for an `IOException`. Within the catch clause:
    - Print " Error reading file:" along with the message available in the `IOException` object
    - Exit the application.
13. Run the project. You should see the content of the `D:\labs\resources\DeclarationOfIndependence.txt` file displayed in the output window.

## Practice 9-1: Detailed Level: Catching Exceptions

---

### Overview

In this practice, you will create a new project and catch checked and unchecked exceptions.

### Assumptions

You have reviewed the exception handling section of this lesson.

### Summary

You will create a project that reads from a file. The file-reading code will be provided to you. Your task is to add the appropriate exception-handling code.

### Tasks

1. Create a new `ExceptionPractice` project as the main project.
  - a. Select File > New Project.
  - b. Select Java under Categories and Java Application under Projects. Click the Next button.
  - c. Enter the following information in the “Name and Location” dialog box:
    - Project Name: `ExceptionPractice`
    - Project Location: `D:\labs\09-Exceptions\practices`.
    - (checked) Create Main Class: `com.example.ExceptionMain`
    - (checked) Set as Main Project
  - d. Click the Finish button.
2. Add the following line to the `main` method.

```
System.out.println("Reading from file:" + args[0]);
```

**Note:** A command-line argument will be used to specify the file that will be read. Currently no arguments will be supplied; do not correct this oversight yet.

3. Run the project. You should see an error message similar to:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
 at com.example.ExceptionMain.main(ExceptionMain.java:7)
Java Result: 1
```

4. Surround the `println` line of code you added with a `try-catch` statement.

- The catch clause should:
  - Accept a parameter of type `ArrayIndexOutOfBoundsException`
  - Print the message: "No file specified, quitting!"
  - Exit the application with an exit status of 1 by using the `System.exit(1)` method

```
try {
 System.out.println("Reading from file:" + args[0]);
} catch (ArrayIndexOutOfBoundsException e) {
 System.out.println("No file specified, quitting!");
 System.exit(1);
}
```

**Note:** Since the compiler did not force you to handle or declare the `ArrayIndexOutOfBoundsException` it is an unchecked exception. Typically you should not need to use a `try-catch` block to deal with an unchecked exception. Checking the length of the `args` array is an alternate way to ensure that a command line argument was supplied.

5. Run the project. You should see an error message similar to:

```
No file specified, quitting!
Java Result: 1
```

6. Add a command-line argument to the project.
- a. Right-click the `ExceptionPractice` project and click `Properties`.
  - b. In the `Project Properties` dialog box, select the `Run` category.
  - c. In the `Arguments` field, enter a value of:  
`D:\labs\resources\DeclarationOfIndependence.txt`
  - d. Click the `OK` button.

7. Run the project. You should see a message similar to:

```
Reading from
file:D:\labs\resources\DeclarationOfIndependence.txt
```

**Warning:** Running the project is not the same as running the file. The command-line argument will only be passed to the `main` method if you run the project.

8. Add the following lines of code to the `main` method below your previously added lines:

```
BufferedReader b =
 new BufferedReader(new FileReader(args[0]));
String s = null;
while((s = b.readLine()) != null) {
 System.out.println(s);
}
```

9. Run the `Fix Imports` wizard by right-clicking in the source-code window.

10. You should now see compiler errors in some of the lines that you just added. These lines potentially generate checked exceptions. By manually building the project or holding your cursor above the line with errors, you should see a message similar to:

```
unreported exception FileNotFoundException; must be caught or
declared to be thrown
```

11. Modify the project properties to support the `try-with-resources` statement.
- Right-click the `ExceptionPractice` project and select `Properties`.
  - In the `Project Properties` dialog box, select the `Sources` category.
  - In the `Source/Binary Format` drop-down list, select `JDK 7`.
  - Click the `OK` button.
12. Surround the file IO code provided in step 8 with a `try-with-resources` statement.
- The line that creates and initializes the `BufferedReader` should be an automatically closed resource.
  - Add a catch clause for a `FileNotFoundException`. Within the catch clause:
    - Print `"File not found:" + args[0]`
    - Exit the application.
  - Add a catch clause for an `IOException`. Within the catch clause:
    - Print `"Error reading file:"` along with the message available in the `IOException` object
    - Exit the application.

```
try (BufferedReader b =
 new BufferedReader(new FileReader(args[0]));) {
 String s = null;
 while((s = b.readLine()) != null) {
 System.out.println(s);
 }
} catch (FileNotFoundException e) {
 System.out.println("File not found:" + args[0]);
 System.exit(1);
} catch (IOException e) {
 System.out.println("Error reading file:" + e.getMessage());
 System.exit(1);
}
```

13. Run the project. You should see the content of the `D:\labs\resources\DeclarationOfIndependence.txt` file displayed in the output window.

## Practice 9-2: Summary Level: Extending Exception

---

### Overview

In this practice, you will take an existing application and refactor the code to make use of a custom exception class and a custom auto-closeable resource.

### Assumptions

You have reviewed the exception handling section of this lesson.

### Summary

You have been given a project that implements the logic for a human resources application. The application allows for creating, retrieving, updating, deleting, and listing of `Employee` objects. This is the same project that you completed in the “Applying the DAO Pattern” practice.

Currently the only exceptions generated by the DAO implementation (`EmployeeDAOMemoryImpl`) are unchecked exceptions such as `ArrayIndexOutOfBoundsException`.

Future DAO implementations should not require any rewriting of the application logic (`EmployeeTestInteractive`). However, some DAO implementations will generate checked exceptions that must be dealt with. By creating a custom checked exception class that will be used to wrap any DAO generated exceptions, all DAO implementations can appear to generate the same type of exception. This will completely eliminate the need to change any application logic when you create database enabled DAO implementations in later practices.

### Tasks

1. Open the `DAOException` project as the main project.
  - a. Select `File > Open Project`.
  - b. Browse to `D:\labs\09-Exceptions\practices`.
  - c. Select `DAOException` and select the “Open as Main Project” check box.
  - d. Click the Open Project button.
2. Expand the project directories.
3. Run the project. You should see a menu. Test all the menu choices.

|          |        |          |          |        |         |
|----------|--------|----------|----------|--------|---------|
| [C]reate | [R]ead | [U]pdate | [D]elete | [L]ist | [Q]uit: |
|----------|--------|----------|----------|--------|---------|

4. Create a `DAOException` class in the `com.example.dao` package.
5. Complete the `DAOException` class. The `DAOException` class should:
  - Extend the `Exception` class
  - Contain four constructors with parameters matching those of the four public constructors present in the `Exception` class. For each constructor, use `super()` to invoke the parent class constructor with matching parameters.
6. Modify the `EmployeeDAO` interface.
  - All methods should declare that a `DAOException` may be thrown during execution.
  - Extend the `AutoCloseable` interface.

7. Modify the `add` method within the `EmployeeDAOMemoryImpl` class to:

- Declare that a `DAOException` may be produced during execution of this method.
- Use an if statement to validate that an existing employee will not be overwritten by the `add`. If one would, generate a `DAOException` and deliver it to the caller of the method. The `DAOException` should contain a message String indicating what went wrong and why.
- Use a try-catch block to catch the `ArrayIndexOutOfBoundsException` unchecked exception that could possibly be generated.
- Within the catch block that you just created, generate a `DAOException` and deliver it to the caller of the method. The `DAOException` should contain a message String indicating what went wrong and why.

**Note:** Checking the length of the `employeeArray` could be used to determine whether the `DAOException` should be thrown. However, the use of a try-catch block will be typical of the structure used when creating a database-enabled DAO.

8. Modify the `update` method within the `EmployeeDAOMemoryImpl` class to:

- Declare that a `DAOException` may be produced during execution of this method.
- Use an if statement to validate that an existing employee is being updated. If one would not be, generate a `DAOException` and deliver it to the caller of the method. The `DAOException` should contain a message String indicating what went wrong and why.
- Use a try-catch block to catch the `ArrayIndexOutOfBoundsException` unchecked exception that could possibly be generated.
- Within the catch block that you just created, generate a `DAOException` and deliver it to the caller of the method. The `DAOException` should contain a message String indicating what went wrong and why.

9. Modify the `delete` method within the `EmployeeDAOMemoryImpl` class to:

- Declare that a `DAOException` may be produced during execution of this method.
- Use an if statement to validate that an existing employee is being deleted. If one would not be, generate a `DAOException` and deliver it to the caller of the method. The `DAOException` should contain a message String indicating what went wrong and why.
- Use a try-catch block to catch the `ArrayIndexOutOfBoundsException` unchecked exception that could possibly be generated.
- Within the catch block that you just created, generate a `DAOException` and deliver it to the caller of the method. The `DAOException` should contain a message String indicating what went wrong and why.

10. Modify the `findById` method within the `EmployeeDAOMemoryImpl` class to:

- Declare that a `DAOException` may be produced during execution of this method.
- Use a try-catch block to catch the `ArrayIndexOutOfBoundsException` unchecked exception that could possibly be generated.
- Within the catch block that you just created, generate a `DAOException` and deliver it to the caller of the method. The `DAOException` should contain a message String indicating what went wrong and why.

11. Add a close method within the EmployeeDAOMemoryImpl class to implement the AutoCloseable interface.

```
@Override
public void close() {
 System.out.println("No database connection to close just yet");
}
```

**Note:** The EmployeeDAOMemoryImpl class implements EmployeeDAO which extends AutoCloseable and, therefore, EmployeeDAOMemoryImpl class must provide a close method.

12. Modify the EmployeeTestInteractive class to handle the DAOException objects that are thrown by the EmployeeDAO.
- Import the com.example.dao.DAOException class.
  - Modify the executeMenu method to declare that it throws an additional exception of type DAOException.
  - Remove the throws statement from the main method.

```
public static void main(String[] args) throws Exception
```

- Modify the main method to use a try-with-resources statement.
  - Surround the do-while loop with a try block.
  - Convert the EmployeeDAO and BufferedReader references into auto-closed resources.
  - Add a catch clause for an IOException to the end of the try block to handle both I/O errors thrown from the executeMenu method and when auto-closing the BufferedReader.

```
catch (IOException e) {
 System.out.println("Error " + e.getClass().getName() +
 " , quitting.");
 System.out.println("Message: " + e.getMessage());
}
```

- Add a second catch clause for an Exception to the end of the try block to handle errors when auto-closing the EmployeeDAO.

```
catch (Exception e) {
 System.out.println("Error closing resource " +
 e.getClass().getName());
 System.out.println("Message: " + e.getMessage());
}
```

**Note:** At this point the application will compile and run, but DAOException instances generated will cause the application to terminate. For example, if you create an employee with an ID of 100, the application will break out of the do-while loop and pass to this catch clause.

- e. Add a nested try-catch block in the main method that handles exceptions of type `DAOException` that may be thrown by the `executeMenu` method.

```
try {
 timeToQuit = executeMenu(in, dao);
} catch (DAOException e) {
 System.out.println("Error " + e.getClass().getName());
 System.out.println("Message: " + e.getMessage());
}
```

13. Run the project. You should see a menu. Test all the menu choices.

```
[C]reate | [R]ead | [U]pdate | [D]elete | [L]ist | [Q]uit:
```

Attempt to delete an employee that does not exist. You should see a message similar to:

```
Error com.example.dao.DAOException
Message: Error deleting employee in DAO, no such employee 1
```



## Practice 9-2: Detailed Level: Extending Exception

---

### Overview

In this practice, you will take an existing application and refactor the code to make use of a custom exception class and a custom auto-closeable resource.

### Assumptions

You have reviewed the exception handling section of this lesson.

### Summary

You have been given a project that implements the logic for a human resources application. The application allows for creating, retrieving, updating, deleting, and listing of `Employee` objects. This is the same project that you completed in the “Applying the DAO Pattern” practice.

Currently the only exceptions generated by the DAO implementation (`EmployeeDAOMemoryImpl`) are unchecked exceptions such as `ArrayIndexOutOfBoundsException`.

Future DAO implementations should not require any rewriting of the application logic (`EmployeeTestInteractive`). However, some DAO implementations will generate checked exceptions that must be dealt with. By creating a custom-checked exception class that will be used to wrap any DAO generated exceptions, all DAO implementations can appear to generate the same type of exception. This will completely eliminate the need to change any application logic when you create database enabled DAO implementations in later practices.

### Tasks

1. Open the `DAOException` project as the main project.
  - a. Select `File > Open Project`.
  - b. Browse to `D:\labs\09-Exceptions\practices`.
  - c. Select `DAOException` and select the “Open as Main Project” check box.
  - d. Click the Open Project button.
2. Expand the project directories.
3. Run the project. You should see a menu. Test all the menu choices.

|          |        |          |          |        |         |
|----------|--------|----------|----------|--------|---------|
| [C]reate | [R]ead | [U]pdate | [D]elete | [L]ist | [Q]uit: |
|----------|--------|----------|----------|--------|---------|

4. Create a `DAOException` class in the `com.example.dao` package.
5. Complete the `DAOException` class. The `DAOException` class should:
  - Extend the `Exception` class.
  - Contain four constructors with parameters matching those of the four public constructors present in the `Exception` class. For each constructor, use `super()` to invoke the parent class constructor with matching parameters.

|                                                                                       |
|---------------------------------------------------------------------------------------|
| <pre>public class DAOException extends Exception {      public DAOException() {</pre> |
|---------------------------------------------------------------------------------------|

```

 super();
 }

 public DAOException(String message) {
 super(message);
 }

 public DAOException(Throwable cause) {
 super(cause);
 }

 public DAOException(String message, Throwable cause) {
 super(message, cause);
 }
}

```

6. Modify all the methods in the `EmployeeDAO` interface.

- All methods should declare that a `DAOException` may be thrown during execution.
- Extend the `AutoCloseable` interface.

```

public interface EmployeeDAO extends AutoCloseable {

 public void add(Employee emp) throws DAOException;

 public void update(Employee emp) throws DAOException;

 public void delete(int id) throws DAOException;

 public Employee findById(int id) throws DAOException;

 public Employee[] getAllEmployees() throws DAOException;

}

```

7. Modify the `add` method within the `EmployeeDAOMemoryImpl` class to:

- Declare that a `DAOException` may be produced during execution of this method.
- Use an `if` statement to validate that an existing employee will not be overwritten by the `add`. If one would, generate a `DAOException` and deliver it to the caller of the method. The `DAOException` should contain a message `String` indicating what went wrong and why.
- Use a `try-catch` block to catch the `ArrayIndexOutOfBoundsException` unchecked exception that could possibly be generated.

- Within the catch block that you just created, generate a `DAOException` and deliver it to the caller of the method. The `DAOException` should contain a message String indicating what went wrong and why.

```
public void add(Employee emp) throws DAOException {
 if(employeeArray[emp.getId()] != null) {
 throw new DAOException("Error adding employee in DAO,
employee id already exists " + emp.getId());
 }
 try {
 employeeArray[emp.getId()] = emp;
 } catch (ArrayIndexOutOfBoundsException e) {
 throw new DAOException("Error adding employee in DAO, id
must be less than " + employeeArray.length);
 }
}
```

**Note:** Checking the length of the `employeeArray` could be used to determine whether the `DAOException` should be thrown however the use of a try-catch block will be typical of the structure used when create a database enabled DAO.

8. Modify the update method within the `EmployeeDAOMemoryImpl` class to:
  - Declare that a `DAOException` may be produced during execution of this method.
  - Use an if statement to validate that an existing employee is being updated. If one would not be, generate a `DAOException` and deliver it to the caller of the method. The `DAOException` should contain a message String indicating what went wrong and why.
  - Use a try-catch block to catch the `ArrayIndexOutOfBoundsException` unchecked exception that could possibly be generated.
  - Within the catch block that you just created, generate a `DAOException` and deliver it to the caller of the method. The `DAOException` should contain a message String indicating what went wrong and why.

```
public void update(Employee emp) throws DAOException {
 if(employeeArray[emp.getId()] == null) {
 throw new DAOException("Error updating employee in DAO,
no such employee " + emp.getId());
 }
 try {
 employeeArray[emp.getId()] = emp;
 } catch (ArrayIndexOutOfBoundsException e) {
 throw new DAOException("Error updating employee in DAO,
id must be less than " + employeeArray.length);
 }
}
```

9. Modify the delete method within the `EmployeeDAOMemoryImpl` class to:
  - Declare that a `DAOException` may be produced during execution of this method.

- Use an if statement to validate that an existing employee is being deleted. If one would not be, generate a `DAOException` and deliver it to the caller of the method. The `DAOException` should contain a message String indicating what went wrong and why.
- Use a try-catch block to catch the `ArrayIndexOutOfBoundsException` unchecked exception that could possibly be generated.
- Within the catch block that you just created, generate a `DAOException` and deliver it to the caller of the method. The `DAOException` should contain a message String indicating what went wrong and why.

```
public void delete(int id) throws DAOException {
 if(employeeArray[id] == null) {
 throw new DAOException("Error deleting employee in DAO,
no such employee " + id);
 }
 try {
 employeeArray[id] = null;
 } catch (ArrayIndexOutOfBoundsException e) {
 throw new DAOException("Error deleting employee in DAO,
id must be less than " + employeeArray.length);
 }
}
```

10. Modify the `findById` method within the `EmployeeDAOMemoryImpl` class to:

- Declare that a `DAOException` may be produced during execution of this method.
- Use a try-catch block to catch the `ArrayIndexOutOfBoundsException` unchecked exception that could possibly be generated.
- Within the catch block that you just created, generate a `DAOException` and deliver it to the caller of the method. The `DAOException` should contain a message String indicating what went wrong and why.

```
public Employee findById(int id) throws DAOException {
 try {
 return employeeArray[id];
 } catch (ArrayIndexOutOfBoundsException e) {
 throw new DAOException("Error finding employee in DAO",
e);
 }
}
```

11. Add a `close` method within the `EmployeeDAOMemoryImpl` class to implement the `AutoCloseable` interface.

```
@Override
public void close() {
 System.out.println("No database connection to close just
yet");
}
```

**Note:** The `EmployeeDAOMemoryImpl` class implements `EmployeeDAO` which extends `AutoCloseable` and, therefore, `EmployeeDAOMemoryImpl` class must provide a `close` method.

12. Modify the `EmployeeTestInteractive` class to handle the `DAOException` objects that are thrown by the `EmployeeDAO`.

- a. Import the `com.example.dao.DAOException` class.

```
import com.example.dao.DAOException;
```

- b. Modify the `executeMenu` method to declare that it throws an additional exception of type `DAOException`.

```
public static boolean executeMenu(BufferedReader in, EmployeeDAO
dao) throws IOException, DAOException {
```

- c. Remove the `throws` statement from the `main` method.

```
public static void main(String[] args) throws Exception
```

- d. Modify the `main` method to use a `try-with-resources` statement.

- Surround the `do-while` loop with a `try` block.
- Convert the `EmployeeDAO` and `BufferedReader` references into auto-closed resources.
- Add a catch clause for an `IOException` to the end of the `try` block to handle both I/O errors thrown from the `executeMenu` method and when auto-closing the `BufferedReader`.
- Add a second catch clause for an `Exception` to the end of the `try` block to handle errors when auto-closing the `EmployeeDAO`.

```
try (EmployeeDAO dao = factory.createEmployeeDAO();
 BufferedReader in =
new BufferedReader(new InputStreamReader(System.in))) {
 do {
 timeToQuit = executeMenu(in, dao);
 } while (!timeToQuit);
} catch (IOException e) {
 System.out.println("Error " + e.getClass().getName() +
 " , quitting.");
 System.out.println("Message: " + e.getMessage());
} catch (Exception e) {
 System.out.println("Error closing resource " +
 e.getClass().getName());
 System.out.println("Message: " + e.getMessage());
}
```

**Note:** At this point, the application will compile and run, but `DAOException` instances generated will cause the application to terminate. For example, if you create an employee with an ID of 100, the application will break out of the `do-while` loop and pass to this catch clause.

- e. Add a nested try-catch block in the main method that handles exceptions of type `DAOException` that may be thrown by the `executeMenu` method.

```
try {
 timeToQuit = executeMenu(in, dao);
} catch (DAOException e) {
 System.out.println("Error " + e.getClass().getName());
 System.out.println("Message: " + e.getMessage());
}
```

13. Run the project. You should see a menu. Test all the menu choices.

```
[C]reate | [R]ead | [U]pdate | [D]elete | [L]ist | [Q]uit:
```

Attempt to delete an employee that does not exist. You should see a message similar to:

```
Error com.example.dao.DAOException
Message: Error deleting employee in DAO, no such employee 1
```

## **Practices for Lesson 10: Java I/O Fundamentals**

### **Chapter 10**

## Practices for Lesson 10: Overview

---

### Practices Overview

In these practices, you will use some of the java.io classes to read from the console, open and read files, and serialize and deserialize objects to and from the file system.



## Practice 10-1: Summary Level: Writing a Simple Console I/O Application

---

### Overview

In this practice, you will write a simple console-based application that reads from and writes to the system console. In NetBeans, the console is opened as a window in the IDE.

### Tasks

1. Open the project `FileScanner` in the following directory:  
`D:\labs\10-IO_Fundamentals\practices\`
2. Open the file `FileScanInteractive`.  
 Notice that the class has a method called `countTokens` already written for you. This method takes a `String file` and `String search` as parameters. The method will open the file name passed in and use an instance of a `Scanner` to look for the search token. For each token encountered, the method increments the integer field `instanceCount`. When the file is exhausted, it returns the value of `instanceCount`. Note that the class rethrows any `IOException` encountered, so you will need to be sure to use this method inside a try-catch block.
3. Code the main method to check the number of arguments passed. The application expects at least one argument (a string representing the file to open). If the number of arguments is less than one, exit the application with an error code (-1).
  - a. The main method is passed an array of `Strings`. Use the `length` attribute to determine whether the array contains less than one argument.
  - b. Print a message if there is less than one argument, and use `System.exit` to return an error code. (-1 typically is used to indicate an error.)
4. Save the first argument passed into the application as a `String`.
5. Create an instance of the `FileScanInteractive` class. You will need this instance to call the `countTokens` method.
6. Open the system console for input using a buffered reader.
  - a. Use a try-with-resources to open a `BufferedReader` chained to the system console input. (Recall that `System.in` is an input stream connected to the system console.)
  - b. Be sure to add a catch statement to the try block. Any exception returned will be an `IOException` type.
  - c. In a while loop, read from the system console into a string until the string "q" is entered on the console by itself.  
**Note:** You can use `equalsIgnoreCase` to allow your users to enter an upper- or lowercase "Q.". Also the `trim()` method is a good choice to remove any whitespace characters from the input.
  - d. If the string read from the console is not the terminate character, call the `countTokens` method, passing in the file name and the search string.
  - e. Print a string indicating how many times the search token appeared in the file.
  - f. Add any missing import statements.

7. Save the `FileScanInteractive` class.
8. If you have no compilation errors, you can test your application by using a file from the resources directory.
  - a. Right-click the project and select Properties.
  - b. Click Run.
  - c. Enter the name of a file to open in the Arguments text box (for example, `D:\labs\resources\DeclarationOfIndependence.txt`).
  - d. Click OK
  - e. Run the application and try searching for some words like `when`, `rights`, and `free`. Your output should look something like this:

```
Searching through the file:
D:\labs\resources\DeclarationOfIndependence.txt
Enter the search string or q to exit: when
The word "when" appears 3 times in the file.
Enter the search string or q to exit: rights
The word "rights" appears 3 times in the file.
Enter the search string or q to exit: free
The word "free" appears 4 times in the file.
Enter the search string or q to exit: q
BUILD SUCCESSFUL (total time: 16 seconds)
```

## Practice 10-1: Detailed Level: Writing a Simple Console I/O Application

---

### Overview

In this practice, you will write a simple console-based application that reads from and writes to the system console. In NetBeans, the console is opened as a window in the IDE.

### Tasks

1. Open the project `FileScanner` in the following directory:  
`D:\labs\10-IO_Fundamentals\practices\`
  - a. Select `File > Open Project`.
  - b. Browse to `D:\labs\10-IO_Fundamentals\practices`.
  - c. Select `FileScanner` and select the “Open as Main Project” check box.
  - d. Click the Open Project button.

2. Open the file `FileScanInteractive`.

Notice that the class has a method called `countTokens` already written for you. This method takes a `String file` and `String search` as parameters. The method will open the file name passed in and use an instance of a `Scanner` to look for the search token. For each token encountered, the method increments the integer field `instanceCount`. When the file is exhausted, it returns the value of `instanceCount`. Note that the class rethrows any `IOException` encountered, so you will need to be sure to use this method inside a try-catch block.

3. Code the main method to check the number of arguments passed. The application expects at least one argument (a string representing the file to open). If the number of arguments is less than one, exit the application with an error code (-1).
  - a. The main method is passed an array of `Strings`. Use the `length` attribute to determine whether the array contains less than one argument.
  - b. Print a message if there is less than one argument, and use `System.exit` to return an error code. (-1 typically is used to indicate an error.) For example:

```
if (args.length < 1) {
 System.out.println("Usage: java FileScanInteractive <file to
search>");
 System.exit(-1);
}
```

4. Save the first argument passed into the application as a `String`.
 

```
String file = args[0];
```
5. Create an instance of the `FileScanInteractive` class. You will need this instance to call the `countTokens` method.

```
FileScanInteractive scan = new FileScanInteractive ();
```

6. Open the system console for input using a buffered reader.
  - a. Use a try-with-resources to open a `BufferedReader` chained to the system console input. (Recall that `System.in` is an input stream connected to the system console.)
  - b. Be sure to add a catch statement to the try block. Any exception returned will be an `IOException` type. For example:

```
try (BufferedReader in =
 new BufferedReader(new InputStreamReader(System.in))) {

} catch (IOException e) { // Catch any IO exceptions.
 System.out.println("Exception: " + e);
 System.exit(-1);
}
```

- c. In the try block that you created, add a while loop. The while loop should run until a break statement. Inside the while loop, read from the system console into a string until the string "q" is entered on the console by itself.  
**Note:** You can use `equalsIgnoreCase` to allow your users to enter an upper- or lowercase "Q." Also the `trim()` method is a good choice to remove any whitespace characters from the input.
- d. If the string read from the console is not the terminate character, call the `countTokens` method, passing in the file name and the search string.
- e. Print a string indicating how many times the search token appeared in the file.
- f. Your code inside the try block should look something like this:

```
String search = "";
System.out.println ("Searching through the file: " + file);
while (true) {
 System.out.print("Enter the search string or q to exit: ");
 search = in.readLine().trim();
 if (search.equalsIgnoreCase("q")) break;
 int count = scan.countTokens(file, search);
 System.out.println("The word \"" + search + "\" appears "
 + count + " times in the file.");
}
```

- g. Add any missing import statements.
7. Save the `FileScanInteractive` class.

8. If you have no compilation errors, you can test your application by using a file from the resources directory.
  - a. Right-click the project and select Properties.
  - b. Click Run.
  - c. Enter the name of a file to open in the Arguments text box (for example, `D:\labs\resources\DeclarationOfIndependence.txt`).
  - d. Click OK
  - e. Run the application and try searching for some words like `when`, `rights`, and `free`. Your output should look something like this:

```
Searching through the file:
D:\labs\resources\DeclarationOfIndependence.txt
Enter the search string or q to exit: when
The word "when" appears 3 times in the file.
Enter the search string or q to exit: rights
The word "rights" appears 3 times in the file.
Enter the search string or q to exit: free
The word "free" appears 4 times in the file.
Enter the search string or q to exit: q
BUILD SUCCESSFUL (total time: 16 seconds)
```

## Practice 10-2: Summary Level: Serializing and Deserializing a ShoppingCart

---

### Overview

In this practice, you use the `java.io.ObjectOutputStream` class to write a Java object to the file system (serialize), and then use the same stream to read the file back into an object reference. You will also customize the serialization and deserialization of the `ShoppingCart` object.

### Tasks

1. Open the `SerializeShoppingCart` project in the `D:\labs\10-IO_Fundamentals\practices` directory.
2. Expand the `com.example.test` package. Notice there are two Java main classes in this package, `SerializeTest` and `DeserializeTest`. You will be writing the code in these main classes to serialize and deserialize `ShoppingCart` objects.
3. Open the `SerializeTest` class. You will write the methods in this class to write several `ShoppingCart` objects to the file system.
  - a. Read through the code. You will note that the class prompts for the cart ID and constructs an instance of `ShoppingCart` with the cart ID in the constructor.
  - b. The code then adds three `Item` objects to the `ShoppingCart`.
  - c. The code then prints out the number of items in the cart, and the total cost of the items in the cart. Look through the `ShoppingCart` and `Item` classes in the `com.example.domain` package for details on how these classes work.
  - d. You will be writing the code to open an `ObjectOutputStream` and write the `ShoppingCart` as a serialized object on the file system.
4. Create the try block to open a `FileOutputStream` chained to an `ObjectOutputStream`. The file name is already constructed for you.
  - a. Your code will go where the comment line is at the bottom of the file.
  - b. Open a `FileOutputStream` with the `cartFile` string in a try-with-resources block.
  - c. Pass the file output stream instance to an `ObjectOutputStream` to write the serialized object instance to the file.
  - d. Write the `cart` object to the object output stream instance by using the `writeObject` method.
  - e. Be sure to catch any `IOException` and exit with an error as necessary.
  - f. Add a success message before the method ends:
 

```
System.out.println ("Successfully serialized shopping cart with
ID: " + cart.getCartId());
```
  - g. Save the file.
5. Open the `DeserializeTest` class. The main method in this class reads from the console for the ID of the customer shopping cart to deserialize.

6. Your code will go where the comment line is at the bottom of the file.
  - a. Open a `FileInputStream` with the `cartFile` string in a try-with-resources block.
  - b. Pass the file input stream instance to an `ObjectInputStream` to read the serialized object instance from the file.
  - c. Read the `cart` object from the object input stream using the `readObject` method. Be sure to cast the result to the appropriate object type.
  - d. You will need to catch both `ClassNotFoundException` and `IOException`, so use a multi-catch expression.
  - e. Finally, print out the results of the `cart` (all of its contents) and the `cart` total cost using the following code:

```
System.out.println ("Shopping Cart contains: ");
List<Item> cartContents = cart.getItems();
for (Item item : cartContents) {
 System.out.println (item);
}
System.out.println ("Shopping cart total: " +
 NumberFormat.getCurrencyInstance().format(cart.getCartTotal()));
```

- f. Save the file.
7. Open the `ShoppingCart` class. You will customize the serialization and deserialization of this class by adding the two methods called during serialization/deserialization.
  - a. Add a `writeObject` method invoked during serialization. This method should serialize the current object fields and then add a timestamp (`Date` object instance) to end of the object stream.
8. Add a method to the `ShoppingCart` class that is invoked during deserialization.
  - a. Add a `readObject` method with the appropriate signature. This method will recalculate the total cost of the shopping cart and print the timestamp that was added to the stream.
  - b. Save the file.
9. Test the application. This application has two main methods, so you will need to run each main in turn.
  - a. To run the `SerializeTest` class, right-click the class name and select Run File.
  - b. The output will look like this:

```
Enter the ID of the cart file to create and serialize or q exit.
101
Shopping cart 101 contains 3 items
Shopping cart total: $58.39
Successfully serialized shopping cart with ID: 101
```

- c. To run the `DeserializeTest`, right-click the class name and select Run File.

- d. Enter the ID 101 and the output will look like something this:

```
Enter the ID of the cart file to deserialize or q exit.
101
Restored Shopping Cart from: Oct 26, 2011
Successfully deserialized shopping cart with ID: 101
Shopping cart contains:
Item ID: 101 Description: Duke Plastic Circular Flying Disc
Cost: 10.95
Item ID: 123 Description: Duke Soccer Pro Soccer ball Cost:
29.95
Item ID: 45 Description: Duke "The Edge" Tennis Balls - 12-Ball
Bag Cost: 17.49
Shopping cart total: $58.39
```



## Practice 10-2: Detailed Level: Serializing and Deserializing a ShoppingCart

---

### Overview

In this practice, you use the `java.io.ObjectOutputStream` class to write a Java object to the file system (serialize), and then use the same stream to read the file back into an object reference. You will also customize the serialization and deserialization of the `ShoppingCart` object.

### Tasks

1. Open the `SerializeShoppingCart` project in the `D:\labs\10-IO_Fundamentals\practices` directory.
  - a. Select **File > Open Project**.
  - b. Browse to the `D:\labs\10-IO_Fundamentals\practices` directory.
  - c. Select the project `SerializeShoppingCart`.
  - d. Click the **Open Project** button.
2. Expand the `com.example.test` package. Notice there are two Java main classes in this package, `SerializeTest` and `DeserializeTest`. You will be writing the code in these main classes to serialize and deserialize `ShoppingCart` objects.
3. Open the `SerializeTest` class. You will write the methods in this class to write several `ShoppingCart` objects to the file system.
  - a. Read through the code. You will note that the class prompts for the cart ID and constructs an instance of `ShoppingCart` with the cart ID in the constructor.
  - b. The code then adds three `Item` objects to the `ShoppingCart`.
  - c. The code then prints out the number of items in the cart, and the total cost of the items in the cart. Look through the `ShoppingCart` and `Item` classes in the `com.example.domain` package for details on how these classes work.
  - d. You will be writing the code to open an `ObjectOutputStream` and write the `ShoppingCart` as a serialized object on the file system.
4. Create the try block to open a `FileOutputStream` chained to an `ObjectOutputStream`. The file name is already constructed for you.
  - a. Your code will go where the comment line is at the bottom of the file.
  - b. Open a `FileOutputStream` with the `cartFile` string in a `try-with-resources` block.
  - c. Pass the file output stream instance to an `ObjectOutputStream` to write the serialized object instance to the file.
  - d. Write the `cart` object to the object output stream instance by using the `writeObject` method.
  - e. Be sure to catch any `IOException` and exit with an error as necessary.

- f. Your code might look like this:

```
try (FileOutputStream fos = new FileOutputStream (cartFile);
 ObjectOutputStream o = new ObjectOutputStream (fos)) {
 o.writeObject(cart);
} catch (IOException e) {
 System.out.println ("Exception serializing " + cartFile + ": "
+ e);
 System.exit (-1);
}
```

- g. Add a success message before the method ends:

```
System.out.println ("Successfully serialized shopping cart with
ID: " + cart.getCartId());
```

- h. Add any missing import statements.

- i. Save the file.

5. Open the `DeserializeTest` class. The main method in this class reads from the console for the ID of the customer shopping cart to deserialize.

6. Your code will go where the comment line is at the bottom of the file.

- Open a `FileInputStream` with the `cartFile` string in a try-with-resources block.
- Pass the file input stream instance to an `ObjectInputStream` to read the serialized object instance from the file.
- Read the `cart` object from the object input stream using the `readObject` method. Be sure to cast the result to the appropriate object type.
- You will need to catch both `ClassNotFoundException` and `IOException`, so use a multi-catch expression.
- Your code should look like this:

```
try (FileInputStream fis = new FileInputStream (cartFile);
 ObjectInputStream in = new ObjectInputStream (fis)) {
 cart = (ShoppingCart)in.readObject();
} catch (final ClassNotFoundException | IOException e) {
 System.out.println ("Exception deserializing " + cartFile +
": " + e);
 System.exit (-1);
}
System.out.println ("Successfully deserialized shopping cart
with ID: " + cart.getCartId());
```

- f. Finally, print out the results of the cart (all of its contents) and the cart total cost using the following code:

```
System.out.println ("Shopping cart contains: ");
List<Item> cartContents = cart.getItems();
for (Item item : cartContents) {
 System.out.println (item);
}
System.out.println ("Shopping cart total: " +
 NumberFormat.getCurrencyInstance().format(cart.getCartTotal()));
```

- g. Save the file.

7. Open the ShoppingCart class. You will customize the serialization and deserialization of this class by adding the two methods called during serialization/deserialization.

- a. Add a method invoked during serialization that will add a timestamp (Date object instance) to the end of the object stream.
- b. Add a method with the signature:

```
private void writeObject(ObjectOutputStream oos) throws
IOException {
```

- c. Make sure that the method serializes the current object fields first, and then write the Date object instance:

```
oos.defaultWriteObject();
oos.writeObject(new Date());
}
```

8. Add a method to the ShoppingCart class that is invoked during deserialization. This method will recalculate the total cost of the shopping cart and print the timestamp that was added to the stream.

- a. Add a method with the signature:

```
private void readObject(ObjectInputStream ois) throws
IOException, ClassNotFoundException {
```

- b. This method will deserialize the fields from the object stream, and recalculate the total dollar value of the current cart contents:

```
ois.defaultReadObject();
if (cartTotal == 0 && (items.size() > 0)) {
 for (Item item : items)
 cartTotal += item.getCost();
}
```

- c. Get the Date object from the serialized stream and print the timestamp to the console.

```
Date date = (Date)ois.readObject();
System.out.println ("Restored Shopping Cart from: " +
 DateFormat.getDateInstance().format(date));
}
```

- d. Save the ShoppingCart.

9. Test the application. This application has two main methods, so you will need to run each main in turn.
- To run the `SerializeTest` class, right-click the class name and select Run File. Enter a cart id, such as 101.
  - The output will look like this:

```
Enter the ID of the cart file to create and serialize or q exit.
101
Shopping cart 101 contains 3 items
Shopping cart total: $58.39
Successfully serialized shopping cart with ID: 101
```

- To run the `DeserializeTest`, right-click the class name and select Run File.
- Enter the ID 101 and the output will look like this:

```
Enter the ID of the cart file to deserialize or q exit.
101
Restored Shopping Cart from: Oct 26, 2011
Successfully deserialized shopping cart with ID: 101
Shopping cart contains:
Item ID: 101 Description: Duke Plastic Circular Flying Disc
Cost: 10.95
Item ID: 123 Description: Duke Soccer Pro Soccer ball Cost:
29.95
Item ID: 45 Description: Duke "The Edge" Tennis Balls - 12-Ball
Bag Cost: 17.49
Shopping cart total: $58.39
```

## **Practices for Lesson 11: Java File I/O (NIO.2)**

### **Chapter 11**

## Practices for Lesson 11: Overview

---

### Practices Overview

In the first practice, you will use the JDK 7 NIO.2 API to write an application to create custom letters by merging a template letter with a list of names, utilizing `Files` and `Path` methods. In the second practice, you will use the `walkFileTree` method to copy all the files and directories from one folder to another on the disk. In the final optional practice, you will use the same method to write an application to recursively find and delete all the files that match a supplied pattern.

## Practice 11-1: Summary Level: Writing a File Merge Application

---

### Overview

In this practice, you will use the `Files.readAllLines` method to read the entire contents of two files: a form template, and a list of names to send form letters to. After creating a form letter with a name from the name list, you will use the `Files.write` method to create the custom letter. You will also use the `Pattern` and `Matcher` classes that you saw in the “String Processing” lesson.

### Assumptions

You participated in the lecture for this lesson. Note there are Netbeans projects in the example directory to help you understand how to use the `Files` class `readAllLines` and `write` methods.

### Tasks

1. Open the file `FormTemplate` in the `resources` directory.  
Note that this is a form letter with a string `<NAME>` that will be replaced by a name from the name list file.
2. Open the file `NamesList.txt` in the `resources` directory.
  - a. This file contains the names to send the form letters to.
  - b. Add your name to the end of the list.
  - c. Save the file.
3. Open the project `FormLetterWriter` in the `practices` directory.
4. Expand the `FormLetterWriter` class. Notice that this class contains the main method, and that the application requires two parameters: One is the path to the form letter template, and the second is the path to the file containing the list of names to substitute in the form letter.
  - a. After checking for a valid number of arguments, the main method then checks to see whether the `Path` objects point to valid files.
  - b. The main method creates an instance of the `FileMerge` class with the form letter `Path` object and the list of names `Path` object.
  - c. In a try block, the main method calls the `writeMergedForm` method of the `FileMerge` class. This is the method that you will write in this practice.
5. Expand the `FileMerge` class.
  - a. Note the `writeMergedForms` method is empty. This is the method that you will write in this practice.
  - b. The `mergeName` method uses the `Pattern` object defined in the field declarations to replace the string from the form template (first argument) with a name from the name list (second argument). It returns a `String`. For example, it replaces "Dear `<NAME>`," with "Dear Wilson Ball,".
  - c. The `hasToken` method returns a `boolean` to indicate whether the string passed in contains the token. This is useful to identify which string has the token to replace with the name from the name list.

6. Code the `writeMergedForms` method. The overall plan for this method is to read in the entire form letter, line by line, and then read in the entire list of names and merge the names with the form letter, replacing the placeholder in the template with a name from the list and then writing that out as a file. The net result should be that if you have ten names in the name list, you should end up with ten custom letter files addressed to the names from the name list. These ten files will be written to the resources directory.
  - a. Read in all of the lines of the form letter into the `formLetter` field, and all of the lines from the name list into the `nameList` field.
 

**Note:** Because `writeMergedForms` throws `IOException`, you do not need to put these statements into a try block. The caller of this method is responsible for handling any exceptions thrown.
  - b. Create a `for` loop to iterate through the list of names (`nameList`) strings.
  - c. Inside this `for` loop, create a new `List` object to hold the strings of the form letter. You need this new `List` to hold the modified form template strings to write out.
  - d. Still inside the `for` loop, you will need to create a name for the custom letter. One easy way to do this is to use the name from the name list. You should replace any spaces in the name with underscores for readability of the file name. Create a new `Path` object relative to the form template path.
  - e. Create another `for` loop, nested in the first loop, to iterate through the lines of the form template and look for the token string ("`<NAME>`") to replace with the `String` name from the `nameList`. Use the `hasToken` method to look for the `String` that contains the token string and replace that string with one containing the name from the `nameList`. Use the `mergeName` method to create the new `String`. Add the modified `String` and all of the other `Strings` from the `formLetter` to the new `customLetter` `List`.
  - f. Still inside the first `for` loop, write the modified `List` of `Strings` that represents the customized form letter to the file system by using the `Files.write` method. Print a message that the file write was successful and close the outer `for` loop.
  - g. Save the `FileMerge` class.
7. Modify the `FormLetterWriter` project to pass the form letter file and the name list file to the main method.
  - a. Right-click the project and select Properties.
  - b. Select Run.
  - c. In the Arguments text field, enter: `D:\labs\resources\FormTemplate.txt`  
`D:\labs\resources\NamesList.txt`
  - d. Click OK.



8. Run the project. You should see new files created with the names from the name list. Each file should be customized with the name from the name list. For example, the Tom\_McGinn.txt file should contain:

Dear Tom McGinn,

It has come to our attention that you would like to prove your Java Skills. May we recommend that you consider certification from Oracle? Oracle has globally recognized Certification exams that will test your Java knowledge and skills.

Start with the Oracle Certified Java Associate exam, and then continue to the Oracle Certified Java Programmer Professional for a complete certification profile.

Good Luck!

Oracle University

## Practice 11-1: Detail Level: Writing a File Merge Application

---

### Overview

In this practice, you will use the `Files.readAllLines` method to read the entire contents of two files: a form template, and a list of names to send form letters to. After creating a form letter with a name from the name list, you will use the `Files.write` method to create the custom letter. You will also use the `Pattern` and `Matcher` classes that you saw in the “String Processing” lesson.

### Assumptions

You participated in the lecture for this lesson. Note there are Netbeans projects in the example directory to help you understand how to use the `Files` class `readAllLines` and `write` methods.

### Tasks

1. Open the file `FormTemplate` in the `resources` directory.
  - a. Select `File > Open File`
  - b. Navigate to the `resources` directory in `D:\labs`
  - c. Select the file `FormTemplate.txt` and click the `Open` button.

Note that this is a form letter with a string placeholder token `<NAME>` that will be replaced by a name from the name list file.

2. Open the file `NamesList.txt` in the `resources` directory.
  - a. This file contains the names to send the form letters to.
  - b. Add your name to the end of the list.
  - c. Save the file.
3. Open the project `FormLetterWriter` in the `practices` directory.
  - a. Select `File > Open Project`.
  - b. Browse to `D:\labs\11-NIO.2\practices`.
  - c. Select `FormLetterWriter`.
  - d. Select the “Open as Main Project” check box.
  - e. Click the `Open Project` button.
4. Expand the `FormLetterWriter` class. Notice that this class contains the main method, and that the application requires two parameters: One is the path to the form letter template, and the second is the path to the file containing the list of names to substitute in the form letter.
  - a. After checking for a valid number of arguments, the main method then checks to see whether the `Path` objects point to valid files.
  - b. The main method creates an instance of the `FileMerge` class with the form letter `Path` object and the list of names `Path` object.
  - c. In a try block, the main method calls the `writeMergedForm` method of the `FileMerge` class. This is the method that you will write in this practice.

5. Expand the `FileMerge` class.
  - a. Note the `writeMergedForms` method is empty. This is the method that you will write in this practice.
  - b. The `mergeName` method uses the `Pattern` object defined in the field declarations to replace the string from the form template (first argument) with a name from the name list (second argument). It returns a `String`. For example, it replaces "Dear <NAME>," with "Dear Wilson Ball,".
  - c. The `hasToken` method returns a `boolean` to indicate whether the string passed in contains the token. This is useful to identify which string has the token to replace with the name from the name list.
6. Code the `writeMergedForms` method. The overall plan for this method is to read in the entire form letter, line by line, and then read in the entire list of names and merge the names with the form letter, replacing the placeholder in the template with a name from the list and then writing that out as a file. The net result should be that if you have ten names in the name list, you should end up with ten custom letter files addressed to the names from the name list. These ten files will be written to the resources directory.
  - a. Create an instance of the default `Charset`. This argument is required for the `Files.readAllLines` method.
 

```
Charset cs = Charset.defaultCharset();
```
  - b. Read in all of the lines of the form letter into the `formLetter` field, and all of the lines from the name list into the `nameList` field.

**Note:** Because `writeMergedForms` throws `IOException`, you do not need to put these statements into a try block. The caller of this method is responsible for handling any exceptions thrown.

```
formLetter = Files.readAllLines(form, cs);
nameList = Files.readAllLines(list, cs);
```

- c. Create a `for` loop to iterate through the list of names (`nameList`) strings.
- d. Inside this `for` loop, create a new `List` object to hold the strings of the form letter. You will need this new `List` to hold the modified form template strings to write out.

```
for (int j = 0; j < nameList.size(); j++) {
 customLetter = new ArrayList<>();
```

- e. Still inside the `for` loop, you need to create a name for the custom letter. One easy way to do this is to use the name from the name list. You should replace any spaces in the name with underscores for readability of the file name. Create a new `Path` object relative to the form template path.

```
String formName = nameList.get(j).replace(' ',
'_').concat(".txt");
Path formOut = form.getParent().resolve(formName);
```

- f.

Create another `for` loop, nested in the first loop, to iterate through the lines of the form template and look for the token placeholder string ("`<NAME>`") to replace with the `String` `name` from the `nameList`. Use the `hasToken` method to look for the `String` that contains the token string and replace that string with one containing the name from the `nameList`. Use the `mergeName` method to create the new `String`. Add the modified `String` and all of the other `Strings` from the `formLetter` to the new `customLetter List`.

```
for (int k = 0; k < formLetter.size(); k++) {
 if (hasToken(formLetter.get(k))) {
 customLetter.add(mergeName(formLetter.get(k),
nameList.get(j)));
 } else {
 customLetter.add(formLetter.get(k));
 }
}
```

- g. Finally, still inside the first `for` loop, write the modified `List of Strings` that represents the customized form letter to the file system by using the `Files.write` method. Print a message that the file write was successful and close the outer `for` loop.

```
Files.write(formOut, customLetter, cs);
System.out.println ("Wrote form letter to: " +
nameList.get(j));
} // closing brace for the outer for loop
```

- h. Reformat the code to ensure that you have everything in the right place. Press the `Ctrl-Alt-F` key combination or right-click in the editor pane and choose `Format`.
- i. Save the `FileMerge` class.
7. Modify the `FormLetterWriter` project to pass the form letter file and the name list file to the main method.
- Right-click the project and select `Properties`.
  - Select `Run`.
  - In the `Arguments` text field, enter: `D:\labs\resources\FormTemplate.txt`  
`D:\labs\resources\NamesList.txt`
  - Click `OK`.

8. Run the project. You should see new files created with the names from the name list. Each file should be customized with the name from the name list. For example, the Tom\_McGinn.txt file should contain:

Dear Tom McGinn,

It has come to our attention that you would like to prove your Java Skills. May we recommend that you consider certification from Oracle? Oracle has globally recognized Certification exams that will test your Java knowledge and skills.

Start with the Oracle Certified Java Associate exam, and then continue to the Oracle Certified Java Programmer Professional for a complete certification profile.

Good Luck!

Oracle University

## Practice 11-2: Summary Level: Recursive Copy

---

### Overview

In this practice, you write Java classes that use the `FileVisitor` class to recursively copy one directory to another.

### Assumptions

You participated in the lecture for this lesson.

### Tasks

1. Open the project `RecursiveCopyExercise` in the directory `D:\labs\11-NIO.2\practices`.
2. Expand the `Source Packages` folder and subfolders and look at the `Copy.java` class.
  - a. Note that the `Copy.java` class contains the `main` method.
  - b. The application takes two arguments, a `source` and `target` paths.
  - c. If the target file or directory exists, the user is prompted whether to overwrite.
  - d. If the answer is yes (or the letter y), the method continues.
  - e. An instance of the `CopyFileTree` class is created with the `source` and `target`.
  - f. This instance is then passed to the `walkFileTree` method (with the `source Path` object).

You will need to provide method bodies for the methods in the `CopyFileTree.java` class.

3. Open the `CopyFileTree.java` class.
  - a. This class implements the `FileVisitor` interface. Note that `FileVisitor` is a generic interface, and this interface is boxed with the `Path` class. This allows the interface to define the type of the arguments passed to its methods.
  - b. The `CopyFileTree` implements all the methods defined by `FileVisitor`.
  - c. Your task is to write method bodies for the `preVisitDirectory` and `visitFile` methods. You will not need the `postVisitDirectory` method, and you have been provided a method body for the `visitFileFailed` method.
4. Write the method body for `preVisitDirectory`. This method is called for the starting node of the tree and every subdirectory. Therefore, you should copy the directory of the source to the target. If the file already exists, you can ignore that exception (because you are doing the copy because the user elected to overwrite.)
  - a. Start by creating a new directory that is relative to the target passed in, but is the node name from the source. The method call to do this is:
 

```
Path newdir = target.resolve(source.relativeize(dir));
```
  - b. In a try block, copy the directory passed to the `preVisitDirectory` method to the `newdir` that you created.

- c. You can ignore any `FileAlreadyExistsException` thrown, because you are overwriting any existing folders and files in this copy.
  - d. Catch any other `IOExceptions`, and use the `SKIP_SUBTREE` return to avoid repeated errors.
5. Write the method body for the `visitFile` method. This method is called when the node reached is a file. The file is passed as an argument to the method.
- a. As with the `preVisitDirectory`, you must rationalize the file reached (source path) with the path that you wanted for the target. Use the same method call as above (only using `file` instead of `dir`):
 

```
Path newdir = target.resolve(source.relativeTo(file));
```
  - b. As in the `preVisitDirectory` method, use the `Files.copy` method in a try block. Make sure that you pass `REPLACE_EXISTING` in as an option to overwrite any existing file in the directory.
  - c. Catch any `IOException` thrown and report an error.
  - d. Fix any missing imports.
  - e. Save your class.
6. Test your application by copying a directory (ideally with subdirectories) to another location on the disk. For example, copy the `D:\labs\11-NIO.2` directory to `D:\Temp`.
- a. Right-click the project and select Properties.
  - b. Click Run.
  - c. Enter the following as Arguments:
 

```
D:\labs\11-NIO.2 D:\Temp
```
  - d. Click OK.
7. Run the project and you should see the following message:
- ```
Successfully copied D:\labs\11-NIO.2 to D:\Temp
```
- a. Run the project again, and you should be prompted:


```
Target directory exists. Overwrite existing files? (yes/no):
```

Practice 11-2: Detailed Level: Recursive Copy

Overview

In this practice, you write Java classes that use the `FileVisitor` class to recursively copy one directory to another.

Assumptions

You participated in the lecture for this lesson.

Tasks

1. Open the project `RecursiveCopyExercise` in the directory `D:\labs\11-NIO.2\practices`.
 - a. Select `File > Open Project`.
 - b. Browse to `D:\labs\11-NIO.2\practices`.
 - c. Select `RecursiveCopyExercise`.
 - d. Select the "Open as Main Project" check box.
 - e. Click the `Open Project` button.
2. Expand the `Source Packages` folder and subfolders and look at the `Copy.java` class.
 - a. Note that the `Copy.java` class contains the `main` method.
 - b. The application takes two arguments, a `source` and `target` paths.
 - c. If the target file or directory exists, the user is prompted whether to overwrite.
 - d. If the answer is yes (or the letter `y`), the method continues.
 - e. An instance of the `CopyFileTree` class is created with the `source` and `target`.
 - f. This instance is then passed to the `walkFileTree` method (with the `source Path` object).

You will need to provide method bodies for the methods in the `CopyFileTree.java` class.

3. Open the `CopyFileTree.java` class.
 - a. This class implements the `FileVisitor` interface. Note that `FileVisitor` is a generic interface, and this interface is boxed with the `Path` class. This allows the interface to define the type of the arguments passed to its methods.
 - b. The `CopyFileTree` implements all the methods defined by `FileVisitor`.
 - c. Your task is to write method bodies for the `preVisitDirectory` and `visitFile` methods. You will not need the `postVisitDirectory` method, and you have been provided a method body for the `visitFileFailed` method.

4. Write the method body for `preVisitDirectory`. This method is called for the starting node of the tree and every subdirectory. Therefore, you should copy the directory of the source to the target. If the file already exists, you can ignore that exception (because you are doing the copy because the user elected to overwrite.)

- a. Start by creating a new directory that is relative to the target passed in, and is the node name from the source. The method call to do this is:

```
Path newdir = target.resolve(source.relativeTo(dir));
```

- b. In a try block, copy the directory passed to the `preVisitDirectory` method to the `newdir` that you created.

```
try {
    Files.copy(dir, newdir);
```

- c. You can ignore any `FileAlreadyExistsException` thrown, because you are overwriting any existing folders and files in this copy.

```
} catch (FileAlreadyExistsException x) {
    // ignore
```

- d. Do catch any other `IOExceptions`, and use the `SKIP_SUBTREE` return to avoid repeated errors.

```
} catch (IOException x) {
    System.err.format("Unable to create: %s: %s%n",
                      newdir, x);
    return SKIP_SUBTREE;
}
```

5. Write the method body for the `visitFile` method. This method is called when the node reached is a file. The file is passed as an argument to the method.

- a. As with the `preVisitDirectory`, you must rationalize the file reached (source path) with the path that you wanted for the target. Use the same method call as above (only using `file` instead of `dir`):

```
Path newdir = target.resolve(source.relativeTo(file));
```

- b. As in the `preVisitDirectory` method, use the `Files.copy` method in a try block. Make sure that you pass `REPLACE_EXISTING` in as an option to overwrite any existing file in the directory.

```
try {
    Files.copy(file, newdir, REPLACE_EXISTING);
```

Note: To use the `REPLACE_EXISTING` enum type, you must import the `java.nio.file.StandardCopyOption` enum class using a static import, like this:

```
import static java.nio.file.StandardCopyOption.*;
```

- c. Catch any `IOException` thrown and report an error.

```
} catch (IOException x) {
    System.err.format("Unable to copy: %s: %s%n", source, x);
}
```

- d. Fix any missing imports.
 - e. Save your class.
6. Test your application by copying a directory (ideally with subdirectories) to another location on the disk. For example, copy the `D:\labs\11-NIO.2` directory to `D:\Temp`.
- a. Right-click the project and select Properties.
 - b. Click Run.
 - c. Enter the following as Arguments:
`D:\labs\11-NIO.2 D:\Temp`
 - d. Click OK.
7. Run the project and you should see the following message:
- `Successfully copied D:\labs\11-NIO.2 to D:\Temp`
- a. Run the project again, and you should be prompted:
`Target directory exists. Overwrite existing files? (yes/no):`

(Optional) Practice 11-3: Summary Level: Using PathMatcher to Recursively Delete

Overview

In this practice, you write a Java main that creates a `PathMatcher` class and uses `FileVisitor` to recursively delete a file or directory pattern.

Assumptions

You have completed the previous practice.

Tasks

1. Open the project `RecursiveDeleteExercise` in the practices directory.
2. Expand the Source Packages folders.
3. Open the `Delete.java` class file. This is the class that contains the `main` method. The main class accepts two arguments: the first is the starting path and the other the pattern to delete.
4. You must code the remainder of this class. Look at the comments for hints as to what to do.
 - a. Start by creating a `PathMatcher` object from the search string passed in as the second argument. To obtain a `PathMatcher` instance, you will need to use the `FileSystems` class to get a path matcher instance from the default file system.
 - b. Create a `Path` object from the first argument.
 - c. If the starting path is a file, check it against the pattern using the `PathMatcher` instance that you created. If there is a match, delete the file, and then terminate the application.
 - d. If the starting path is a directory, create an instance of the `DeleteFileTree` with the starting directory and the `PathMatcher` object as initial arguments in the constructor. Pass the starting directory and the file tree to a `Files.walkFileTree` method to recursively look for the pattern to delete.
 - e. Fix any missing imports.
 - f. Save the `Delete` class.
5. Open the `DeleteFileTree` class file. This class implements `FileVisitor`. This class recursively looks for instances of files or directories that match the `PathMatcher` object passed into the constructor. This class is complete with the exception of the `delete` method.
 - a. The `delete` method is called by the `preVisitDirectory` and `visitFile` methods. You must check whether the file or directory reached by these methods matches the pattern.
 - b. We only want to match the path name at the node, so use the `Path.getFileName` method to obtain the file name at the end of the full path.
 - c. If the name matches, use the `Files.delete` method to attempt to delete the file pattern and print a result statement, or print an error if an `IOException` is thrown.
 - d. Save the `DeleteFileTree` class.

6. Run the `Delete` application using a temporary directory.
 - a. For example, if you completed the first practice, you can delete all the Java class files from the `D:\Temp` directory.
 - b. Right-click the project and select Properties.
 - c. Click Run and enter the following in the Arguments text field:
`D:\Temp\examples *.class`
 - d. Run the project.

(Optional) Practice 11-3: Detailed Level: Using PathMatcher to Recursively Delete

Overview

In this practice, you write a Java main that creates a `PathMatcher` class and uses `FileVisitor` to recursively delete a file or directory pattern.

Assumptions

You have completed the previous practice.

Tasks

1. Open the project `RecursiveDeleteExercise` in the practices directory.
 - a. Select File > Open Project.
 - b. Browse to `D:\labs\11-NIO.2\practices`.
 - c. Select `RecursiveDeleteExercise`.
 - d. Click the Open Project button.
2. Expand the Source Packages folders.
3. Open the `Delete.java` class file. This is the class that contains the `main` method. The main class accepts two arguments: the first is the starting path and the other the pattern to delete.
4. You must code the remainder of this class. Look at the comments for hints as to what to do.
 - a. Start by creating a `PathMatcher` object from the search string passed in as the second argument. To obtain a `PathMatcher` instance, you will need to use the `FileSystems` class to get a path matcher instance from the default file system.


```
PathMatcher matcher =
FileSystems.getDefault().getPathMatcher("glob:" + args[1]);
```
 - b. Create a `Path` object from the first argument.


```
Path root = Paths.get(args[0]);
```

- c. If the starting path is a file, check it against the pattern using the `PathMatcher` instance that you created. If there is a match, delete the file, and then terminate the application.

```
if (!Files.isDirectory(root)) {
    Path name = root.getFileName();
    if (name != null && matcher.matches(name)) {
        try {
            Files.delete(root);
            System.out.println("Deleted : " + root);
            System.exit(0);
        } catch (IOException e) {
            System.err.println("Exception deleting file: " +
                               root);
            System.err.println("Exception: " + e);
            System.exit(-1);
        }
    }
}
```

- d. If the starting path is a directory, create an instance of the `DeleteFileTree` with the starting directory and the `PathMatcher` object as initial arguments in the constructor. Pass the starting directory and the file tree to a `Files.walkFileTree` method to recursively look for the pattern to delete.

```
DeleteFileTree deleter = new DeleteFileTree(root, matcher);
try {
    Files.walkFileTree(root, deleter);
} catch (IOException e) {
    System.out.println("Exception: " + e);
}
```

- e. Fix any missing imports.
- f. Save the `Delete` class.
5. Open the `DeleteFileTree` class file. This class implements `FileVisitor`. This class recursively looks for instances of files or directories that match the `PathMatcher` object passed into the constructor. This class is complete with the exception of the `delete` method.
- a. The `delete` method is called by the `preVisitDirectory` and `visitFile` methods. You must check whether the file or directory reached by these methods matches the pattern.
- b. We only want to match the path name at the node, so use the `Path.getFileName` method to obtain the file name at the end of the full path.

```
Path name = file.getFileName();
```

- c. If the name matches, use the `Files.delete` method to attempt to delete the file pattern and print a result statement, or print an error if an `IOException` is thrown.

```

if (matcher.matches(name)) {
    //if (name != null && matcher.matches(name)) {
    try {
        Files.delete(file);
        System.out.println("Deleted: " + file);
    } catch (IOException e) {
        System.err.println("Unable to delete: " + name);
        System.err.println("Exception: " + e);
    }
}

```

- d. Save the `DeleteFileTree` class.
6. Run the `Delete` application using a temporary directory.
- For example, if you completed the first practice, you can delete all the Java class files from the `D:\Temp` directory.
 - Right-click the project and select `Properties`.
 - Click `Run` and enter the following in the `Arguments` text field:
`D:\Temp\examples *.class`
 - Run the project.

NAVEEN UNNIKRIISHNAN (naveenunnikrishnan3.14@gmail.com)
has a non-transferable license to use this Student Guide.

Practices for Lesson 12: Threading

Chapter 12

NAVEEN UNNIKRISHNAN (naveenuunnikrishnan14@gmail.com)
has a non-transferable license to use this Student Guide.

Practices for Lesson 12: Overview

Practices Overview

In these practices, you will use the multithreaded features of the Java programming language.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

NAVEEN UNNIKRIISHNAN (naveenunnikrishnan3.14@gmail.com)
has a non-transferable license to use this Student Guide.

Practice 12-1: Summary Level: Synchronizing Access to Shared Data

Overview

In this practice, you will add code to an existing application. You must determine whether the code is run in a multithreaded environment, and, if so, make it thread-safe.

Assumptions

You have reviewed the sections covering the use of the `Thread` class and the `synchronized` keyword of this lesson.

Summary

You will open a project that purchases shirts from a store. The file-reading code will be provided to you. Your task is to add the appropriate exception handling code.

Tasks

1. Open the project `Synchronized` as the main project.
 - a. Select `File > Open Project`.
 - b. Browse to `D:\labs\12-Threading\practices`.
 - c. Select `Synchronized` and select the "Open as Main Project" check box.
 - d. Click the Open Project button.
2. Expand the project directories but avoid opening and review the provided classes at this point. You will attempt to discover whether this application is multithreaded by observing the behavior of code that you provide.
3. Create a `PurchasingAgent` class in the `com.example` package.
4. Complete the `PurchasingAgent` class.
 - a. Add a purchase method.

```
public void purchase() {}
```

- b. Complete the purchase method. The `purchase()` method should:
 - Obtain a `Store` reference. Note that the `Store` class implements the Singleton design pattern.

```
Store store = Store.getInstance();
```

- Buy a Shirt.
 - Verify that the store has at least one shirt in stock.

```
store.getShirtCount()
```

- Use the store to authorize a credit card purchase. Use a credit card account number of "1234" and a purchase amount of 15.00. A `boolean` result is returned.

```
store.authorizeCreditCard("1234", 15.00)
```

- If there are shirts in stock and the credit card purchase was authorized, you should take a shirt from the store.

```
Shirt shirt = store.takeShirt();
```

- Print out the shirt and a success message if a shirt was acquired or a failure message if one was not.
5. Run the project multiple times. Note that the store contains only a single shirt. You can see many possible variations of output. You might see:
 - Two success messages and two shirts (output may appear in varying order)
 - Two success messages, one shirt, and one null
 - Two success messages, one shirt, and one exception
 - One success message, one shirt, and one failure message (desired behavior, but least likely)
 6. Discover how the `PurchasingAgent` class is being used.
 - a. Use a constructor and a print statement to discover how many instances of the `PurchasingAgent` class are being created when running the application.

Reminder: Sometimes objects are created per-request and sometimes an object may be shared by multiple requests. The variations in the model affect which code must be thread-safe.
 - b. Within the `purchase` method use the `Thread.currentThread()` method to obtain a reference to the thread currently executing the `purchase()` method. Use a single print statement to print the name and ID of the executing thread.
 - c. Run the project and observe the output.
 7. Open the `Store` class and add a delay to the `authorizeCreditCard` method.
 - Obtain a random number in the range of 1–3, the number of seconds to delay. Print a message indicating how many seconds execution will be delayed.

```
int seconds = (int) (Math.random() * 3 + 1);
```

 - Use the appropriate static method in the `Thread` class to delay execution for 1 to 3 seconds.

Optional Task: What if your delay is interrupted? How can you be sure that execution is delayed for the desired number of seconds? Or should a different action be taken?
 8. Run the project multiple times. You should see a stack trace for a `java.util.NoSuchElementException`. Locate the line within the `com.example.PurchasingAgent.purchase` method that is displayed in the stack trace. Review the action occurring on that line.

9. Use a `synchronized` code block to create predictable behavior.
 - Modify the `purchase` method in the `PurchasingAgent` class to contain a `synchronized` code block.
Note: Adding `synchronized` to the method signature or using a `synchronized` block that uses the `this` object's monitor will not work.
10. Run the project. You should now see the desired behavior. In the output window, you should see one success message, one shirt, and one failure message.

Practice 12-1: Detailed Level: Synchronizing Access to Shared Data

Overview

In this practice, you will add code to an existing application. You must determine whether the code is run in a multithreaded environment, and, if so, make it thread-safe.

Assumptions

You have reviewed the sections covering the use of the `Thread` class and the `synchronized` keyword of this lesson.

Summary

You will open a project that purchases shirts from a store. The file-reading code will be provided to you. Your task is to add the appropriate exception handling code.

Tasks

1. Open the project `Synchronized` as the main project.
 - a. Select `File > Open Project`.
 - b. Browse to `D:\labs\12-Threading\practices`.
 - c. Select `Synchronized` and select the "Open as Main Project" check box.
 - d. Click the Open Project button.
2. Expand the project directories but avoid opening and review the provided classes at this point. You will attempt to discover whether this application is multithreaded by observing the behavior of code that you provide.
3. Create a `PurchasingAgent` class in the `com.example` package.
4. Complete the `PurchasingAgent` class.
 - a. Add a purchase method. The `purchase()` method should:
 - Obtain a `Store` reference. Note that the `Store` class implements the Singleton design pattern.
 - Buy a `Shirt`.
 - Verify that the store has at least one shirt in stock.
 - Use the store to authorize a credit card purchase. Use a credit card account number of "1234" and a purchase amount of 15.00. A `boolean` result is returned.
 - If there are shirts in stock and the credit card purchase was authorized, you should take a shirt from the store.
 - Print out the shirt and a success message if a shirt was acquired or a failure message if one was not.

```

public class PurchasingAgent {

    public void purchase() {
        Store store = Store.getInstance();
        if (store.getShirtCount() > 0 &&
store.authorizeCreditCard("1234", 15.00)) {
            Shirt shirt = store.takeShirt();
            System.out.println("The shirt is ours!");
            System.out.println(shirt);
        } else {
            System.out.println("No shirt for you");
        }
    }
}

```

5. Run the project multiple times. Note that the store contains only a single shirt. You can see many possible variations of output. You might see:

- Two success messages and two shirts (output may appear in varying order)

```

Adding a shirt to the store.
Total shirts in stock: 1
The shirt is ours!
The shirt is ours!
Shirt ID: 1
Description: Polo
Color: Rainbow
Size: Large

Shirt ID: 1
Description: Polo
Color: Rainbow
Size: Large

```

- Two success messages, one shirt, and one null

```

Adding a shirt to the store.
Total shirts in stock: 1
The shirt is ours!
The shirt is ours!
null
Shirt ID: 1
Description: Polo
Color: Rainbow
Size: Large

```

- Two success messages, one shirt, and one exception

```

Adding a shirt to the store.
Total shirts in stock: 1
The shirt is ours!
Shirt ID: 1
Description: Polo
Color: Rainbow
Size: Large
Exception in thread "Thread-0" java.util.NoSuchElementException

```

- One success message, one shirt, and one failure message (desired behavior but least likely)

```

Adding a shirt to the store.
Total shirts in stock: 1
The shirt is ours!
Shirt ID: 1
Description: Polo
Color: Rainbow
Size: Large

No shirt for you

```

6. Discover how the `PurchasingAgent` class is being used.
 - a. In the `PurchasingAgent` class, use a constructor and a print statement to discover how many instance of the `PurchasingAgent` class are being created when running the application.

```

public PurchasingAgent() {
    System.out.println("Creating a purchasing agent");
}

```

Reminder: Sometimes objects are created per-request and sometimes an object may be shared by multiple requests. The variations in the model affect which code must be thread-safe.

- b. Within the `purchase` method use the `Thread.currentThread()` method to obtain a reference to the thread currently executing the `purchase()` method. Use a single print statement to print the name and ID of the executing thread.

```

Thread t = Thread.currentThread();
System.out.println("Thread:" + t.getName() + "," + t.getId());

```

- c. Run the project and observe the output.

7. Open the `Store` class and add a delay to the `authorizeCreditCard` method.

- `Math.random()` is used to obtain a random number in the range of 1–3, the number of seconds to delay.

```
int seconds = (int) (Math.random() * 3 + 1);
System.out.println("Sleeping for " + seconds + " seconds");
try {
    Thread.sleep(seconds * 1000);
} catch (InterruptedException e) {
    System.out.println("Interrupted");
}
```

8. Run the project multiple times. You should see a stack trace for a `java.util.NoSuchElementException`. Locate the line within the `com.example.PurchasingAgent.purchase` method that is displayed in the stack trace. The exception is being generated by the call to `store.takeShirt()`.

Note: The delay introduced in the previous step makes it more likely that concurrent `PurchasingAgent.purchase` methods calls will both believe that a shirt can be taken but take the shirt at a different time. Taking the shirt at the near the same time typically results in some of the other errors shown in step 5.

9. Use a synchronized code block to create predictable behavior.

- Modify the `purchase` method in the `PurchasingAgent` class to contain a synchronized code block.

```
synchronized (store) {
    if (store.getShirtCount() > 0 &&
        store.authorizeCreditCard("1234", 15.00)) {
        Shirt shirt = store.takeShirt();
        System.out.println("The shirt is ours!");
        System.out.println(shirt);
    } else {
        System.out.println("No shirt for you");
    }
}
```

Note: Adding `synchronized` to the method signature or using a synchronized block that uses the `this` object's monitor will not work.

10. Run the project. You should now see the desired behavior. In the output window, you should see one success message, one shirt, and one failure message.

```
Adding a shirt to the store.
Total shirts in stock: 1
The shirt is ours!
Shirt ID: 1
Description: Polo
Color: Rainbow
Size: Large

No shirt for you
```

Practice 12-2: Summary Level: Implementing a Multithreaded Program

Overview

In this practice, you will create a new project and start a new thread.

Assumptions

You have reviewed the sections covering the use of the `Thread` class.

Summary

You will create a project that slowly prints an incrementing number. A new thread will be used to increment and print the number. The application should wait for Enter to be pressed before interrupting any threads.

Tasks

1. Create a new project `ThreadInterrupted` as the main project.
 - a. Select File > New Project.
 - b. Select Java under Categories and Java Application under Projects. Click the Next button.
 - c. Enter the following information in the “Name and Location” dialog box:
 - Project Name: `ThreadInterrupted`
 - Project Location: `D:\labs\12-Threading\practices`.
 - (checked) Create Main Class: `com.example.ThreadInterruptedMain`
 - (checked) Set as Main Project
 - d. Click the Finish button.
2. Create a `Counter` class in the `com.example` package.
3. Complete the `Counter` class. The `Counter` class should:
 - Implement the `Runnable` interface.
 - Within the `run` method:
 - Create an `int` variable called `x` and initialize it to zero.
 - Construct a loop that will repeat until the executing thread is interrupted.
 - Within the loop, print and increment the value of `x`.
 - Within the loop, delay for 1 second. Return from the `run` method or exit the loop if the thread is interrupted while delayed.
4. Add the following to the `main` method in the `ThreadInterruptedMain` class:
 - Create a `Counter` instance.
 - Create a thread and pass to its constructor the runnable `Counter`.
 - Start the thread.

5. Run the project. You should see an incrementing sequence of numbers with a one second delay between each number. Notice that while the `main` method has completed the application continues to run.
6. Stop the project.
 - a. Open the Run menu.
 - b. Click Stop Build/Run.

Note: You can also stop a build/run by clicking the red square along the left side of the output window.

7. Modify the project properties to support the `try-with-resources` statement.
8. Modify the main method in the `ThreadInterruptedMain` class.
 - After starting the thread, wait for Enter to be pressed in the output window. You can use the following code:

```
try(BufferedReader br = new BufferedReader(new
InputStreamReader(System.in))) {
    br.readLine();
} catch (IOException e) {}
```

Note: You may need to fix your imports and update the project properties to support JDK 7 features.

- Print out a message indicating whether or not the thread is alive.
 - Interrupt the thread.
 - Delay for one second (to allow the thread time to complete) and then print out a message indicating whether or not the thread is alive.
9. Run the project. You should see an incrementing sequence of numbers with a one second delay between each number. Press Enter while the output window is selected to gracefully terminate the application.

Practice 12-2: Detailed Level: Implementing a Multithreaded Program

Overview

In this practice, you will create a new project and start a new thread.

Assumptions

You have reviewed the sections covering the use of the `Thread` class.

Summary

You will create a project that slowly prints an incrementing number. A new thread will be used to increment and print the number. The application should wait for Enter to be pressed before interrupting any threads.

Tasks

1. Create a new project `ThreadInterrupted` as the main project.
 - a. Select File > New Project.
 - b. Select Java under Categories and Java Application under Projects. Click the Next button.
 - c. Enter the following information in the “Name and Location” dialog box:
 - Project Name: `ThreadInterrupted`
 - Project Location: `D:\labs\12-Threading\practices.`
 - (checked) Create Main Class: `com.example.ThreadInterruptedMain`
 - (checked) Set as Main Project
 - d. Click the Finish button.
2. Create a `Counter` class in the `com.example` package.
3. Complete the `Counter` class. The `Counter` class should:
 - Implement the `Runnable` interface.
 - Within the `run` method:
 - Create an `int` variable called `x` and initialize it to zero.
 - Construct a loop that will repeat until the executing thread is interrupted.
 - Within the loop, print and increment the value of `x`.
 - Within the loop, delay for 1 second. Return from the `run` method or exit the loop if the thread is interrupted while delayed.

```

int x = 0;
while(!Thread.currentThread().isInterrupted()) {
    System.out.println("The current value of x is: " + x++);
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        return;
    }
}

```

4. Add the following to the `main` method in the `ThreadInterruptedMain` class:

- Create a `Counter` instance.
- Create a thread and pass to its constructor the runnable `Counter`.
- Start the thread.

```

Runnable r = new Counter();
Thread t = new Thread(r);
t.start();

```

5. Run the project. You should see an incrementing sequence of numbers with a one second delay between each number. Notice that while the `main` method has completed the application continues to run.
6. Stop the project.
- a. Open the Run menu.
 - b. Click Stop Build/Run.

Note: You can also stop a build/run by clicking the red square along the left side of the output window.

7. Modify the project properties to support the `try-with-resources` statement.
- a. Right-click the `ThreadInterrupted` project and click Properties.
 - b. In the Project Properties dialog box select the Sources category.
 - c. In the Source/Binary Format drop-down list select JDK 7.
 - d. Click the OK button.

8. Modify the `main` method in the `ThreadInterruptedMain` class.

- After starting the thread, wait for Enter to be pressed in the output window. You can use the following code:

```

try(BufferedReader br = new BufferedReader(new
InputStreamReader(System.in))) {
    br.readLine();
} catch (IOException e) {}

```

- Add the needed import statements.

```
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;
```

- Print out a message indicating whether or not the thread is alive.

```
System.out.println("Thread is alive:" + t.isAlive() );
```

- Interrupt the thread.

```
t.interrupt();
```

- Delay for one second (to allow the thread time to complete), and then print out a message indicating whether or not the thread is alive.

```
try {  
    Thread.sleep(1000);  
} catch (InterruptedException e) {  
}  
System.out.println("Thread is alive:" + t.isAlive());
```

9. Run the project. You should see an incrementing sequence of numbers with a one second delay between each number. Press Enter while the output window is selected to gracefully terminate the application.

Practices for Lesson 13: Concurrency

Chapter 13

NAVEEN UNNIKRIISHNAN (naveenunnikrishnan14@gmail.com)
has a non-transferable license to use this Student Guide.

Practices for Lesson 13: Overview

Practices Overview

In these practices, you will use the `java.util.concurrent` package and sub-packages of the Java programming language.

(Optional) Practice 13-1: Using the `java.util.concurrent` Package

Overview

In this practice, you will modify an existing project to use an `ExecutorService` from the `java.util.concurrent` package.

Assumptions

You have reviewed the sections covering the use of the `java.util.concurrent` package.

Summary

You will create a multithread networking client that will rapidly read the price of a shirt from several different servers. Instead of manually creating threads, you will leverage an `ExecutorService` from the `java.util.concurrent` package.

Tasks

1. Open the `ExecutorService` project as the main project.
 - a. Select `File > Open Project`.
 - b. Browse to `D:\labs\13-Concurrency\practices`.
 - c. Select `ExecutorService` and select the "Open as Main Project" check box.
 - d. Click the `Open Project` button.
2. Expand the project directories.
3. Run the `NetworkServerMain` class in the `com.example.server` package by right-clicking the class and selecting `Run File`.
4. Open the `NetworkClientMain` class in the `com.example.client` package.
5. Run the `NetworkClientMain` class package by right-clicking the class and selecting `Run File`. Notice the amount of time it takes to query all the servers sequentially.
6. Create a `NetworkClientCallable` class in the `com.example.client` package.
 - Add a constructor and a field to receive and store a `RequestResponse` reference.
 - Implement the `Callable` interface with a generic type of `RequestResponse`.

```
public class NetworkClientCallable implements
    Callable<RequestResponse>
```

 - Complete the `call` method by using a `java.net.Socket` and a `java.util.Scanner` to read the response from the server. Store the result in the `RequestResponse` object and return it.

Note: You may want to use a `try-with-resource` statement to ensure that the `Socket` and `Scanner` objects are closed.
7. Modify the `main` method of the `NetworkClientMain` class to query the servers concurrently by using an `ExecutorService`.
 - a. Comment out the contents of the `main` method.
 - b. Obtain an `ExecutorService` that reuses a pool of cached threads.

- c. Create a Map that will be used to tie a request to a future response.

```
Map<RequestResponse, Future<RequestResponse>> callables = new
HashMap<>();
```

- d. Code a loop that will create a `NetworkClientCallable` for each network request.
- The servers should be running on localhost, ports 10000–10009.
 - Submit each `NetworkClientCallable` to the `ExecutorService`. Store each `Future` in the Map created in step 7c.
- e. Shut down the `ExecutorService`.
- f. Await the termination of all threads within the `ExecutorService` for 5 seconds.
- g. Loop through the `Future` objects stored in the Map created in step 7c. Print out the servers' response or an error message with the server details if there was a problem communicating with a server.
8. Run the `NetworkClientMain` class by right-clicking the class and selecting Run File. Notice the amount of time it takes to query all the servers concurrently.
9. When done testing your client, be sure to select the `ExecutorService` output tab and terminate the server application.

(Optional) Practice 13-2: Using the Fork-Join Framework

Overview

In this practice, you will modify an existing project to use the Fork-Join framework.

Assumptions

You have reviewed the sections covering the use of the Fork-Join framework.

Summary

You are given an existing project that already leverages the Fork-Join framework to process the data contained within an array. Before the array is processed, it is initialized with random numbers. Currently the initialization is single-thread. You must use the Fork-Join framework to initialize the array with random numbers.

Tasks

1. Open the `ForkJoinFindMax` project as the main project.
 - a. Select `File > Open Project`.
 - b. Browse to `D:\labs\13-Concurrency\practices`.
 - c. Select `ForkJoinFindMax` and select the "Open as Main Project" check box.
 - d. Click the `Open Project` button.
2. Expand the project directories.
3. Open the `Main` class in the `com.example` package.
 - Review the code within the `main` method. Take note of how the `compute` method splits the data array if the count of elements to process is too great.
4. Open the `FindMaxTask` class in the `com.example` package.
 - Review the code within the class. Take note of the `for` loop used to initialize the data array with random numbers.
5. Create a `RandomArrayAction` class in the `com.example` package.
 - a. Add four fields.

```
private final int threshold;
private final int[] myArray;
private int start;
private int end;
```

- b. Add a constructor that receives parameters and saves their values within the fields defined in the previous step.

```
public RandomArrayAction(int[] myArray, int start, int end, int
threshold)
```

- c. Extend the `RecursiveAction` class from the `java.util.concurrent` package.

Note: A `RecursiveAction` is used when a `ForkJoinTask` with no return values is needed.

- d. Add the `compute` method. Note that unlike the `compute` method from a `RecursiveTask`, the `compute` method in a `RecursiveAction` returns `void`.

```
protected void compute() { }
```

- e. Begin the `compute` method. If the number of elements to process is below the threshold, you should initialize the array.

```
for (int i = start; i <= end; i++) {
    myArray[i] = ThreadLocalRandom.current().nextInt();
}
```

Note: `ThreadLocalRandom` is used instead of `Math.random()` because `Math.random()` does not scale when executed concurrently by multiple threads and would eliminate any benefit of applying the Fork-Join framework to this task.

- f. Complete the `compute` method. If the number of elements to process is above or equal to the threshold you should find the midway point in the array and create two new `RandomArrayAction` instances for each section of the array to process. Start each `RandomArrayAction`.

Note: When starting a `RecursiveAction`, you can use the `invokeAll` method instead of the `fork/join/compute` combination typically seen with a `RecursiveTask`.

```
RandomArrayAction r1 = new RandomArrayAction(myArray, start,
midway, threshold);
RandomArrayAction r2 = new RandomArrayAction(myArray, midway +
1, end, threshold);
invokeAll(r1, r2);
```

6. Modify the main method of the `Main` class to use the `RandomArrayAction` class.
- Comment out the `for` loop within the main method that initializes the data array with random values.
 - After the line that creates the `ForkJoinPool`, create a new `RandomArrayAction`.
 - Use the `ForkJoinPool` to invoke the `ForkJoinPool`.
7. Run the `ForkJoinFindMax` project by right-clicking the project and choosing *Run*.

Note: If you have a multi-CPU system you can use `System.currentTimeMillis()` to benchmark the sequential and Fork-Join solutions.

Practices for Lesson 14: Building Database Applications with JDBC

Chapter 14

Practices for Lesson 14: Overview

Practices Overview

In these practices, you will work with the JavaDB (Derby) database, creating, reading, updating, and deleting data from a SQL database by using the Java JDBC API.

Practice 14-1: Summary Level: Working with the Derby Database and JDBC


Overview

In this practice, you will start the JavaDB (Derby) database, load some sample data using a script, and write an application to read the contents of an employee database table and print the results to the console.

Tasks

1. Create the Employee Database by using the SQL script provided in the resource directory.
 - a. Open the Services window by selecting Windows > Services, or by pressing Ctrl-5.
 - b. Expand the Databases folder.
 - c. Right-click JavaDB and select Start Server.
 - d. Right-click JavaDB again and select Create Database.
 - e. Enter the following information:

Window/Page Description	Choices or Values
Database Name	EmployeeDB
User Name	public
Password	tiger
Confirm Password	tiger

- f. Click OK
- g. Right-click the connection that you created:
`jdbc:derby://localhost:1527/EmployeeDB[public on PUBLIC]` and select Connect.
- h. Select File > Open File.
- i. Browse to `D:\labs\resources` and open the `EmployeeTable.sql` script. The file will open in a SQL Execute window.
- j. Select the connection that you created from the drop-down list, and click the Run-SQL icon  or press Ctrl-Shift-E to run the script.
- k. Expand the `EmployeeDB` connection. You will see that the `PUBLIC` schema is now created. Expand the `PUBLIC` Schema and look at the table `Employee`.
- l. Right-click the connection again and select Execute Command to open another SQL window. Enter the command:
`select * from Employee`
 and click the Run-SQL icon to see the contents of the `Employee` table.

2. Open the `SimpleJDBCExample` project and run it.
 - a. You should see all the records from the `Employee` table displayed.
3. (Optional) Add a SQL command to add a new `Employee` record.
 - a. Modify the `SimpleJDBCExample` class to add a new `Employee` record to the database.
 - b. The syntax for adding a row in a SQL database is:
`INSERT INTO <table name> VALUES (<column 1 value>, <column 2 value>, ...)`
 - c. Use the `Statement executeUpdate` method to execute the query. What is the return type for this method? What value should the return type be? Test to make sure that the value of the return is correct.

Practice 14-1: Detailed Level: Working with the Derby Database and JDBC


Overview

In this practice, you will start the JavaDB (Derby) database, load some sample data using a script, and write an application to read the contents of an employee database table and print the results to the console.

Tasks

1. Create the Employee Database by using the SQL script provided in the resource directory.
 - a. Open the Services Window by selecting Windows > Services, or by pressing Ctrl-5.
 - b. Expand the Databases folder.
 - c. Right-click JavaDB and select Start Server.
 - d. Right-click JavaDB again and select Create Database.
 - e. Enter the following information:

Window/Page Description	Choices or Values
Database Name	EmployeeDB
User Name	public
Password	tiger
Confirm Password	tiger

- f. Click OK.
- g. Right-click the connection that you created:
`jdbc:derby://localhost:1527/EmployeeDB[public on PUBLIC]` and select Connect.
- h. Select File > Open File.
- i. Browse to `D:\labs\resources` and open the `EmployeeTable.sql` script. The file will open in a SQL Execute window.
- j. Select the connection that you created from the drop-down list and click the Run-SQL icon  or press Ctrl-Shift-E to run the script.
- k. Expand the `EmployeeDB` connection. You will see that the `PUBLIC` schema is now created. Expand the `PUBLIC` Schema, expand Tables, and then expand the table `Employee`.
- l. Right-click the connection again and select Execute Command to open another SQL window. Enter the command:
`select * from Employee`
 and click the Run-SQL icon to see the contents of the `Employee` table.

2. Open the SimpleJDBCExample Project and run it.
 - a. Select Windows > Projects, or press Ctrl-1.
 - b. Select File > Open Project.
 - c. Select D:\labs\12-JDBC\practices\SimpleJDBCExample.
 - d. Select "Open as Main Project."
 - e. Click OK.
 - f. Expand the Source Packages and test packages and look at the SimpleJDBCExample.java class.
 - g. Run the project: Right-click the project and select Run, or click the Run icon, or press F6.
 - h. You should see all the records from the Employee table displayed.
3. (Optional) Add a SQL command to add a new Employee record.
 - a. Modify the SimpleJDBCExample class to add a new Employee record to the database.
 - b. The syntax for adding a row in a SQL database is:
`INSERT INTO <table name> VALUES (<column 1 value>, <column 2 value>, ...)`
 - c. Use the Statement `executeUpdate` method to execute the query. What is the return type for this method? What value should the return type be? Test to make sure that the value of the return is correct.
 - d. Your code may look like this:

```

query = "INSERT INTO Employee VALUES (200, 'Bill',
'Murray', '1950-09-21', 150000)";
if (stmt.executeUpdate(query) != 1) {
    System.out.println ("Failed to add a new employee record");
}

```

Note: If you run the application again, it will throw an exception, because this key already exists in the database.

Practice 14-2: Summary Level: Using the Data Access Object Pattern

Overview

In this practice, you will take the existing Employee DAO Memory application and refactor the code to use JDBC instead. The solution from the “Exceptions and Assertions” lesson has been renamed to `EmployeeDAOJDBC`. You will need to create an `EmployeeDAOJDBCImpl` class to replace the `EmployeeDAOMemoryImpl` class, and modify the `EmployeeDAOFactory` to return an instance of your new implementation class instead of the Memory version.

You will not have to alter the other classes. This example illustrates how a well designed Data Access Object application can use an alternative persistence class without significant change.

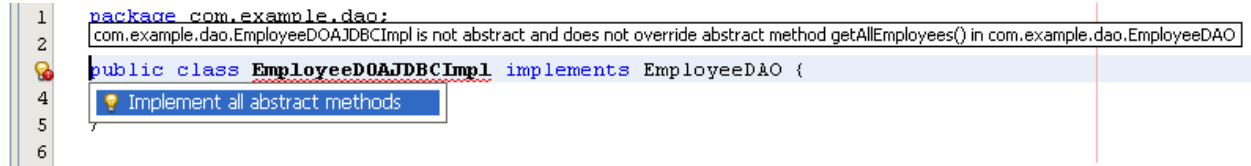
Tasks

1. Open and examine the `EmployeeDAOJDBC` project in the `D:\labs\12-JDBC\practices` folder.
 - a. In the `com.example.test` package, you see the class `EmployeeTestInteractive`. This class contains the main method and provides a console-based user interface. Through this UI, you will be able to create new records, read all the records, update a record, and delete a record from the Employee database. Note how the main method creates an instance of a Data Access Object (DAO).
 - b. In the `com.example.model` package, look at the `Employee` class. This class is a Plain Old Java Object (POJO) that encapsulates all of the data from a single employee record and row in the Employee table. Note that there are no set methods in this class, only get methods. Once an Employee object is created, it cannot be changed. It is immutable.
 - c. Expand the `com.example.dao` package. Look at the `EmployeeDAO` class and you see the methods that an implementation of this interface is expected to implement. Each of these methods throws a `DAOException`. Note that this interface extends `AutoCloseable`. Therefore, you will need to provide a `close()` method to satisfy the contract with `AutoCloseable`.
 - d. Look at the `EmployeeDAOFactory`. You see that this class has one method, `getFactory()`, that returns an instance of an `EmployeeDAOMemoryImpl`.
 - e. Look at the `EmployeeDAOMemoryImpl` class. This class is the workhorse of the DAO pattern. This is the class that the `EmployeeDAOJDBCFactory` returns as an instance from the `createEmployeeDAO` method. This is the class that you will replace with a JDBC implementation.
2. Create a new class, `EmployeeDAOJDBCImpl`, that implements `EmployeeDAO` in the `com.example.dao` package.
 - a. Note that the class has an error.

3. Implement the method signatures defined by `EmployeeDAO`.

- Click anywhere in the line that is showing an error (a light bulb with a red dot on it):

```
public class EmployeeDAOJDBCImpl implements EmployeeDAO {
```
- Press the Alt-Enter key combination to show the suggestions for fixing the error in this class. You should see the following:



Note: Your line numbers may differ from the picture shown.

- The class must implement all the methods in the interface because this is a concrete class (not abstract). You can have NetBeans provide all the method bodies by pressing the Enter key to accept the suggestion “Implement all abstract methods.”
 - You will notice that the error in the file goes away immediately, and NetBeans has provided all the method signatures based on the `EmployeeDAO` interface declarations.
 - Your next task is to add a constructor and fill in the bodies of the methods.
- Add a private instance variable, `con`, to hold a `Connection` object instance.
 - Write a package-level constructor for the class. The constructor for this class will create an instance of a `Connection` object that the methods of this class can reuse during the lifetime of the application. Be sure to catch a `SQLException`.
 - Write a method body for the `add` method. The `add` method creates a new record in the database from the `Employee` object passed in as a parameter. Recall that the SQL command to create a new record in the database is: `INSERT INTO <table> VALUES (...)`.

Note: Use single quotes for the strings and the date.

- Rethrow any `SQLException` caught as a `DAOException`.
- Write a method body for the `findById` method. This method is used by the `update` and `delete` methods and is used to locate a single record to display. Recall that the SQL command to read a single record is: `SELECT * FROM <table> WHERE <pk>=<value>`.
 - Rethrow any `SQLException` caught as a `DAOException`.
 - Write a method body for the `update` method. The `update` method updates an existing record in the database from the `Employee` object passed in as a parameter. Recall that the SQL command to create a new record in the database is: `UPDATE <table> SET COLUMNNAME=<value>, COLUMNNAME=<value>, ... WHERE <pk>=<value>`.

Note: Be sure to add single quotes around string and date values.

 - Rethrow any `SQLException` caught as a `DAOException`.
 - Write a method body for the `delete` method. The `delete` method tests to see whether an employee exists in the database by using the `findById` method, and then deletes the record if it exists. Recall that the SQL command to delete a record from the database is: `DELETE FROM <table> WHERE <pk>=<value>`.

- a. Rethrow any `SQLException` caught as a `DAOException`.
10. Write the method body for the `getAllEmployees` method. This method returns an array of `Employee` records. The SQL query to return all records is quite simple: `SELECT * FROM <table>`.
 - a. Rethrow any `SQLException` caught as a `DAOException`.
11. Write the method body for the `close` method. This method is defined by the `AutoCloseable` interface. This method should explicitly close the `Connection` object that you created in the constructor.
 - a. Rather than rethrow the `SQLException`, simply report it.
12. Save the class. Fix any missing imports and compilation errors if you have not already done so.
13. Update the `EmployeeDAOFactory` to return an instance of your new `EmployeeDAOJDBCImpl`.


```
return new EmployeeDAOJDBCImpl();
```
14. Add the JDBC Derby driver class to the project, but adding the `derbyclient.jar` file to the Libraries for the project.
 - a. Right-click the Libraries folder in the project and select Add Jar/Folder.
 - b. Browse to `D:\Program Files\Java\jdk1.7.0\db\lib`.
 - c. Select `derbyclient.jar`.
 - d. Absolute Path should be checked.
 - e. Click Open.
15. Save the updated class, and if you have no errors, compile and run the project. This application has an interactive feature that allows you to query the database and read one or all of the records, find an employee by ID, and update and delete an employee record.

Practice 14-2: Detailed Level: Using the Data Access Object Pattern

Overview

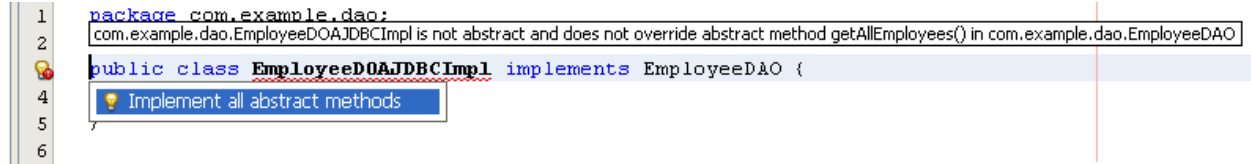
In this practice, you will take the existing Employee DAO Memory application and refactor the code to use JDBC instead. The solution from the “Exceptions and Assertions” lesson has been renamed to `EmployeeDAOJDBC`. You will need to create an `EmployeeDAOJDBCImpl` class to replace the `EmployeeDAOMemoryImpl` class, and modify the `EmployeeDAOFactory` to return an instance of your new implementation class instead of the Memory version.

You will not have to alter the other classes. This example illustrates how a well designed Data Access Object application can use an alternative persistence class without significant change.

Tasks

1. Open and examine the `EmployeeDAOJDBC` project in the `D:\labs\12-JDBC\practices` folder.
 - a. In the `com.example.test` package, you see the class `EmployeeTestInteractive`. This class contains the main method and provides a console-based user interface. Through this UI, you will be able to create new records, read all the records, update a record, and delete a record from the Employee database. Note how the main method creates an instance of a Data Access Object (DAO).
 - b. In the `com.example.model` package, look at the `Employee` class. This class is a Plain Old Java Object (POJO) that encapsulates all the data from a single employee record and row in the Employee table. Note that there are no set methods in this class, only get methods. Once an Employee object is created, it cannot be changed. It is immutable.
 - c. Expand the `com.example.dao` package. Look at the `EmployeeDAO` class and you see the methods that an implementation of this interface is expected to implement. Each of these methods throws a `DAOException`. Note that this interface extends `AutoCloseable`; therefore, you will need to provide a `close()` method to satisfy the contract with `AutoCloseable`.
 - d. Look at the `EmployeeDAOFactory`. You see that this class has one method, `getFactory()`, that returns an instance of an `EmployeeDAOMemoryImpl`.
 - e. Look at the `EmployeeDAOMemoryImpl` class. This class is the workhorse of the DAO pattern. This is the class that the `EmployeeDAOJDBCFactory` returns as an instance from the `createEmployeeDAO` method. This is the class that you will replace with a JDBC implementation.
2. Create a new class, `EmployeeDAOJDBCImpl`, in the `com.example.dao` package.
 - a. Right-click on the `com.example.dao` package and choose New Java Class
 - b. Enter `EmployeeDAOJDBCImpl` as the class name and click Finish.
 - c. Change this class to implement `EmployeeDAO`.
 - d. Note that causes an error.

3. Implement the method signatures defined by `EmployeeDAO`.
 - a. Click anywhere in the line that is showing an error (a light bulb with a red dot on it):
 - b. Press the Alt-Enter key combination to show the suggestions for fixing the error in this class. You should see the following:



Note: Your line numbers may differ from the picture shown.

- c. The class must implement all the methods in the interface because this is a concrete class (not abstract). You can have NetBeans provide all of the method bodies by pressing the Enter key to accept the suggestion "Implement all abstract methods."
 - d. You will notice that the error in the file goes away immediately, and NetBeans has provided all the method signatures based on the `EmployeeDAO` interface declarations.
 - e. Your next task is to add a constructor and fill in the bodies of the methods.
4. Add a private instance variable, `con`, to hold a `Connection` object instance.

```
private Connection con = null;
```

5. Write a constructor for the class. The constructor for this class will create an instance of a `Connection` object that the methods of this class can reuse during the lifetime of the application.
 - a. Write the constructor to use package-level access. This will enable only classes within the package to create an instance of this class (like the `EmployeeDAOFactory`).
 - b. Open the connection using the JDBC URL, name, and password from the `SimpleJDBCExample` application:

```
String url = "jdbc:derby://localhost:1527/EmployeeDB";
String username = "public";
String password = "tiger";
```

- c. In a try block (not a try-with-resources, because you want to keep this connection open until you exit the application) create an instance of a `Connection` object and catch any exception. If the connection cannot be made, exit the application.

Note: Ideally you would rather indicate to the user that the connection could not be made and retry the connection a number of times before exiting.

```
try {
    con = DriverManager.getConnection(url, username, password);
} catch (SQLException se) {
    System.out.println("Error obtaining connection with the
database: " + se);
    System.exit(-1);
}
```

6. Write a method body for the `add` method. The `add` method creates a new record in the database from the `Employee` object passed in as a parameter. Recall that the SQL command to create a new record in the database is: `INSERT INTO <table> VALUES (...)`.
- Delete the boiler-plate code created by NetBeans for the `add` method.
 - Look at the other methods in the class. They each begin by creating an instance of a `Statement` object in a `try-with-resources` statement:

```
try (Statement stmt = con.createStatement()) {
}
```

- Inside the `try` block, create a query to insert the values passed in the `Employee` instance to the database. Your query string should look something like this:

```
String query = "INSERT INTO EMPLOYEE VALUES (" + emp.getId()
    + ", '" + emp.getFirstName() + "', "
    + "'" + emp.getLastName() + "', "
    + "'" +
    new java.sql.Date(emp.getBirthDate().getTime()) + "', "
    + emp.getSalary() + ")";
```

Note the use of single quotes for the strings and the date.

- Since you are not expecting a result from the query, the appropriate `Statement` class method to use is `updateQuery`. Make sure to test to see whether the statement executed properly by looking at the integer result of the method. For example:

```
if (stmt.executeUpdate(query) != 1) {
    throw new DAOException("Error adding employee");
}
```

- At the end of the `try` block, catch any `SQLException` thrown, and wrap them in the `DAOException` to be handled by the calling application. For example:

```
catch (SQLException se) {
    throw new DAOException("Error adding employee in DAO", se);
}
```

7. Write a method body for the `findById` method. This method is used by the `update` and `delete` methods and is used to locate a single record to display. Recall that the SQL command to read a single record is: `"SELECT * FROM <table> WHERE <pk>=<value>"`.

- Delete the boiler-plate code created by NetBeans for the `findById` method.
- Create an instance of a `Statement` object in a `try-with-resources` block:

```
try (Statement stmt = con.createStatement()) {
}
```

-

Inside the try block, write a query statement to include the integer id passed in as an argument to the method and execute the query, returning a `ResultSet` instance:

```
String query = "SELECT * FROM EMPLOYEE WHERE ID=" + id;
ResultSet rs = stmt.executeQuery(query);
```

- d. Test the `ResultSet` instance for null using the `next()` method and return the result as a new `Employee` object:

```
if (!rs.next()) {
    return null;
}
return (new Employee(rs.getInt("ID"),
                    rs.getString("FIRSTNAME"),
                    rs.getString("LASTNAME"),
                    rs.getDate("BIRTHDATE"),
                    rs.getFloat("SALARY")));
```

- e. At the end of the try block, catch any `SQLException` thrown, and wrap them in the `DAOException` to be handled by the calling application. For example

```
catch (SQLException se) {
    throw new DAOException("Error finding employee in DAO", se);
}
```

8. Write a method body for the update method. The update method updates an existing record in the database from the `Employee` object passed in as a parameter. Recall that the SQL command to create a new record in the database is: "UPDATE <table> SET COLUMNNAME=<value>, COLUMNNAME=<value>, ... WHERE <pk>=<value>".

Note: Be sure to add single quotes around string and date values.

- a. Delete the boiler-plate code created by NetBeans for the update method.
b. Create an instance of a `Statement` object in a try-with-resources block:

```
try (Statement stmt = con.createStatement()) {
    ...
}
```

- c. Inside the try block, create the SQL UPDATE query from the `Employee` object passed in:

```
String query = "UPDATE EMPLOYEE "
    + "SET FIRSTNAME='" + emp.getFirstName() + "',"
    + "LASTNAME='" + emp.getLastName() + "',"
    + "BIRTHDATE='" + new
java.sql.Date(emp.getBirthDate().getTime()) + "',"
    + "SALARY=" + emp.getSalary()
    + "WHERE ID=" + emp.getId();
```

- d. You may want to test to see that the update was successful by evaluating the return value of the `executeUpdate` method:

```

if (stmt.executeUpdate(query) != 1) {
    throw new DAOException("Error updating employee");
}

```

- e. At the end of the try block, catch any `SQLException` thrown, and wrap them in the `DAOException` to be handled by the calling application. For example

```

catch (SQLException se) {
    throw new DAOException("Error updating employee in DAO",
se);
}

```

9. Write a method body for the `delete` method. The `delete` method tests to see whether an employee exists in the database by using the `findById` method, and then deletes the record if it exists. Recall that the SQL command to delete a record from the database is: "DELETE FROM <table> WHERE <pk>=<value>".

- a. Delete the boiler-plate code created by NetBeans for the `delete` method.
- b. Call the `findById` method with the `id` passed in as a parameter and if the record returned is null, throw a new `DAOException`.

```

Employee emp = findById(id);
if (emp == null) {
    throw new DAOException("Employee id: " + id + " does not
exist to delete.");
}

```

- c. Create an instance of a `Statement` object in a try-with-resources block:

```

try (Statement stmt = con.createStatement()) {
}

```

- d. Inside the try block, create the SQL DELETE query and test the result returned to make sure a single record was altered – throw a new `DAOException` if not:

```

String query = "DELETE FROM EMPLOYEE WHERE ID=" + id;
if (stmt.executeUpdate(query) != 1) {
    throw new DAOException("Error deleting employee");
}

```

- e. At the end of the try block, catch any `SQLException` thrown, and wrap them in the `DAOException` to be handled by the calling application. For example:

```

catch (SQLException se) {
    throw new DAOException("Error deleting employee in DAO",
se);
}

```

10. Write the method body for the `getAllEmployees` method. This method returns an array of `Employee` records. The SQL query to return all records is quite simple: "SELECT * FROM <table>".

- Delete the boiler-plate code created by NetBeans for the `getAllEmployees` method.
- Create an instance of a `Statement` object in a `try-with-resources` block:

```
try (Statement stmt = con.createStatement()) {
    }

```

- Inside the `try` block, create and execute the query to return all the employee records:

```
String query = "SELECT * FROM EMPLOYEE";
ResultSet rs = stmt.executeQuery(query);

```

- The easiest way to create an array of employees to return is to use a `Collection` object, `ArrayList`, and then convert the `ArrayList` object to an array. Iterate through the `ResultSet` and add each record to the `ArrayList`. In the return statement, use the `toArray` method to convert the collection to an array:

```
ArrayList<Employee> emps = new ArrayList<>();
while (rs.next()) {
    emps.add(new Employee(rs.getInt("ID"),
                          rs.getString("FIRSTNAME"),
                          rs.getString("LASTNAME"),
                          rs.getDate("BIRTHDATE"),
                          rs.getFloat("SALARY")));
}
return emps.toArray(new Employee[0]);

```

- At the end of the `try` block, catch any `SQLException` thrown, and wrap them in the `DAOException` to be handled by the calling application. For example:

```
catch (SQLException se) {
    throw new DAOException("Error getting all employees in DAO",
    se);
}

```

11. Write the method body for the `close` method. This method is defined by the `AutoCloseable` interface. This method should explicitly close the `Connection` object that you created in the constructor.

- Delete the boiler-plate code created by NetBeans for the `close` method.
- In a `try` block (you must use a `try` block, because `Connection.close` throws an exception that must be caught or rethrown), call the `close` method on the `Connection` object instance, `con`. Rather than rethrow the exception, simply report it.

```
try {
    con.close();
} catch (SQLException se) {
    System.out.println ("Exception closing Connection: " + se);
}

```

12. Save the class. Fix any missing imports and compilation errors if you have not already.
13. Update the `EmployeeDAOFactory` to return an instance of your new `EmployeeDAOJDBCImpl`.

```
return new EmployeeDAOJDBCImpl();
```

14. Add the JDBC Derby driver class to the project, by adding the `derbyclient.jar` file to the Libraries for the project.

- a. Right-click the Libraries folder in the project and select Add Jar/Folder.
- b. Browse to `D:\Program Files\Java\jdk1.7.0\db\lib`.
- c. Select `derbyclient.jar`.
- d. Absolute Path should be checked.
- e. Click Open.

15. Save the updated class, and if you have no errors, compile and run the project. This application has an interactive feature that allows you to query the database and read one or all of the records, find an employee by ID, and update and delete an employee record.

Practices for Lesson 15: Localization

Chapter 15

NAVEEN UNNIKRIISHNAN (naveenunnikrishnan14@gmail.com)
has a non-transferable license to use this Student Guide.

Practices for Lesson 15: Overview

Practices Overview

In these practices, you create a date application that is similar to the example used in the lesson. For each practice, a NetBeans project is provided for you. Complete the project as indicated in the instructions.

Practice 15-1: Summary Level: Creating a Localized Date Application

Overview

In this practice, you create a text-based application that displays dates and times in a number of different ways. Create the resource bundles to localize the application for French, Simplified Chinese, and Russian.

Assumptions

You have attended the lecture for this lesson. You have access to the JDK7 API documentation.

Summary

Create a simple text-based date application that displays the following date information for today:

- Default date
- Long date
- Short date
- Full date
- Full time
- Day of the week
- And a custom day and time that displays: day of the week, long date, era, time, and time zone

Localize the application so that it displays this information in Simplified Chinese and Russian. The user should be able to switch between the languages.

The application output in English is shown here.

```
=== Date App ===
Default Date is: Aug 1, 2011
Long Date is: August 1, 2011
Short Date is: 8/1/11
Full Date is: Monday, August 1, 2011
Full Time is: 10:13:56 AM MDT
Day of week is: Monday
My custom day and time is: Monday August 1, 2011 AD 10:13:56
Mountain Daylight Time
```

```
--- Choose Language Option ---
```

1. Set to English
 2. Set to French
 3. Set to Chinese
 4. Set to Russian
 - q. Enter q to quit
- Enter a command:

Tasks

Open the `Localized-Practice01` project in NetBeans and make the following changes:

1. Edit the `DateApplication.java` file.
2. Create a message bundle for Russian and Simplified Chinese.
 - The translated text for the menus can be found in the `MessagesText.txt` file in the `practices` directory.
3. Add code to display the specified date formats (indicated with comments) and localized text.
4. Add code to change the `Locale` based on the user input.
5. Run the `DateApplication.java` file and verify that it operates as described.

Practice 15-1: Detailed Level: Creating a Localized Date Application

Overview

In this practice, you create a text-based application that displays dates and times in a number of different ways. Create the resource bundles to localize the application for French, Simplified Chinese, and Russian.

Assumptions

You have attended the lecture for this lesson. You have access to the JDK7 API documentation.

Summary

Create a simple text-based date application that displays the following date information for today:

- Default date
- Long date
- Short date
- Full date
- Full time
- Day of the week
- And a custom day and time that displays: day of the week, long date, era, time, and time zone

Localize the application so that it displays this information in Simplified Chinese and Russian. The user should be able to switch between languages.

The application output in English is shown here.

```
=== Date App ===
Default Date is: Aug 1, 2011
Long Date is: August 1, 2011
Short Date is: 8/1/11
Full Date is: Monday, August 1, 2011
Full Time is: 10:13:56 AM MDT
Day of week is: Monday
My custom day and time is: Monday August 1, 2011 AD 10:13:56
Mountain Daylight Time
```

```
--- Choose Language Option ---
```

1. Set to English
 2. Set to French
 3. Set to Chinese
 4. Set to Russian
 - q. Enter q to quit
- Enter a command:

Tasks

Open the Localized-Practice01 project in NetBeans and make the following changes:

1. Edit the `DateApplication.java` file.
2. Open the `MessagesText.txt` file found in the `practices` directory for this practice in a text editor.
3. Create a message bundle file for Russian text named `MessagesBundle_ru_RU.properties`.
 - Right-click the project and select `New > Other > Other > Properties File`.
 - Click `Next`.
 - Enter `MessagesBundle_ru_RU` in the `File Name` field.
 - Click `Browse`.
 - Select the `src` directory.
 - Click `Select Folder`.
 - Click `Finish`.
 - Paste the localized Russian text into the file and save it.
4. Create a message bundle file for Simplified Chinese text named `MessagesBundle_zh_CN.properties`.
 - Right-click the project and select `New > Other > Other > Properties File`.
 - Click `Next`.
 - Enter `MessagesBundle_zh_CN` in the `File Name` field.
 - Click `Finish`.
 - Paste the localized Simplified Chinese text into the file and save it.
5. Update the code that sets the locale based on user input.

```

public void setEnglish() {
    currentLocale = Locale.US;
    messages = ResourceBundle.getBundle("MessagesBundle",
currentLocale);
}

public void setFrench() {
    currentLocale = Locale.FRANCE;
    messages = ResourceBundle.getBundle("MessagesBundle",
currentLocale);
}

public void setChinese() {
    currentLocale = Locale.SIMPLIFIED_CHINESE;

```

```

        messages = ResourceBundle.getBundle("MessagesBundle",
currentLocale);
    }

    public void setRussian(){
        currentLocale = ruLocale;
        this.messages =
ResourceBundle.getBundle("MessagesBundle", currentLocale);
    }

```

6. Add the code that displays the date information to the `printMenu` method.

```

        df = DateFormat.getDateInstance(DateFormat.DEFAULT,
currentLocale);
        pw.println(messages.getString("date1") + " " +
df.format(today));
        df = DateFormat.getDateInstance(DateFormat.LONG,
currentLocale);
        pw.println(messages.getString("date2") + " " +
df.format(today));
        df = DateFormat.getDateInstance(DateFormat.SHORT,
currentLocale);
        pw.println(messages.getString("date3") + " " +
df.format(today));
        df = DateFormat.getDateInstance(DateFormat.FULL,
currentLocale);
        pw.println(messages.getString("date4") + " " +
df.format(today));
        df = DateFormat.getTimeInstance(DateFormat.FULL,
currentLocale);
        pw.println(messages.getString("date5") + " " +
df.format(today));
        sdf = new SimpleDateFormat("EEEE", currentLocale);
        pw.println(messages.getString("date6") + " " +
sdf.format(today));
        sdf = new SimpleDateFormat("EEEE MMMM d, y G kk:mm:ss
zzzz", currentLocale);
        pw.println(messages.getString("date7") + " " +
sdf.format(today));

```

7. Run the `DateApplication.java` file and verify that it operates as described.

Practice 15-2: Summary Level: Localizing a JDBC Application (Optional)

Overview

In this practice, you localize the JDBC application that you created in the practices for the “Building Database Applications with JDBC” lesson.

Assumptions

You have attended the lecture for this lesson. You have completed the practices for the “Building Database Applications with JDBC” lesson.

Summary

Localize the JDBC application from the previous lesson. Identify any object that displays menu or object information and change them so that localized messages are displayed instead of static text.

Localize the application so that it displays this information in English, French, and Russian. The user should be able to switch between languages.

Tasks

You have a couple of project choices in this lab. First you can use the project files from Lesson 14, Practice 2, “Using the Data Access Object Pattern,” and just continue with that project. Alternatively, open the `Practice02` project for this lesson. Perform the following steps:

1. Open the `EmployeeTestInteractive.java` file. Examine the source code and determine which messages printed to the console should be converted into resource bundles. Notice that not all text output is included in this class file.

Note: You do not have to include error messages in the bundle. Only prompt and informational messages should be included.

2. A slight change to the user interface is required.

Currently the main interface looks like this:

```
[C]reate | [R]ead | [U]pdate | [D]elete | [L]ist | [Q]uit:
```

Changing the user interface to the following makes it easier to translate just the words in the menu.

```
[C] - Create | [R] - Read | [U] - Update | [D] - Delete | [L] -  
List | [S] - Set Language | [Q] - Quit:
```

This separates the single character commands from the words. For the solution only the words were translated. You could, of course, translate both. Notice a new option has been added to set the language.

3. Create a message bundle for English, French, and Russian.
 - The translated text for the menus can be found in the `MessagesText02.txt` file in the `practices` directory.
4. Add a `ResourceBundle` object to any object that displays menu-related information. Replace the static text with a call to the resource bundle and get the appropriate string message.

5. Examine all the date-related source code. Make sure that date information will print in the appropriate localized format.
6. When you have finished, run `EmployeeTestInteractive.java` and make sure that all the menus have been localized.
7. Additional improvements you could make:
 - Localize all the error messages in the application.
 - Localize the single character options for the main menu.

NAVEEN UNNIKRIISHNAN (naveenunnikrishnan3.14@gmail.com)
has a non-transferable license to use this Student Guide.